Rich web applications with OpenLaszlo

LASZLO in Action

Norman Klein
Max Carlson
with Glenn MacEwen

MANNING

Laszlo in Action

Laszlo in Action

NORMAN KLEIN
MAX CARLSON
with GLENN MACEWEN

MANNING
Greenwich
(74° w. long.)

For online information and ordering of this and other Manning books, please visit www.manning.com. The publisher offers discounts on this book when ordered in quantity. For more information, please contact:

Special Sales Department
Manning Publications Co.
Sound View Court 3B   fax: (609) 877-8256
Greenwich, CT 06830   email: orders@manning.com

∞ Recognizing the importance of preserving what has been written, it is Manning's policy to have the books we publish printed on acid-free paper, and we exert our best efforts to that end. Recognizing also our responsibility to conserve the resources of our planet, Manning books are printed on paper that is at least 15% recycled and processed without the use of elemental chlorine.

Manning Publications Co.
Sound View Court 3B
Greenwich, CT 06830

Copyeditor: Liz Welch
Typesetter: Gordan Salinovic
Cover designer: Leslie Haimes

ISBN 1-932394-83-4
Printed in the United States of America
1 2 3 4 5 6 7 8 9 10 – MAL – 13 12 11 10 09 08 07

To our parents
and our children who will follow us

brief contents

PART 4 INTEGRATING DHTML AND FLASH.............. 351

PART 5 SERVER AND OPTIMIZATION ISSUES 435

contents

PART 1 THE BASICS 1

PART 3 LASZLO DATASETS.................................... 249

preface

When we started Laszlo Systems in 2000, our vision was to enrich web browsers with desktop-style interaction. Just as many other software languages, such as Java, had obscure beginnings in unrelated fields, we needed a tool to program Flash-based set-top boxes. Flash's timeline-based authoring tools had proven to be too unwieldy for building large applications.

We decided that rather than build a framework to assist Flash, we'd just view the Flash Player as a virtual machine and build our own compiler to output Flash files. This approach was the genesis of Laszlo LZX.

A benefit of this compiled approach was the complete freedom to specify our own programming model, which we named LZX, to write Flash-based applications. We modeled the initial design of the LZX language on Dynamic HTML (DHTML), because we wanted the familiarity offered by tags, events, and JavaScript and the benefits provided by a declarative language approach.

We designed LZX to be a meta-language capable of being ported to multiple environments. This would allow it to take advantage of the strengths offered by each platform. Therefore, we designed the LZX language and APIs to be completely separate from Flash. Although this was a requirement from the project's inception, it took several years of development before we were ready to support other runtimes. After a lot of hard work, we launched the first commercial release of Laszlo in 2002, which targeted the ubiquitous Flash 5 Player.

OpenLaszlo was born on October 5, 2004, when we released the source code for the entire Laszlo platform under an open source license. Since then, a vibrant user community has developed around OpenLaszlo. There are user groups based in Japan, India, China, and much of Europe. We're constantly amazed at the breadth and quality of the community's contributions!

In 2007, we delivered on our initial promise of runtime independence with the release of OpenLaszlo 4. Now, a single LZX source application could be compiled to Flash, DHTML/Ajax, or Java/J2ME with Sun's Project Orbit and execute in an identical manner. OpenLaszlo 4 makes it easy to add new runtimes, so we expect to see LZX applications running on an increasingly wide variety of devices and platforms in coming years.

As an open source project with a business-friendly licensing and no ties to a specific runtime or platform, OpenLaszlo is uniquely positioned to be the de facto language for new web-based platforms; already fledging support has been established for Apple's iPhone. New projects are under way to take advantage of future platforms and provide native support for mobile devices.

Thank you for exploring the possibilities of OpenLaszlo and for purchasing this book. Norman Klein, Glenn MacEwen, and I look forward to seeing the exciting applications you create!

MAX CARLSON

acknowledgments

Creating a book isn't possible without support from a community of individuals. First, we'd like to thank Laszlo Systems for converting their server product to OpenLaszlo and contributing it to the open source community. Second, we'd like to thank Laszlo Systems for creating such an innovative technology that has arrived like a breath of fresh air to web developers everywhere. We would also like to thank P.T. Withington, Henry Minsky, Philip Romanik, and the rest of the OpenLaszlo team for diligently providing bug fixes for those rare bugs that we encountered. Next, we'd like to thank the OpenLaszlo community for its warm support. Their contributions to the OpenLaszlo platform are reflected in many of the chapters of this book.

We'd also like to thank the staff at Manning who made this book possible: Cynthia Kane, who taught us the importance of using motivations; Marjan Bace, who provided the guidance that kept us going; Ron Tomich, who handled all of the marketing efforts for this book; Karen Tegtmeyer, who organized all the peer reviews of the manuscript; Mary Piergies, who coordinated the production team's efforts to transform our rough manuscript into a finished book; Derek Lane and John Sundman, who provided technical proofreading services and guidance on technical writing issues; Liz Welch, who found our mistakes in grammar and style, and who taught us when and where to use the hyphen; and Katie Tennant, our proofreader, for her careful final pass through the book before it went to press. Thanks to all of you for your hard work, and to any others at Manning whom we may have overlooked—you made this book possible.

The authors would also like to acknowledge and thank all the people at Laszlo who provided in-depth technical information. Many of the chapters in this book strongly reflect their contributions: Ben Shine and Bret Simister for providing lots of information and examples for the animation chapter; Grig Bilham for providing the initial inspiration and design of our Laszlo Market; Adam Wolf and his "Secret Art of Science" blog that explained the mysteries of using the `initstage` attribute and also for being a tremendous source of Laszlo-related information; Elliot Winard for his assistance in understanding Laszlo's profiling utilities; and Dan Stowell for providing assistance with many of the Laszlo services. Although we have already thanked John Sundman for his technical writing contributions, he deserves additional accolades for his overall contributions to this book. He was involved in every phase of this book's development right from the start.

Others who reviewed the manuscript at different stages of development and to whom we are grateful for their invaluable feedback are Stephen Haberman, Joshua M. Woods, Andrew Glover, Edmon Begoli, Keith Tarbell, Jack Herrington, Don Hopkins, Jeff Cunningham, Bob Hutchinson, Tyler Anderson, Matt Raible, Elliot Winard, Derek Lane, Doug Warren, Robi Sen, Ryan Stewart, Tamara Jean (TJ) Fredrikson, Bob Hutchinson, and Geert Bevin.

NORMAN KLEIN

I would also like to give thanks to family, friends, and everyone in the OpenLaszlo community for their support and assistance throughout the writing process. To my friends, Wynnia Kerr and Kristy Windom, for their support and encouragement. And to my little guys, whose spirits guide my way.

MAX CARLSON

I want to thank my friends and family for helping me throughout the writing process. And thank you to my wonderful wife Hiroko—I wouldn't have been able to do it without you!

GLENN MACEWEN

My thanks go to my ever-patient wife Eva, who supported me during the writing, and many rewritings, of this book.

about this book

Welcome to *Laszlo in Action*. This book presents a comprehensive overview of the fundamentals of the Laszlo LZX language. We've taken a slightly different organizational approach than is used by most books on programming languages, where the concepts are neatly laid out in a methodical order. But we believe this isn't the way most people learn new languages; people need to be immersed in a language to gain fluency in it. So we organized our book to have an intensive set of introductory chapters, and then, at the earliest opportunity, we begin applying our rudimentary Laszlo LZX knowledge toward creating an initial prototype for a Rich Internet Application (RIA) called the Laszlo Market. Each chapter is designed both to cover the fundamentals of Laszlo LZX and to incrementally build our application.

This approach has the benefit of allowing us to first demonstrate concepts in a simple stand-alone manner, and then later apply these concepts within an application context. Because we are continually building and enhancing the work of previous chapters, this provides a scale that would not be achievable through other presentation methods. As a result, we strongly recommend that you read the chapters in consecutive order rather than skipping around.

An additional advantage of this approach is that we give you a full-featured initial working application that you can later tweak to create entirely new applications. This should save you lots of time and aggravation attempting to get many of these features working for the first time. The source code for the Laszlo Market application is available at www.manning.com/klein as well as at www.laszloinaction.com.

Who should read this book

Since Laszlo LZX was designed to be a natural extension of XHTML and JavaScript, it should be accessible to a wide audience of developers. Enterprise-level web developers accustomed to working with object-oriented languages should feel comfortable working with its blend of prototype and class-based objects. LZX is a natural evolutionary step for Ajax developers, and once they become accustomed to working with the delegate-event and data-binding communication systems, they will appreciate the corresponding productivity increase that it delivers. Although many Ajax developers are adamant about only working with open standard technologies, such as Dynamic HTML (DHTML), they are generally open to using proprietary Flash-based solutions as long as they are compartmentalized. This approach allows DHTML applications to be extended with multimedia capabilities. Because the Flash-based code is compartmentalized, it can be easily replaced with an equivalent open source solution in the future. Laszlo supports multiple platforms, and thus provides an optimum solution for creating hybrid applications. We have dedicated an entire chapter, chapter 15, to explaining how these applications can be built.

We also hope that the Laszlo Market application is compelling enough to reach beyond this immediate audience and to the larger community of DHTML web developers. We selected an online store as our sample application for many reasons. It provides a tutorial application that nicely illustrates the major features of Laszlo LZX; also, online shopping is still the killer app of the Web. Online store applications have the largest audience, produce the most revenue, and have the largest web developer community. We believe that the benefits provided by an RIA approach will result in the widespread upgrading of many existing HTML-based stores to use newer technologies such as Laszlo. Because our book delivers a working online store, DHTML developers only need to tweak the Laszlo Market to support their store's product line. In addition, the architecture of a store can be easily modified to support many other different types of applications. We're sure that developers will continually surprise us with their innovative uses for the Laszlo Market.

Roadmap

Laszlo in Action consists of 18 chapters that are divided into five parts and supplemented by two online appendixes that provide information on the supporting HTTP server applications. Each chapter's material is applied to the incremental construction of the Laszlo Market application. In the later chapters, this application is connected to an HTTP server, featured in the appendixes, to supply the application with XML data from a database. By the book's conclusion, you will have built and optimized a Laszlo online store application containing a branded appearance that can execute across both the Flash and DHTML platforms.

In the book's first part, "The basics," which covers chapters 1 through 5, we cover the preliminary Laszlo LZX skills necessary to design and create an initial prototype. In the first chapter, we provide a system-level view of Laszlo. In chapter 2, we examine the declarative and imperative architecture of Laszlo LZX applications. In the third chapter, we make the transition from an abstract to a hands-on approach to Laszlo LZX by taking a look at its language fundamentals. Chapter 4 explores the feature set of the LzView object, which serves as the superclass for all visible objects in Laszlo. By the end of chapter 5, you'll have created a functional skeleton prototype for the Laszlo Market that clearly illustrates its overall operation.

Part 2, "Prototyping the Laszlo Market," covering chapters 6 through 9, takes our initial skeleton prototype and begins to embellish it by defining the appearance of its interior screens. In chapter 6, we'll use the layout object to organize the visual appearance of a screen into a series of patterns. Chapter 7 introduces components that provide a wide variety of interface elements that solve many presentation issues. In chapter 8, we explore the publisher-subscriber communications by demonstrating the relationship between event handlers, methods, events, and attributes. In chapter 9, we look at how services are used to support user and system input functionality. Finally, we apply these services to the Laszlo Market by adding a modal login window and beginning the construction of a drag-and-drop network. This network will be continually enhanced in subsequent chapters.

Part 3, "Laszlo datasets," covering chapters 10 to 12, deals with Laszlo's approach to data handling, which is performed through its data-binding system. Chapter 10 starts by introducing resident datasets, which can contain XML data. Chapter 11 expands on the data-binding techniques introduced in the previous chapter by adding the ability to manipulate a data-binding relationship to traverse the data elements in an XML document. Chapter 12 introduces the concept of "scoreboarding" an application. This creates a central repository, in this case a dataset, that provides a set of interface methods to tabulate information. This central repository implements the Laszlo Market's shopping cart.

Part 4, "Integrating DHTML and Flash," covering chapters 13 through 15, examines usability issues. In chapter 13, we explain how animation can be used to impart a sense of physicality to an application's operation. Since this conforms to a user's experience dealing with physical objects, it makes the operation of an application appear intuitive. In chapter 14, we brand the application to have an appearance that is appropriate for its target audience. In addition, we examine the issues involved with maintaining an identical appearance across different platforms. In chapter 15, we describe the advantages of building hybrid or cross-platform applications and how they can be used to address shortcomings within a platform.

Part 5, "Server and optimization issues," covering chapters 16 through 18, integrates our Laszlo application with a back-end server, which also introduces optimization issues. In chapter 16, we show how to seamlessly transition from using resident datasets for development to HTTP-supplied datasets for production. Chapter 17 introduces a multifaceted approach to handling these optimization issues using lazy replication to control the allocation of resources for the current display and paging to redistribute the loading of data pages. Finally, in chapter 18, we extend this concept of redistributing costs to handle system optimization to reduce an application's startup time.

Two online appendices supplement this book and are available for download from the publisher's website at http://www.manning.com/klein or http://www.manning.com/LaszloinAction. To supply our Laszlo Market with XML data, appendix A contains a Java-based Struts server-side application and appendix B provides a Ruby on Rails server-side application. The purpose is to demonstrate how to provide server-side support for a Laszlo application. These appendices are not intended to be authoritative sources on either Struts or Ruby on Rails development. Please refer to other sources, such as Manning's *Struts in Action* or *Ruby for Rails*, for further information.

Code conventions

All source code in listings or in text is in a `fixed-width font` to separate it from ordinary text. We make use of several different programming languages—JavaScript, Java, and Ruby—as well as markup languages—XML, HTML, and CSS—in this book, but have maintained a consistent approach to their usage. `This fixed-width font` is used for method and function names, object properties, XML elements, and attributes in text.

We have used a system of code annotations to provide explanatory assistance and highlight important concepts. Several styles of code annotations are used: numbered bullets link to explanations that directly follow the listing, and arrows are used to indicate in-line explanations.

Code downloads

Source code for all of the working examples in this book is available for download from http://www.manning.com/klein or http://www.manning.com/LaszloinAction.com.

Code listings are divided into two categories: a code listing that builds the Laszlo Market on a chapter basis and a code listing for the individual examples used in each chapter.

Installing the Laszlo Market application requires that several servers, in addition to the OpenLaszlo server, be installed and configured. Since the installation procedures for new versions of these servers are continually being updated, and are already extensively covered by their online documentation, we suggest visiting these URLs for this information:

- Tomcat: http://tomcat.apache.org
- OpenLaszlo: http://www.openlaszlo.org/documentation
- Struts: http://struts.apache.org/
- Ruby on Rails: http://www.rubyonrails.org/docs
- Red5: http://osflash.org/red5
- MySQL: http://dev.mysql.com/doc

For those readers who don't have a suitable hardware environment to support these servers or don't have an adequate amount of product data, we have created an operational Laszlo Market at the http://www.laszloinaction.com website. This website also features a number of useful links and resources to assist the development of your own store.

If you decide to install the Laszlo Market in your local environment, please be aware that it's also necessary to copy the zipcodevalidator.lzx file into the $LPS/lps/components/validators directory and then update its library.file to include it. Further information on this procedure is included in chapter 7. If you encounter any problems with this local configuration, the Laszlo forums are probably the best place to seek further information and assistance.

Author Online

Purchase of *Laszlo in Action* includes free access to a private web forum run by Manning Publications where you can make comments about the book, ask technical questions, and receive help from the authors and from other users. To access the forum and subscribe to it, point your web browser to www.manning.com/LaszloinAction or www.manning.com/klein. This page provides information on how to get on the forum once you are registered, what kind of help is available, and the rules of conduct on the forum.

Manning's commitment to our readers is to provide a venue where a meaningful dialogue between individual readers and between readers and the authors can take place. It is not a commitment to any specific amount of participation on the part of the authors, whose contribution to the book's forum remains voluntary (and unpaid). We suggest you try asking the authors some challenging questions, lest their interest stray!

The Author Online forum and the archives of previous discussions will be accessible from the publisher's website as long as the book is in print.

about the title

By combining introductions, overviews, and how-to examples, the *In Action* books are designed to help learning and remembering. According to research in cognitive science, the things people remember are things they discover during self-motivated exploration.

Although no one at Manning is a cognitive scientist, we are convinced that for learning to become permanent it must pass through stages of exploration, play, and, interestingly, re-telling of what is being learned. People understand and remember new things, which is to say they master them, only after actively exploring them. Humans learn in action. An essential part of an *In Action* book is that it is example-driven. It encourages the reader to try things out, to play with new code, and explore new ideas.

There is another, more mundane, reason for the title of this book: our readers are busy. They use books to do a job or solve a problem. They need books that allow them to jump in and jump out easily and learn just what they want just when they want it. They need books that aid them in action. The books in this series are designed for such readers.

about the cover illustration

The figure on the cover of *Laszlo in Action* is called “The Coast Guard.” The illustration is taken from a French travel book, *Encyclopedie des Voyages* by J. G. St. Saveur, published in 1796. Travel for pleasure was a relatively new phenomenon at the time and travel guides such as this one were popular, introducing both the tourist as well as the armchair traveler to the inhabitants of other regions of the world, as well as to the uniforms and costumes of French soldiers, civil servants, tradesmen, merchants, and peasants.

The diversity of the drawings in the *Encyclopedie des Voyages* speaks vividly of the uniqueness and individuality of the world’s towns and provinces just 200 years ago. This was a time when the dress codes of two regions separated by a few dozen miles identified people uniquely as belonging to one or the other. The travel guide brings to life a sense of isolation and distance of that period and of every other historic period except our own hyperkinetic present.

Dress codes have changed since then and the diversity by region, so rich at the time, has faded away. It is now often hard to tell the inhabitant of one continent from another. Perhaps, trying to view it optimistically, we have traded a cultural and visual diversity for a more varied personal life. Or a more varied and interesting intellectual and technical life.

We at Manning celebrate the inventiveness, the initiative, and the fun of the computer business with book covers based on the rich diversity of regional life two centuries ago brought back to life by the pictures from this travel guide.

Part 1

The basics

Part 1 of this book introduces enough of Laszlo to present a lightweight functional model of a real-world application, the Laszlo Market online store. It moves through a crash course in the fundamentals of Laszlo's XML-based LZX language, focuses on its declarative tags, and continues with the core language rules illustrated with a small Laszlo application incorporating important features such as methods, event handlers, attributes, JavaScript libraries, and constraints. It concludes with an introduction to the most important LZX building block, the `LzView` class, the superclass of all visible objects. We explore the visibility control, multimedia resources, the layout control, user event handling, and the relationship between LZX and JavaScript classes. They are all brought together in the Laszlo Market prototype, to be refined further in the remaining chapters.

1 Turbocharging web technology

This chapter covers

- Achieving Web 2.0 goals with RIAs
- Choosing Laszlo
- Animating a “Hello World” application
- Executing on Flash or DHTML

> *Most people make the mistake of thinking design is what it looks like. That's not what we think design is. It's not just how it looks and feels. Design is how it works.*
>
> —Steve Jobs, Apple Computer

The last decade has seen the explosive penetration of HTML-based web applications into our daily routine: you can book a flight, get driving directions, or purchase goods, all on the Web. Although this initial wave of applications has been wildly successful, the next oncoming wave, known as *rich Internet applications* (RIAs), should prove to be even more pervasive. While HTML-based applications have been limited by their static page-orientation nature, RIAs enjoy a fluidity rivaling the operation of desktop applications.

You might wonder what distinguishes an RIA from an HTML-based application. With HTML pages, there is a master-slave relationship, where processing is performed on a server and a client browser is only used to display static content. This results in application state having to be stored on the server. So each step requires a round-trip to the server to advance the application state. This synchronized communication keeps the browser operating in lockstep with the server.

RIAs advance this design by adding a data cache to the browser, allowing it to maintain its own sense of state and operate as an independent client. This enables the RIA to offer a richer and more responsive interface to users. This richer functionality includes any client services such as advanced windowing components, drag-and-drop services, vector-based graphics, audio, and video. Since there is no need to communicate to a server, an RIA can present these services in a more responsive manner. Together, this provides performance matching desktop applications.

The promise of RIAs is so great that they don't simply expand the capabilities of the Web but rather move it close to Tim Berners Lee's original vision for it. Industry pundits have labeled this movement as Web 2.0. Although many of the objectives of Web 2.0 are still being debated, there is general agreement as to the importance of achieving these goals:

- Web-based applications need to be as interactive as desktop applications.
- Application development shouldn't be limited to large development teams.
- The hypertext-based heritage of the Web needs to be preserved.
- Democratization of the Web should empower users.

Like the other RIA systems that form Web 2.0, OpenLaszlo is designed to meet these goals. But while OpenLaszlo is an example of an RIA, it can also be viewed as an RIA architectural approach. As a result, the OpenLaszlo system isn't tied to a particular platform such as Dynamic HTML (DHTML), Flash, or Java, but instead can be applied across all these platforms. OpenLaszlo is able to operate across differing platforms by its adherence to open standards. In a sense, OpenLaszlo can be viewed as the Unix of RIA systems, since it is designed to be ported to other platforms, such as Microsoft's Silverlight, as they become widely available. With minor exceptions, the same code base can be executed on any of these platforms and will produce identical display.

Even better, its approach allows hybrid applications to be created that simultaneously execute across multiple platforms. This allows critical shortcomings within one platform to be addressed by another. In particular, it allows Flash-based applications to be searchable by web crawlers, enables DHTML-based applications to access multimedia resources such as audio and video, and establishes a competitive landscape that allows the best component implementations to be selected.

In this chapter, we'll try to anticipate some of your questions about Laszlo with some background about the Laszlo system, an introduction to Laszlo's XML-based language LZX, a short discussion about deploying Laszlo applications, and an explanation of the roles of Flash and DHTML in Laszlo.

1.1 Laszlo is for designing and building RIAs

The sole purpose of Laszlo is to support the design and construction of web applications with the performance of desktop software. Laszlo removes many of the barriers separating developers and designers. In the past, these collaborative partners were forced to exist in different worlds, like the left and right hemispheres of the brain. But Laszlo redefines these relationships, so developers can participate in design issues such as usability and the role of emotion. These issues shouldn't be viewed as posing new problems for developers, but rather as providing a richer palette for expressing their creative visions.

A major viewpoint espoused within this book is that effective GUI development is driven by the needs of users. It's not enough to pay lip service to design; users and design specialists need to be an integral part of a development process. This philosophy, known as *user-centered design*, serves as the impetus for all development-related activities. The principles of user-centered design are illustrated through the development of a Laszlo application, the Laszlo Market, to sell online action videos. This application incorporates material from each chapter to demonstrate all facets of Laszlo development. In this chapter, we'll begin with a rationale for choosing Laszlo. Finally, we'll take a short look at Laszlo's history and its open source copyright license.

1.1.1 User-centered design

User-centered design places the user at the center of the design process. It focuses on how users interact with a product. The product could be anything from a potato peeler to the latest software application. The focus is on two main issues: usefulness and usability. Usefulness relates to applicability; how well does a product match a user's needs? Usability relates to ergonomic issues; is using the product intuitive, and does its use bring pleasure or satisfaction? Meeting these requirements determines the overall success of a design.

This emphasis differs from conventional design strategies that focus on maximizing performance. This generally results in a minimal user interface, since an intuitive interface requires resources that might degrade performance. User-centered design decouples an interface from hardware efficiency concerns. In fact, an undue concern with system efficiency doesn't necessarily lead to higher productivity. Consider this example: "What's the quickest way to heat a cup of water in a microwave: for one minute and 10 seconds, or for one minute and 11 seconds?" The answer is the latter (you knew this was a trick question, didn't you?) because it's faster to punch in a sequence of ones without searching for the zero. The lesson is that in many cases user productivity is more important than hardware productivity.

Additionally, there's a certain level of user satisfaction in pounding out a sequence of ones that isn't delivered by searching for a zero. The action of pounding out those ones provides a user with a psychological sense of empowerment, instead of the frustration of searching. Suddenly, heating a cup of water is an enriching experience rather than work. This illustrates the principle of emotional design: the importance of making interfaces fun to use.

1.1.2 Discovering Laszlo: a developer's tale

We believe the best way to learn and understand Laszlo LZX is within an application context, so over the course of the book, we'll build a real-world web application called the Laszlo Market. We'll also use its development to illustrate the benefits of a user-centered approach. We present this "developer's tale" to justify the need for this application and to show why Laszlo is the most appropriate solution for implementing this application.

The problem with current web applications

You have landed a dream assignment to reimplement an existing online store with your choice of implementation technologies, including Java 2 Enterprise Edition (J2EE or, more recently, JEE) tag libraries, any flavor of Ajax, or another RIA language. The current store is plagued by poor sales, so the owners want a complete rebuild. But before you can select a technology, it's necessary to first investigate the reasons for the existing store's failure.

After some research, its problems become apparent. From the sales department, you learn that the store consistently failed to meet sales expectations. What's worse, customers gave it a low positive impression rating, commenting that it's too similar to other stores.

Your next stop is marketing, where you learn that although the store has a strong supply of customers, your customers' stay is often short, with few interactions. In other words, their *stick time* is low and shallow. Even more alarming, 45 percent of customers abandon their shopping cart in the midst of a purchase. The problem is starting to become clear; the store is boring and overly complicated.

The goal of all online stores is identical: to connect with prospects, convert them to customers, and retain them as satisfied clients. To accomplish this, you must understand their needs. For our store to be successful, we must achieve these goals through the following steps.

Talking to the customer

When a product fails, generally its root cause is insufficient communication with end users. Failure to interview end users results in an application that appeals to only a select audience. Competing online stores offer similar features and prices. To get an edge, usability is an obvious goal. But since stick time and abandonment are *the issues*, we also want to elicit an emotional response from users. This creates a positive impression among users so they'll associate a favorable feeling with visiting our store. So first, a *tone* needs to be established for the store. Since the electronic media demographic—and your group of interviewees—consists largely of males in the 20- to 30-year-old age group, your interviews point you to a "high-tech/comic book" tone.

Although you don't want to focus on specific issues too early, the cart abandonment issue is financially important enough to warrant special consideration. So you ask your users the reasons they might abandon a cart.

Reducing shopping cart abandonment

Your interviews produce a remarkably common response: the purchasing process is too complicated. The current store is HTML-based, so purchases require the seven steps shown in figure 1.1. Each step incurs a round-trip to the server, presenting another opportunity to abandon the purchase. Worst of all, correcting previously entered information is clumsy and the Back button only adds additional confusion. You sadly agree that there is little in the current store to inspire trust from customers.

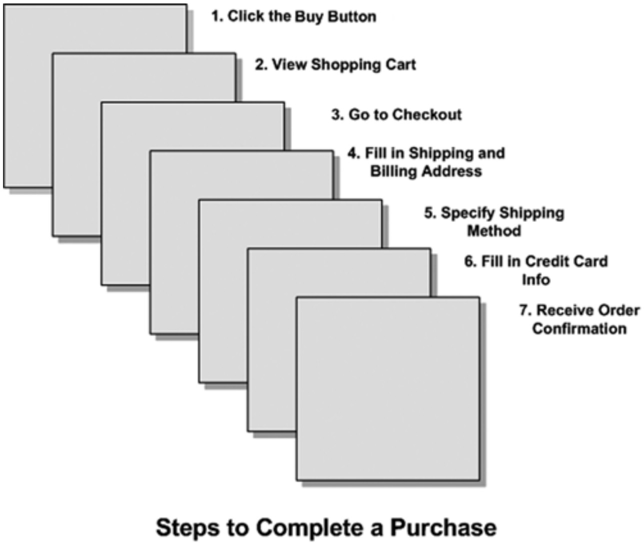

1. Click the Buy Button
2. View Shopping Cart
3. Go to Checkout
4. Fill in Shipping and Billing Address
5. Specify Shipping Method
6. Fill in Credit Card Info
7. Receive Order Confirmation
Steps to Complete a Purchase

Figure 1.1
The shortcoming of an HTML-oriented approach is clearly demonstrated with the shopping cart mechanism in current online stores. It requires a customer to endure a seven-step process, with each step requiring a round-trip journey to the server. This is a major cause of shopping cart abandonment rates that can reach up to 75 percent.

Global Millennia Marketing recently conducted a survey asking customers for the top five reasons they might abandon a shopping cart (www.imscart.com/ecommerce_software_article_1.htm):

- High shipping cost (69 percent)
- Change of mind (61 percent)
- High total cost (49 percent)
- Long checkout process (41 percent)
- Poor site navigation and long download times (31 percent)
- Confusing checkout process (27 percent)

Since high prices and shipping costs are beyond our control, we'll focus on the remaining issues. The goal is to make the purchase process seamless, as outlined in these steps:

- Assemble a shopping cart
- Enter shipping and billing addresses
- Enter credit card information
- Specify a shipping method

There is clearly too much information for one screen. Any workable solution requires that state be stored on the client. This issue eliminates all the J2EE tag-based solutions such as Tapestry, JavaServer Faces (JSF), and JSP Standard Tag Library (JSTL) as candidates, since they require state be stored on the server. Since the Ajax and RIA technologies can maintain state in the browser's local data cache, you decide to look more closely at Ajax and RIAs.

Ajax and RIAs supply high-level components, such as *tabs* and *tabsliders*, to store multiple input fields within subscreens. But even with components, there's still too much information. It's becoming clear that for visual continuity, *multiple virtual screens* will be needed.

From your past experience with user interfaces, you conclude that animation provides the best solution for coordinating the display of subscreens into a single virtual screen. But you don't welcome this conclusion; animation is complex, requiring lots of complex coding. You search for an alternative among the newer technologies: something called *declarative notation.*

The benefits of declarative notation

You know from previous experience that attempting to satisfy the project requirements with a procedural solution will likely result in an unacceptable budget and schedule. With so many elements and actions involved, the coding requirements can quickly become overwhelming.

What's really needed is a shorthand system that only requires a description of *what* needs to occur, without requiring the instructions for *how* to implement it. This is the premise of declarative programming. It only requires an application to be described, and then an outside system is responsible for implementing it. For a more complete discussion of declarative programming, have a look at http://en.wikipedia.org/wiki/Declarative_programming.

Laszlo reads a declarative description of a user interface and instructs a client engine (Flash, browser, or a Java Virtual Machine [JVM]) to render it. This is the same way that HTML and browsers work. HTML markup code can specify a table with a certain number of columns and rows, but unlike a programming language, it contains no instructions on how to build it; the web browser builds it using its own internal rules for constructing tables. Laszlo takes this to a higher level with its declarative LZX language, allowing complex dynamic GUI structures to be specified.

A declarative language requirement removes most Ajax toolkits from consideration. Almost all current Ajax toolkits are based on imperative JavaScript libraries. The few fledging declarative Ajax initiatives, such as XForms and XAP, are too immature compared to the other RIA technologies such as Laszlo and Flex. So now our choices have been whittled down to the RIA technologies.

Examining RIA technologies

The problem with most RIA solutions is they require a special browser plug-in, rather than running natively in the browser. A browser plug-in executes as a separate application within the browser—a separate executing virtual machine with its own state. The biggest problem facing a plug-in is that it must be ubiquitous—installed everywhere. But Adobe's Flash has pretty well resolved that problem by having a penetration rate of about 95 percent. It has also evolved into a rich target platform with vector graphics, downloadable fonts, and video and audio capabilities, which complement the features of a browser.

So Flash shows promise. But in the competitive online store world, you know that those missing 5 percent of users are important. For certain markets, those 5 percent of users can represent a significant percentage of your customers. You conclude that a DHTML version must also be available to reach those users. Furthermore, you don't have staff to maintain two separate source code versions; a single version needs to be executable across both platforms.

The other RIA candidates, Adobe's Flex and Microsoft's XAML (Extensible Application Markup Language), are limited to their respective proprietary plug-in environments. Since they both have a vested interest in promoting their plug-in environments, neither is likely to support the DHTML market anytime soon.

Finally, your budget is limited, so cost is important. But Laszlo is open source; there's no cost in downloading and using their server. You've found your solution! Only Laszlo meets your requirements!

As you call your manager's office to see if she's available, you imagine her first question. What's this Laszlo? Perhaps you had better do some historical digging before you see her. Here's what you find.

1.1.3 A bit of history

Laszlo Systems set out to build an RIA platform with the critical features lacking in current web technologies. Initially attracted to the Flash Player, the Laszlo founders were frustrated by its limitations and the difficulty of effectively using its authoring tools. At the same time, web browsers of that era did not provide the level of rich user experience possible in Flash. Consequently, it became evident that what was needed was a standards-based language that could build on the richness and ubiquity of the Flash Player, utilize existing developer skills, and still mesh with traditional web infrastructure and back ends.

Laszlo Systems grew out of a project started by engineers David Temkin, Bret Simister, and Max Carlson at Excite@Home. It was whimsically named after a Hungarian cinematographer, László Kovács, who filmed the movie *Easy Rider.* In 2000, work on the Laszlo Presentation Server (LPS) was begun. The first general release of LPS occurred in 2002, intended for deployment as a Behr Paint web application. At this time, Laszlo Systems business was oriented toward selling LPS as a closed source server.

In 2004, version 2.2 of the entire Laszlo platform became open source as OpenLaszlo, the purpose of which is to establish a free and open platform for RIA development and deployment. Laszlo Systems has shifted their business focus from selling closed source software to selling services around their open source software.

1.1.4 OpenLaszlo: open source and available to all

Laszlo is now available as OpenLaszlo, a free open source technology. By adopting an open source model, Laszlo Systems hopes to accelerate the adoption of RIAs and to generate community involvement in enhancing Laszlo's quality, performance, and features. The company continues to lead development of the platform, providing customers with application modules, support, and education programs.

SHOULD YOU SAY LASZLO OR OPENLASZLO?

Laszlo consists of the OpenLaszlo server, containing a compiler and runtime libraries, and the Laszlo LZX language. In this book, we'll use the term *OpenLaszlo* when referring specifically to the server or software distribution or in an abstract sense, and the term *Laszlo* or *LZX* when referring to its declarative programming language.

The OpenLaszlo website at www.openlaszlo.org is the place to start learning about Laszlo, the Laszlo community, how to download and install the software, and the terms of the OpenLaszlo license.

ABOUT THE OPENLASZLO LICENSE

OpenLaszlo is available as open source under CPL, the same open source licensing agreement used by IBM for the Eclipse platform. CPL, a *copyleft* license broadly similar to the GNU General Public License (GPL), is the reverse of a copyright, ensuring that any person receiving a copy, or a derived version of a work, can use, modify, and redistribute both the work and any derived versions.

We recommend that readers download and install the OpenLaszlo software. All the chapters feature short illustrative examples. Your *Laszlo in Action* learning experience will be greatly enhanced by executing and viewing these examples on your browser. These examples are available for download at www.manning.com/klein (click on Source Code), or can be manually entered through your favorite text editor. Have fun; we hope you enjoy the ride!

1.2 A first taste of Laszlo LZX

LZX is OpenLaszlo's declarative XML-based language. Since we believe that examples are the best way to learn a language, we'll start by demonstrating how to create a "Hello World" program. Afterward, we'll add some spice and provide a preview of LZX's animation features.

Every Laszlo application is contained within an XML document delimited by the `canvas` tag. This tag serves as the root node and as a parent container for all other elements in an application. It also defines the output space used to display the application. A canvas isn't an abstract entity; it has concrete attributes to define its appearance. For example, it has size attributes, `width` and `height`, specifying the initial screen size for the application. Its background color attribute, `bgcolor`, specifies just that: the background color. If no background color or image resource is specified, the background is transparent.

Let's start by creating a canvas with a height of 200 pixels and a light gray background. XML syntax is used to create a single-line statement:

```
<canvas width="400" height="200" bgcolor="0xBDDDF0"/>
```

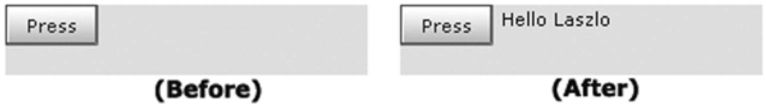
A background color is specified through a simple color name—red, blue, green, or yellow—or, as we have done here, with a hexadecimal RBG value. Congratulations, you have just created your first Laszlo application consisting of a translucent blue screen.

Now that we have a canvas, we can display something in it. Laszlo contains a large selection of standard components—buttons, text, and others—so we have lots of choices for displaying the obligatory "Hello Laszlo" message. We have selected a `button` tag which, when clicked, triggers the display of a `text` object containing the "Hello Laszlo" message:

```
<canvas width="300" height="200" bgcolor="0xBDDDF0">
   <button text="Press" onclick="parent.msg.setText("Hello Laszlo")"/>
   <text name="msg" x="60"/>
</canvas>
```

Running this example produces the screenshot shown in figure 1.2.

Since Laszlo allows us to be more creative, let's make our first Laszlo program signal its arrival by shouting "Hello Laszlo" to the heavens at the top of its lungs.



(Before) (After)

Figure 1.2 Clicking the button displays the "Hello Laszlo" message.

1.2.1 Animating "Hello Laszlo"

Here's an opportunity to don our designer thinking cap. What's the best way to amplify the "Hello Laszlo" message so everyone in the room sees it? One way is to encase our message in an attractive window and animate it to blast out onto the screen. To ensure that no one misses it, let's pulse it between its fully expanded and contracted states. Figure 1.3 shows our pulsing message, just the way Hollywood might do it.

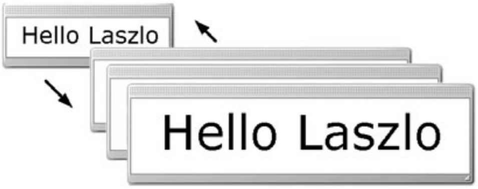


Figure 1.3 The pulsating "Hello Laszlo" example, shown in listing 1.1, uses constraints to create a relationship between the size of the text and the size of the window.

Listing 1.1 shows how to create the pulsing message in figure 1.3. Everything is specified declaratively, so no JavaScript coding is needed. These declarative statements describe to Laszlo *what* we want to see. Laszlo worries about *how* to implement it.

Listing 1.1 The extended "Hello Laszlo" program

```
<canvas>
   <window resizable="true" width="250" height="100">
      <text id="msg" text="Hello Laszlo"
            resize="true" align="center"
            fontsize="${immediateparent.width/7}"
            height="${immediateparent.height}"/>
      <animatorgroup process="sequential" repeat="Infinity">
         <animatorgroup process="simultaneous">
            <animator attribute="width" to="500" duration="1500"/>
            <animator attribute="height" to="150" duration="1500"/>
         </animatorgroup>
         <animatorgroup process="simultaneous">
            <animator attribute="width" to="250" duration="1500"/>
            <animator attribute="height" to="100" duration="1500"/>
         </animatorgroup>
      </animatorgroup>
   </window>
</canvas>
```

❶ **Defines text object**

❷ **Specifies constraints**

❸ **Defines animator groups**

In adding this pulsing action to the "Hello Laszlo" example, we specify a resizable `window` containing a `text` object ❶, whose "Hello Laszlo" text string is centered in the window. To make the effect realistic, the expansion and contraction of the window and text objects must be coordinated. This requires that two relationships be maintained: first, between the text's font size and the width of the inner portion of the window; and second, between the text's height and the height of the inner portion of the window. These relationships are known as *constraints* ❷.

Finally, three *animator groups* ❸ are applied to the window. The first animator group controls the other two, so they execute in a sequential order that repeats indefinitely. The window's `width` and `height` attributes are simultaneously increased from 250 to 500 and 100 to 150 pixels over a period of 1.5 seconds. After the expansion, these attributes decrease back to their original values over the same period of time, thus providing the effect that the window is pulsating its "Hello Laszlo" message.

1.2.2 Executing on Flash or DHTML

This same set of source code can be used to invoke the "Hello Laszlo" message for either the Laszlo Flash or DHTML platform. The platform can be set at runtime by setting the URL parameter `lzr` to either `flash` or `dhtml`. Depending on your configuration, the URL for your "Hello Laszlo" application might look like one of the following:

Flash version:

```
http://localhost:8080/lps/my-apps/hellolaszlo.lzx?lzr=swf7
```

DHTML version:

```
http://localhost:8080/lps/my-apps/hellolaszlo.lzx?lzr=dhtml
```

Because no procedural coding is involved, only declarative specification, it took one of the authors only a couple of minutes to write and test this application. Writing this application in a procedural language would be a significantly more involved exercise. Now that we have your first application, an obvious next question concerns deployment.

1.3 Deploying a Laszlo application

While other web applications must be deployed on either a web server or an application server, a Laszlo application has the flexibility to be deployed on either. This flexibility extends to deployment from an email attachment or from a packaged CD-ROM distribution. These options allow a Laszlo application to be deployed in whatever way best fits the situation.

1.3.1 Server mode

All Laszlo applications are initially created and deployed through the OpenLaszlo Server (see figure 1.4) to deliver Shockwave Flash (SWF) or DHTML JavaScript files. The OpenLaszlo Server is a Java servlet that executes in a Tomcat servlet container or J2EE application server. Whenever an application's LZX source files are updated,

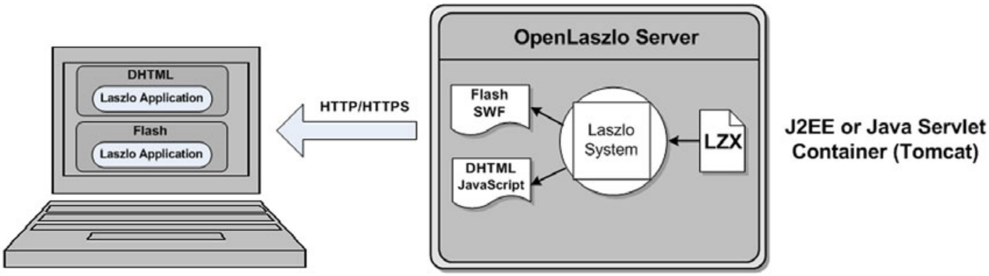


Figure 1.4 The OpenLaszlo Server is a Java-based servlet executing in a J2EE or Java servlet container. It compiles an LZX source file or set of files into a single Flash SWF or JavaScript executable file that is transmitted to a browser for execution.

the server recompiles the application to create a new Flash SWF or DHTML JavaScript file.

Once application development has completed, a range of options are available for deployment. In the following section, we'll examine these deployment options and application types that might be most suitable for a particular situation.

1.3.2 Stand-alone mode

An OpenLaszlo application can be deployed in *stand-alone mode* or, as it's officially called, Stand-alone OpenLaszlo Output (SOLO) mode. This allows web applications to be served on the Internet through a low-cost web server (Microsoft IIS, Apache, and others), sent as an email attachment, or just burned onto a CD and mailed, as shown in figure 1.5.

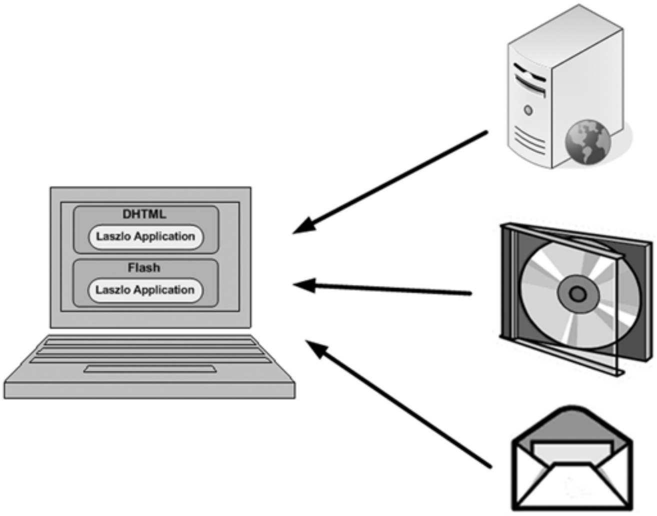


Figure 1.5 A Laszlo application created in SOLO mode offers the ultimate in transportability. Since it is contained in either a Flash SWF or DHTML embed.js file, it can be deployed on any web server and interface to any server-side code (CGI, PHP, .NET, JSP, ASP), opened as an email attachment, or opened directly in a web browser.

Creating a SOLO version of a Laszlo application is simple: just set the `canvas` tag's `proxied` attribute to false:

```
<canvas proxied="false">
```

Or it can be set within the URL query string:

```
http://localhost:8080/lps/my-apps/hellolaszlo.lzx?proxied=false
```

In either case, the OpenLaszlo Server is instructed to create a stand-alone version of your Laszlo application. The completed application is still a Flash SWF or DHTML JavaScript file, but now all linkages to the OpenLaszlo Server have been severed and the application is free to be transported and deployed through any of the delivery systems shown in figure 1.5.

Any Laszlo application can be deployed SOLO, except where data integration via Simple Object Access Protocol (SOAP), Extensible Markup Language–Remote Procedure Call (XML-RPC), or Java-RPC is required. Applications requiring such services must use J2EE deployment. However, these are relatively specialized technologies, so the vast majority of Laszlo applications can be created in SOLO mode.

1.4 Summary

This chapter emphasized the importance of user-centered design and how Laszlo encourages a workflow process whereby user input drives development. Unlike many other languages, which have been adapted for specialized tasks that they are ill-suited to handle, the sole purpose of Laszlo LZX is to create GUIs. Laszlo can be considered both a language, LZX, and an architectural approach for building web-based applications. Since Laszlo is open source, it doesn't lock users into a particular platform, instead espousing a "best of breed" approach. Users are free to select their preferred platform: Flash or DHTML.

Before we begin work on the Laszlo Market application in chapter 5, the next three chapters provide a crash course in the basics of LZX; the second chapter provides examples highlighting features of LZX declarative statements; the third chapter provides a guide to the declarative/procedural interface; and the fourth chapter provides an overview of the capabilities of view-based objects. Finally, in chapter 5, we tackle the Laszlo Market design.

2

The declarative world of LZX

This chapter covers

- Advantages of declarative notation
- Parent-child attribute propagation
- Built-in event handling and constraints
- Data binding to XML elements

> *If our basic tool, the language in which we design and code our programs, is also complicated, the language itself becomes part of the problem rather than part of its solution.*
>
> —C.A.R. Hoare, *The Emperor's Old Clothes*,
> Turing Award Lecture, 1980

In this chapter, we'll focus on the importance of declarative notation for building large complex RIAs. Declarative notation allows "what is to occur" to be specified rather than "how to *make* it occur." This is the difference between telling an auto mechanic to "adjust the valves" rather than having to write a manual on valve adjustment. As you can imagine, this saves a lot of programming work—although at first glance XML might seem an odd and overused choice for a programming language such as LZX. But in this chapter, we'll examine the characteristics of XML that make it the ideal choice for constructing RIAs. If XML didn't already exist, then RIA developers would have had to invent it.

We'll see how XML's hierarchical nature is perfectly suited for declaratively describing the layout of graphical elements. It supports two different types of hierarchies, parent-child and sibling, that describe offset and flat relationships. Later you'll learn how these relationships can be used to create a concise notation to easily describe a physical configuration.

In addition, XML's structure can be enriched. Much of LZX's declarative magic is not performed directly by individual tags, but rather by operations performed on its tags. This leverages its declarative nature to function within a context. This context can be either a horizontal relationship with neighboring objects, or a vertical relationship with its current application state represented in an XML dataset. We'll see how Laszlo embeds two different communication systems in its declarative notation for objects to establish this bidirectional orientation.

In this chapter, we'll illustrate these declarative features by creating a jet formation whose members move relative to one another, depending on the application state. By the conclusion of our example, our jet formation will be ready to fly. Best of all, this is accomplished with declarative tags and no JavaScript coding.

2.1 Architectural support

Most declarative languages are *domain-specific languages (DSLs)*, designed for a specific application domain. For example, HTML is a DSL whose domain is the world of web pages. A limitation of DSLs, and declarative languages in general, is the fact that they aren't *Turing complete,*[1] since there are certain tasks they can't specify. For example, HTML statements can't process input events. For this reason, most declarative languages are combined with a general-purpose language—in HTML's case, JavaScript—to provide a complete solution.

This combined approach yields a development strategy whereby a DSL is used for those portions of an application that play to its strengths, while a general-purpose language performs tasks not expressible within a DSL. Laszlo LZX is a natural extension of this HTML/JavaScript model; it is a declarative DSL whose domain is RIA applications, and it is also combined with JavaScript.

LZX is a declarative language based on XML, which accesses a supporting JavaScript library, the *OpenLaszlo Runtime Library (ORL),* which contains the Laszlo API. Laszlo also features a common derivative of JavaScript that is formally known as JavaScript 1.4, or ECMAScript 262 Edition 3. It contains only the JavaScript language fundamentals and a base set of objects; browser-specific objects are only accessible through the browser.

LZX is distinctive in how it uses its declarative features to leverage the ORL API services. These declarative features include

- Class-based inheritance
- Hierarchical parent-child addressing
- Parent-child attribute propagation
- Built-in event handling
- Animation
- Constraint notation support for event handling
- Data binding to XML elements

All these features, other than class-based inheritance, are defined by the fundamental `LzNode` LZX class. In this chapter, we'll look at the base functions of an `LzNode` object and how they define the major declarative features of LZX. Along the way, we'll also examine how the dynamic object-oriented features of JavaScript are used by objects in Laszlo.

Rather than just explaining abstract concepts, we'll demonstrate the declarative features by applying them to a complex real-world model to maintain position

[1] A programming language is called Turing complete if it has a computational power equivalent to a universal Turing machine. In short, this means that if it's theoretically possible to compute some result, then you can write a program in that language to do the computation. However, this does not imply a capability for easy programming or efficient execution.

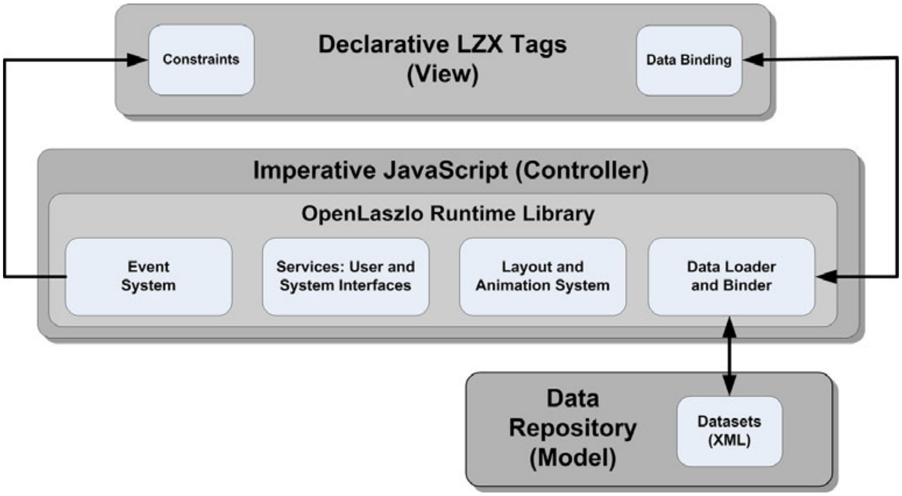
within a formation of aircraft as they fly across the screen. This application demonstrates how the complex problems of maintaining coordinated actions among a large number of objects are easily addressed in LZX. This example is applicable to a wide range of situations requiring related and unrelated objects to act in concert.

2.1.1 Laszlo's three-tiered structure

Like any modern user interface, a Laszlo application adheres to the principles of the *model-view-controller (MVC)* design pattern. In this architecture, the three functions of handling user input, modeling application state, and generating visual results are handled by three separate components: the *controller*, the *model*, and the *view*. The goal of MVC is to separate the functions of the model and the view—decoupling the model from implementation details concerning the view—to allow the controller to handle input actions and determine when the model or the view needs updating.

A Laszlo application has the three-tiered structure illustrated in figure 2.1, which is an instance of the MVC design pattern. The layers are

- A view layer of declarative LZX tags
- A controller layer of imperative JavaScript that accesses the ORL API
- A model layer of XML datasets



Constraints
Declarative LZX Tags (View)
Data Binding
Imperative JavaScript (Controller)
OpenLaszlo Runtime Library
Event System
Services: User and System Interfaces
Layout and Animation System
Data Loader and Binder
Data Repository (Model)
Datasets (XML)

Figure 2.1 The client-side architecture of Laszlo can be viewed as a three-tiered structure. At the top is a layer with the declarative LZX tags; this layer corresponds to the view component of the model-view-controller design pattern. The middle layer, corresponding to the controller component, contains imperative JavaScript and the ORL. The bottom layer, corresponding to the model component, contains the application state stored as XML data.

The view layer provides the graphical structure of an application. The controller layer handles events and other special services in support of the graphical structure. The model layer contains the data constituting the state of the application.

The ORL, accessed by the controller for interacting with the user and the environment, is organized into these areas:

- An *event* system for communication among objects in the view and controller layers
- The Laszlo *services*, including device interfaces, timers, and screen interfaces such as terminal focus and window tracking
- The *layout and animation* system for controlling the presentation of view-based objects
- The *data loader and binder* system for binding view objects to XML data nodes in the model layer

Access to the controller layer is provided through event handlers at either a global or an object level.

The model layer maintains the state of the application within a series of XML documents known as *datasets*. Since each dataset is an XML document, it consists of a root node from which a tree-structured hierarchy emanates. This tree structure consists of data and text nodes, also referred to as elements, which can be either parent or end nodes. Any display objects within the view layer can establish a data-binding relationship to a node through an *XPath expression*. This allows changes to the model's datasets to be immediately reflected in the appearance of bound visible objects.

XPATH XPath is a W3C standardized expression language for specifying data nodes in an XML document. An XPath expression has two components: a *location path* followed by either a *wildcard* or a *predicate operator*. The location path notation mimics the Uniform Resource Identifier (URI) and file path syntax to specify a path to a XML element. Once an element is located, the wildcard or predicate operator is applied to access the data element or one of its attributes.

XML serves two fundamentally different purposes in Laszlo. It is used as a high-level markup language to contain the LZX declarative language, and to encode structured data within a dataset. Since code and data are both XML, they can be integrated so everything is contained in a single XML document.

Although MVC is the underlying architecture for a Laszlo application, it needs additional support for the interactive requirements of RIAs. In the next section, we'll explore Laszlo's approach in using MVC to separate an RIA business model from its screen presentation.

2.1.2 Interfacing Laszlo to a web server

While a Laszlo application can use its MVC architecture to maintain state and work independently for extended periods, it can't permanently store these results. Laszlo can interface to a back-end web server to retrieve or store data, as XML documents, to a database to permanently store its state.

Different web services, such as SOAP, XML-RPC, and Representational State Transfer (REST), are available. Because of its popularity and simplicity, we'll only use REST in this book.

REST REST uses the term *resource* to describe any asset available from a web server. In our case, this is an XML document. Suppose that you want a list of Manning books; the URL to obtain this resource from a REST interface is simply http://www.manning.com/books. Entering this URL returns a resource consisting of an XML document contained in an HTTP response, hence the moniker "XML over HTTP."

A Laszlo client communicates to its supporting server through HTTP requests and responses. Although they perform different tasks, both Laszlo and its supporting web server use an underlying MVC architecture, shown in figure 2.2. The difference between them lies in the complexity of the controller. The server's model layer controls the permanent state of the application by storing the information

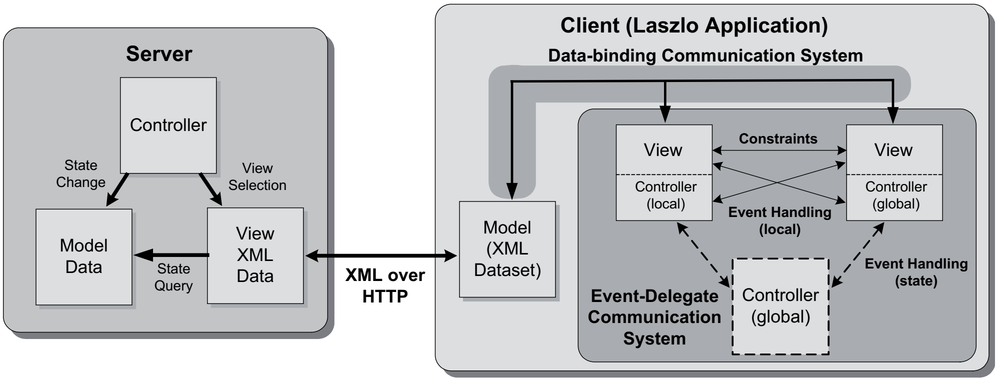


Figure 2.2 In contrast to the MVC architecture used by a back-end HTTP server, Laszlo supplements its basic MVC architecture with two publisher-subscriber communication systems. The event-delegate system supports general interactive RIA needs between the view and controller layers. The data-binding system sits between the model and view layers. Laszlo uses a multiple-controller approach, with controller functionality embedded in each object in the view layer. An optional central controller is indicated with dashed lines.

data contained within an HTTP request into a database, while its view layer returns an HTTP response containing an XML document.

Although a web server requires a relatively simple controller to support its interactive RIA features, Laszlo supplements its controller with two communication systems: event-delegate and data-binding. The event-delegate system allows objects within the view and controller layers to communicate, and the data-binding system allows objects in the view layer and data in the model layer to communicate. Laszlo also uses a multiple-controller approach, allowing local controllers within objects as well as global controllers. This enables Laszlo to maintain local state within objects and as well as global state within central XML datasets.

The ability to have the display—the view layer—immediately reflect changes in its model state—the XML dataset—is a distinguishing feature of RIAs. This dynamic updating provides a sense of continuity between actions and their consequences, as reflected in an updated display. This capability is supported in Laszlo by its two communication systems, both of which conform to the *publisher-subscriber* design pattern, which is also referred to as the observer pattern.

2.1.3 Publisher-subscriber communications

A publisher-subscriber pattern is a one-to-many communication system in which a *subject* publishes *notifications* and *observers* can *subscribe* to receive these notifications. One benefit of this loose coupling among objects is that publishers don't need to be aware of their observers. Laszlo implements this publisher-subscriber paradigm with *delegate* and *event* objects. An event object serving as a publisher contains a list of delegate objects, the subscribers. To be listed, a delegate *registers* itself with an event object. When the attributes of a publisher object change, an event is sent to every subscriber delegate.

Since declarative tag objects are automatically registered by Laszlo, there is no need to explicitly register them as publishers or subscribers. This creates an environment where objects can publish and subscribe freely, unencumbered with coding overhead. Since RIAs require a tremendous amount of communication among objects, this reduces repetitive coding, producing more manageable applications. This allows developers to concentrate on the larger algorithms, instead of being bogged down with implementation details.

Data-binding communication uses the same approach but establishes communication by binding a view object to an XML data node through an XPath expression. The presence or absence of the data node controls the view's visibility. In the case of more than one matching node, multiple copies of the view are generated. But after this relationship has been established, communication still relies on events and delegates.

When the OpenLaszlo Server compiles a Laszlo application, it converts the LZX declarative statements into JavaScript that is linked with the ORL. This close relationship allows ORL services such as event handling and data bindings to be directly embedded into declarative notation. It also allows declarative tags to have different semantics from JavaScript, while still taking advantage of JavaScript's dynamic features. The next section demonstrates how class-based semantics are supported within LZX declarative tags, and how these semantics combine with those of JavaScript.

2.1.4 Combining inheritance models

LZX declarative tags use a *class-based* inheritance model, a programming paradigm familiar to most developers. On the other hand, JavaScript uses a *prototype-based* inheritance model, which makes no distinction between classes and instances; there are only *objects*. In LZX, an object can be implemented either as a JavaScript object or as an instance of an LZX class. The pleasant result of this combination is that an LZX object has the flexibility to dynamically add, override, or delete attributes and methods at runtime. Since all the base LZX classes are written in JavaScript, all LZX objects have this flexibility.

This is quite a departure from the world of static class-based objects, which most developers are familiar with from Java. Therefore, let's take a moment to compare these two object-oriented systems to better understand how each system of inheritance works and where each is most appropriate to use.

JavaScript has no concept of a class; it has only objects. An object in JavaScript is represented by an associative array consisting of named slots that can either contain a *property* or a *method*. A *prototype object* is used as a template to initialize a new object, which can later be modified. Object creation and modification happen at runtime, since there is no discrete compilation phase.

In contrast, with static class-based languages, classes are created at compile time and instances are created at runtime. Once a class has been defined, its attributes and methods can't be changed at runtime. Figure 2.3 shows an example comparing class- and prototype-based inheritance. In class-based inheritance, a *derived class* extends its *base class* by adding new fields—one called `shape` in the example—and a corresponding set of *getter* and *setter* methods—called `getShape` and `setShape` in the example. Once this class definition has been updated, it is compiled to produce an object instance. From this point on, the characteristics of objects instantiated from the class definition are fixed. With prototype-based inheritance there is no compile time, and objects can be modified by adding or subtracting properties or methods, during execution, as shown in the lower part of figure 2.3.

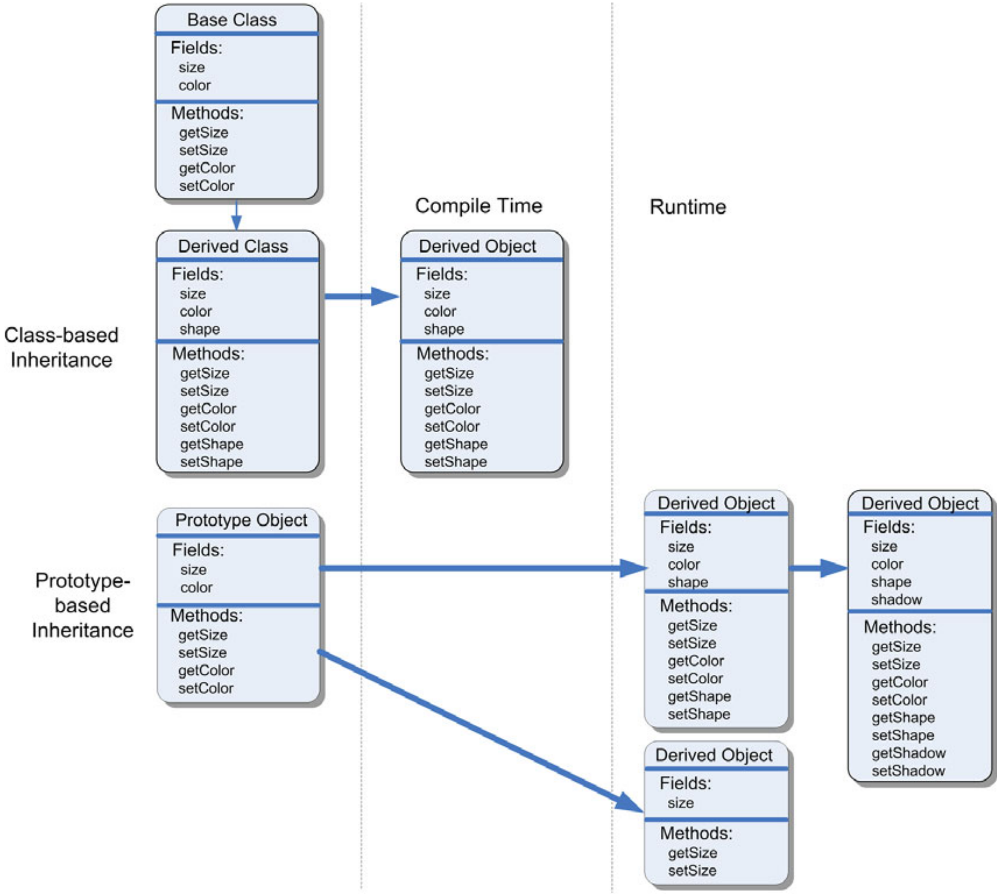


Figure 2.3 Class-based objects are static; they can be defined and extended only at compile time. Prototype-based objects are dynamic, allowing derived objects to be defined and created at runtime. Properties and methods can be added, modified, or deleted from these derived objects. The inability of class-based objects to dynamically create derived objects at runtime reduces programming flexibility.

For many problem domains, the inflexibility of class-based objects is an asset, not a deficiency. For example, static definitions are required for standard interfaces such as JDBC or HTTP. If these interface definitions were not fixed, an object could break interoperability by modifying its definition. In such applications, classes contain identical methods and fields for all instances, and object instances from the same class differ only in their data values.

But this is not the case for user interfaces. Rather than a large number of similar objects, each GUI object typically is unique, with no suitable class definition for all of them. Consequently, it's more practical to take an existing prototype object containing a significant portion of the required behavior for a new object and tweak it by modifying its properties and methods, until the new object has the desired appearance and behavior. This practice only works when properties and methods can be modified directly in the instance.

LZX combines the best of both worlds. It provides the familiar operation of class-based inheritance with declarative tags, while at the same time the dynamic capabilities of prototype-based inheritance are available. This approach allows declarative classes to be created to either directly produce instances or to serve as a convenient starting point for further tweaked unique objects. It also allows an instance to be easily converted into a class definition. Later, we'll see how this leads to a code development strategy known as *instance-first development.*

To understand the operation of LZX's declarative tags, let's start with an in-depth look at the two fundamental LZX superclasses, the `LzNode` and `LzView` classes. These classes define the underlying characteristics for general-purpose LZX declarative tags.

2.2 LZX classes

Before discussing the operations underlying the declarative LZX tags, we need to understand some of the general properties shared by most tags. We'll start by looking at the class hierarchy of LZX tags. Then we'll examine how user classes are created and the relationship of LZX classes to JavaScript classes.

2.2.1 The LzNode class

The `LzNode` class is the superclass for general declarative tags. The properties of an `LzNode` object include its attributes, methods, and the default events it is registered to handle. But not all declarative tags are derived from the `LzNode` class. Some special-purpose tags are intended to support the general-purpose tags, such as the following:

- Resources
- Fonts
- Includes, importing, and libraries
- Methods and handlers
- Attributes
- Script
- Class
- Unit tests

These tags have a non-LzNode lineage and are confined to the top level of a Laszlo application or within a library tag. For simplicity, we normally use the term *declarative tag* to refer only to general-purpose LzNode-derived tags.

Objects derived from the LzNode class, comprising approximately 85 percent of the Laszlo tags, include this wide selection of objects:

- All visible objects (canvas, views, and components)
- Datasets (discussed in chapter 10)
- Animators (discussed in chapter 5)
- States (discussed in chapter 5)
- Data pointers and data paths (discussed in chapter 11)
- Layouts (discussed in chapter 6)

Any JavaScript object derived from the LzNode class can also be represented as a declarative tag. This allows an object to be instantiated either directly by appearing as a declarative tag or through JavaScript's new constructor. Although the final result is the same, the difference is timing.

Since all declarative tags are compiled and executed as JavaScript statements, declarative tags are instantiated at compile time. JavaScript statements, on the other hand, can only dynamically instantiate objects at runtime. The succinct notation of declarative tags is used to create an application's initial structure, while new objects can be dynamically instantiated through JavaScript. This provides an ideal mix of ease of use and flexibility.

2.2.2 The LzView class

The second fundamental superclass, LzView, is a subclass of LzNode. These two classes are closely associated; LzNode defines common behavior for all LZX declarative tags, and LzView defines their visual characteristics. Together they describe the fundamental behavior of all visible declarative tags, as illustrated in figure 2.4.

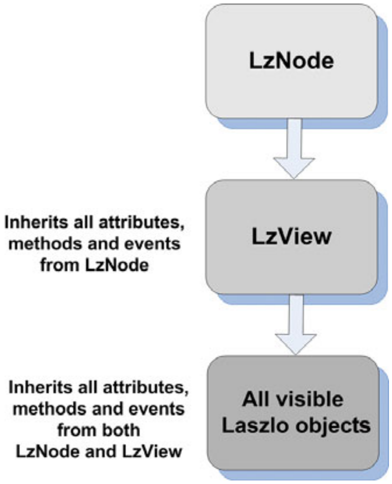

LzNode
LzView
Inherits all attributes, methods and events from LzNode
All visible Laszlo objects
Inherits all attributes, methods and events from both LzNode and LzView

Figure 2.4 The LzView class inherits its base attributes, methods, and events from the LzNode class. This base set of properties is augmented with LzView's base set of attributes, methods, and events to produce a visible object.

NOTE *Some terminology: Inheritance, derivation, extension, subclass, and superclass* The terms *is derived from*, *extends*, *is a subclass of*, and *has as its superclass* are equivalent. All imply class inheritance. For example, *B extends A* implies that B inherits all the properties of A.

Although the `canvas` tag is the root node for an entire application, its class is derived from the `LzView` class. If this is surprising, think of the situation this way: all visible elements must be an instance of a class derived from `LzView`. All Laszlo components—windows, menus, scrollbars, and so on—are of a class extending `LzView`. Consequently, the canvas, being a visual element—to see this, just set its background color with the `bgcolor` attribute—must also be a class extending `LzView`. An application can be considered as an interacting collection of views, contained within a parent view called the canvas.

2.2.3 Defining classes in LZX

Although tags are declarative, LZX embodies the semantics of a class-based object-oriented language. Once common functionality has been identified and a class has been defined, instances can be created and modified. A class can be extended from any `LzNode`-based class. Since we are working with dynamically interpreted declarative statements, whereby an element instance can be replaced by its definition without changing the application semantics, it is almost as easy to convert instances into class definitions as it is to create instances from classes. This is what leads to the code development strategy known as *instance-first development.*

Instance-first development encourages new functionality to be added to specific instances. If during development it turns out that this functionality has broader applicability, then it can be refactored back into the class definition for reuse. This approach also encourages class definitions to be postponed until there are enough potential instances to warrant a new class. This helps avoid many of the problems with prematurely creating abstract classes and then attempting to fit the problem to the solution. Since new functionality has been prototyped and tested before its inclusion into a class definition, this results in a flexible system that scales better to meet new requirements

A subclass inherits the properties and methods of its superclass and can selectively add or override them to provide specialized behavior. LZX only supports single inheritance, meaning that a class may have only one superclass. However, the inheritance chain can run arbitrarily deep; all properties and methods of a class are propagated down through each level, allowing many levels of specialization.

NOTE *More terminology: Naming JavaScript and LZX classes* The LzView class is written in JavaScript. In fact, any class name starting with Lz is written in JavaScript. A lowercase name implies that a class is defined in LZX. Consequently, strictly speaking, it is incorrect to refer to the *view class*, although we, and others, do use that term informally. Later, we'll see some examples of user-defined classes with lowercase names. This naming convention is just that, a convention. But it is widely used throughout the Laszlo API.

A user-defined class is specified with the class tag. Each new class is given a unique name and, optionally, the name of the superclass it extends, using the form

```
<class name="subclassname" extends="superclassname">
…
</class>
```

or

```
<class name="subclassname" extends="superclassname"/>
```

By default, a class extends the LzView class, so the extends attribute is optional. Actually, the meta-name superclassname in the previous code snippet is a bit of a misnomer because it can represent either a class or instance name. This allows for the support of instance-first development. As illustrated in figure 2.5, any node, view, or visible object can serve as the superclass for a class tag. Since LzView is the default class, it's extended by default. Once a class has been created, a new tag of that name is immediately available.

The following example shows how easy it is to create a new class. Here the default LzView class is extended to create a class called box with a size of 100 by 100 pixels:

```
<canvas>
   <class name="box" width="100" height="100" bgcolor="0xBBBBBB"/>
   <box/>
</canvas>
```

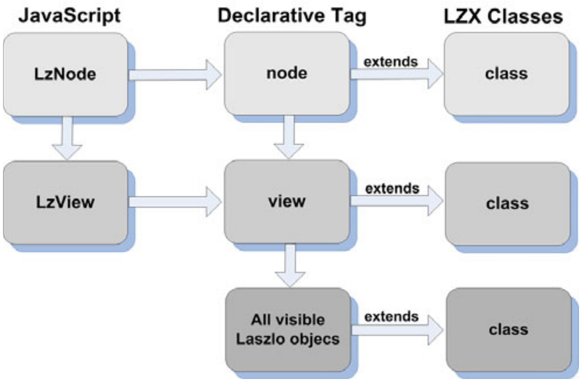


Figure 2.5
The relationship between JavaScript classes, declarative objects, and LZX-defined classes is shown here. Any node, view, or other visible object, whether instantiated from JavaScript or with an LZX tag, can serve as the superclass for a class tag. LzView extends by default if no other class is specified.

A box instance behaves like a view with width and height of 100 pixels and a background color of 0xBBBBBB. Thus, the `box` class is a specialized version of `LzView`. Any class defined in this way can be placed in a library file to allow other applications to use it. For example, if we want to create a red box, we can simply override the `bgcolor` attribute—popular color names are supported within declarative tags, in addition to RGB hexadecimal values—like this:

```
<box bgcolor="red"/>
```

If we want to create a red-labeled box, we can either augment the instance:

```
<box bgcolor="red">
   <text text="red box" valign="middle" align="centered"/>
</box>
```

or we could extend the `box` class to create a new class:

```
<class name="redLabeledBox" extends="box">
   <text text="red box" valign="middle" align="centered"/>
</class>
<redLabeledBox/>
```

This example demonstrates how natural it is to use instance-first development. A red-labeled box was first developed as an instance and then converted into the class `redLabeledBox`.

In the next section, we'll take a deeper look at the declarative features inherited from the `LzNode` class. Rather than just explaining these features in the abstract, we'll apply them to a visual model of a formation of aircraft.

2.3 The fundamentals of tags

In the previous sections we presented overviews of the Laszlo architecture, highlighting the three-tiered structure of declarative, JavaScript, and XML components within its MVC architecture, and LZX's class-based inheritance model. This section demonstrates how this supporting architecture and the class mechanism combine to produce the unified set of capabilities available with declarative tags. These powerful capabilities allow complex animated applications to be easily rendered. The demonstration is carried out with a small but complex visual model of aircraft in formation.

Life in LZX starts with `LzNode`. We have a small example to demonstrate all of its characteristics. Have you ever been to an air show at your local airport and seen a demonstration team, like the Blue Angels or the Snowbirds, flying in a tight formation separated by only a few feet? A typical formation is five aircraft in a "vee" pattern. How do they maintain the equal spacing of their formation? The answer is *relational offsets*.

Rather than using the horizon or instruments, a pilot flying in formation looks only at another aircraft in the group. Only the lead pilot flies freely, as each other aircraft maintains a constant relationship with the lead or some other aircraft. As each pilot maintains a position relative to one other aircraft, the entire formation maintains the correct pattern.

Our example displays a visual model of an aircraft formation. In the upcoming sections, we show how relational offsets in this model can be implemented in LZX, either statically or dynamically. Each of these features is illustrated in the model to show how the properties of an `LzNode` object support that feature. In particular, we'll show how constraint-based relationships are used to coordinate window layouts in a GUI. Just like the formation of aircraft, a window layout may have multiple visual objects that must coordinate their actions with a "lead window."

So before we're ready to fly, we'll need to start with the fundamentals of LZX, which begin with hierarchical addressing.

2.3.1 Hierarchical addressing

An `LzNode` object is instantiated declaratively with the `node` tag. Since `LzNode`-based declarative tags are stored in Laszlo's internal Document Object Model (DOM), they can maintain local data. This allows a tag to have a name and maintain an internal state with its attributes. All `LzNode`-based tags have two identification attributes, `id` and `name`. The `id` attribute is a *global* unique identifier, and `name` is a *local* identifier.

The object hierarchy in Laszlo's DOM corresponds to the parent-child placement in the XML document constituting a Laszlo application. The prefix `parent` on a `name` attribute refers to a local name one level higher within the object hierarchy. This prefix can be chained, as in `parent.parent`, to refer multiple levels upward. Any `LzNode`-based object can be identified by its absolute name, starting at the canvas, or a relative name, starting with a parent or other ancestor.

Global and local names

The `id` and `name` identifiers provide a link between declarative statements and JavaScript code. An attribute specified in any declarative tag can be referenced by JavaScript code. In the code

```
<canvas>
   <node name="top">
      <node>
         <handler name="oninit">
```

```
            this.setAttribute("name", parent.name + "_1");
         </handler>
      </node>
   </node>
</canvas>
```

we have a parent node named top with an unnamed child node containing an *event handler* named oninit. An event handler contains JavaScript code, is encapsulated within a *container* (in this case, the unnamed node), and is executed when an event occurs (in this case, the initialization of its container—that's what oninit means). For the moment, don't worry too much about event handlers—we'll come back to them in just a bit. It's sufficient to know that the JavaScript in this handler is executed during initialization.

However, the point of this little example is not event handlers but to show how the JavaScript in the child node references its parent's name attribute with parent .name. In fact, it uses its own setAttribute method—that's what this.setAttribute means—to set its name to "top_1."

A parent-child hierarchy conforms to the rules of an XML DOM; a parent node can have many children but a child node has only one parent. In LZX, a child's parent is fixed and can't be replaced with another parent. As we develop the aircraft formation example, we'll demonstrate how this parent-child hierarchy is used to simplify the coordinated movements of the aircraft.

Modeling an aircraft formation

Figure 2.6 shows a five-aircraft formation image that we want to display with our first version of the example. Listing 2.1 shows how we start with a simple structure in an XML document.

An XML document contains character *text data* interspersed with markup tags to separate the text data into a hierarchy of *elements, attributes* of those elements, and text data. Every XML document structure must have a single top-level root element called the *document root.* This root element anchors a tree structure of parent nodes containing child nodes.

Nodes in an XML document can form two kinds of hierarchical structure: nested in a parent-child hierarchy, or as siblings in a flat hierarchy. Of course, nodes could also just be located in disparate areas of the document. But since nodes

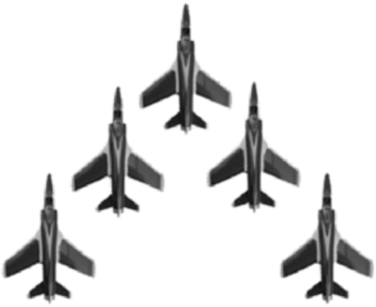


Figure 2.6 This is the five-aircraft formation image that we want to display with our first version of the example. Listing 2.1 shows how we start with a simple XML structure.

can reference each another through relative addressing, they can always establish a flat relationship, no matter where they might be located.

As a first attempt at representing our aircraft formation, we'll use the simple nested parent-child hierarchy shown in listing 2.1. The visual positions of the XML statements neatly correspond to the positions of our jets in formation.

Listing 2.1 A five-aircraft formation modeled in XML

```
<jet>     ◁——— Lead parent aircraft
   <jet>     ◁——— First left child
      <jet/>    ◁——— Child of first left child
   </jet>
   <jet>     ◁——— First right child
      <jet/>    ◁——— Child of first right child
   </jet>
</jet>
```

A matching pair of opening and closing markup tags indicates that `jet` is a parent data node to a child node that is also a `jet` tag. Two of the jets contain a closing `/>` indicating that they're both *end nodes*—nodes with no children. In an XML tree hierarchy, a parent can have many children but a child has only one parent. Currently, none of the `jet` tags contains attributes.

This parent-child hierarchy is a convenient framework to represent the order and relationships of the aircraft. The lead aircraft is the root node with two child nodes, a *first left child* and a *first right child*. The fourth aircraft is a child node of the first left child, while the fifth aircraft is the child node of the first right child. They can also be considered as grandchildren of the root node. As a result, the order and arrangement of the aircraft is clearly indicated by the positional layout of the XML statements. It's an easy step to convert this into a Laszlo application.

Converting our aircraft formation XML statements into the Laszlo application shown in listing 2.2 only requires that beginning and ending `<canvas>` … `</canvas>` tags be added and that an initial class definition be created for a `jet` class. To complete the display, we'll use a graphic image to provide a visual representation of an aircraft. The image is defined as a *resource* called `jet`, which can be found in the source file F18_Hornet.png, available in the accompanying Manning source code archive.

Listing 2.2 The five-aircraft formation as a Laszlo application

```
<canvas>
   <resource name="jet" src="F18_Hornet.png"/>
   <class name="jet" resource="jet"/>
   <jet>
      <jet>
         <jet/>
      </jet>
      <jet>
         <jet/>
      </jet>
   </jet>
</canvas>
```

Specify image file

Define class with visual representation

We now have five jet objects available for use. When we run the application, the display in figure 2.7 appears on the screen. So why do we see only a single image?

Actually, all five jets are displayed, but all at coordinates (0, 0). This is what happens when five jets are stacked on top of each other. The problem is that, although we have captured the general layout, we still need a way to specify the relative distances between certain aircraft. Fortunately, LZX's parent-child hierarchy makes this easy by propagating certain attribute values to the children.

Figure 2.7 The initial result of executing our application shows all five jet object images overlaying one another, so only the top image can be seen.

2.3.2 *Parent-child attribute propagation*

In a parent-child hierarchy, the visual attributes shown in table 2.1 are propagated down from each parent node to its child nodes. This allows a parent node to establish a base value for an attribute, which is automatically propagated to its child nodes, whose values can be set as offsets from this base. We'll discuss each of these attributes in greater depth as we move on in the book.

Table 2.1 Visual attributes that are passed to children

Attribute	Description
`x`	Specifies the x position of a view
`y`	Specifies the y position of a view
`rotation`	Specifies the rotation value in degrees

Table 2.1 Visual attributes that are passed to children *(continued)*

Attribute	Description
`opacity`	Specifies the opacity value ranging from 0 to 1.0 (transparent to opaque)
`stretches`	Causes dimension changes so child nodes fit within the view

Since all internal spatial relationships are based on relative offsets, when these attribute values are updated for a parent view, the result is that its child nodes act in a coordinated manner. This is just the effect needed to allow the aircraft to maintain the conformity required by a jet formation.

Coordinating the aircraft formation

Let's move our formation so that it's initially positioned in the lower portion of the screen at (400, 400) and separate the jets by 50 and 40 pixels in the x and y directions respectively, to create an inverted "vee" shape. The layout and distances are shown in figure 2.8.

Within a parent-child hierarchy, every child node inherits the `x` and `y` attributes of its parent, so rather than specifying absolute distances, the location of each child node is specified as an offset from its parent. The lead aircraft serves as the parent node, so the offset for its left child is (–50, 40). The bottom aircraft in the left wing, inheriting the updated `x` and `y` attributes of its parent, need only apply the same offset (–50, 40). The values are cumulative and so result in a total offset of (–100, 80). An analogous set of offset values (50, 40) is applied to the

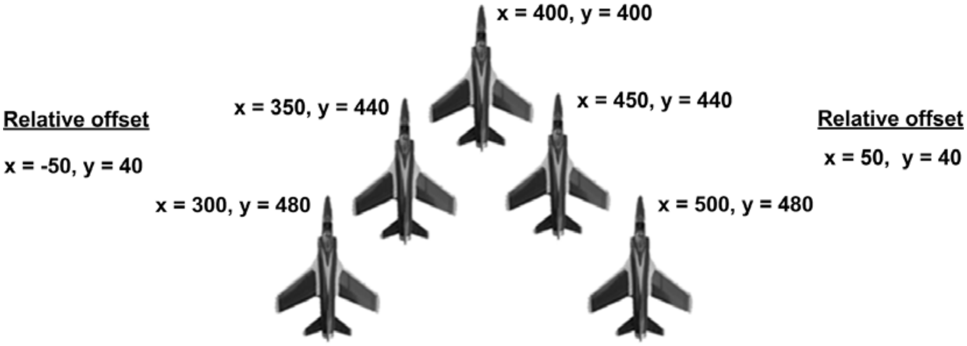


Figure 2.8 To maintain the inverted "vee" shape of the formation, each jet is offset by 50 and 40 pixels in the x and y directions, respectively. The absolute distances appear next to the jets, while their relative offsets appear at the right.

right wing to produce a cumulative total of (100, 80). The LZX statements for the aircraft, updated for these offsets, are shown in listing 2.3.

Listing 2.3 The five-aircraft formation with relative spacing between aircraft

```
 <jet x="400" y="400">
   <jet x="-50" y="40">
      <jet x="-50" y="40"/>
   </jet>
   <jet x="50" y="40">
      <jet x="50" y="40"/>
   </jet>
</jet>
```

A parent-child hierarchy provides the convenience of maintaining base and offset values; it's simple to create and requires modest system resources since the jet objects have no communication overhead. And there's no need to name any of the nodes, because they can be referenced with relative notation. The parent node provides a single location to effect a visual change throughout an entire hierarchy. After updating and executing our model, we see a jet formation identical to the image in figure 2.6.

However, there are limitations to working with a parent-child hierarchy. Since a child node can't change its parent, this relationship must be created at compile time; it can't be dynamically created during runtime. Additionally, the set of propagated attributes is limited to those listed in table 2.1. For example, a parent's `width` and `height` attributes aren't available to be offset by its children. We can cheat, though, by setting the `stretches` attribute to be `both`, `width`, or `height`, which gives the same result.

An alternative to a parent-child hierarchy is to use a sibling, or flat tag, structure. This results in slightly more verbose code, but while parent-child relationships must be defined at compile time, a flat hierarchy can be created at runtime and any node can participate.

2.3.3 Flat tag hierarchies

Since a flat hierarchy doesn't have the automatically propagated values of a parent-child hierarchy, within our formation we explicitly build a hierarchy by establishing communication between a leader and its followers. One jet is designated as the leader to establish base attribute values and to communicate value changes by sending events to its followers. For easy reference, the leader needs a name:

```
<jet name="LEAD" x="400" y="400"/>
```

Since "LEAD" is a local name, it is referenced with a relative address of parent.LEAD. To maintain the formation, each following jet maintains a single relationship with the lead jet—yes, we know that this isn't how formation pilots would do this, but let's not stretch our analogy too much. To model this behavior, each following jet updates its properties to conform to those of its leader. In our case, for example, the left following jet could update its local value with the sum of its offset and this new value:

```
this.x = parent.LEAD.x - 50
this.y = parent.LEAD.y + 40
```

This establishes an offset of –50 for its x attribute and 40 for its y attribute. Although this example is quite small, a real application could potentially have a large number of property changes that need to be communicated.

At this point, we've introduced some basic ideas on how to model the aircraft formation. Our purpose has been to illustrate how LZX objects in a hierarchy are named, how they store local data, how data values can be propagated down in the hierarchy, and how they can be organized in an alternative flat hierarchy. In the last case, a flat hierarchy, we've introduced the requirement for event communication among the jets.

The following sections continue with the formation example to illustrate how LZX's built-in event handling and constraint notation perform the communication job for us, as well as how XML data binding allows data to control the display, and finally how animation makes the model formation scoot dramatically across the screen.

2.3.4 Built-in event handling

In our formation example, the leader event object shown in figure 2.9 is the lead jet. The follower delegates are the remaining jets in the formation.

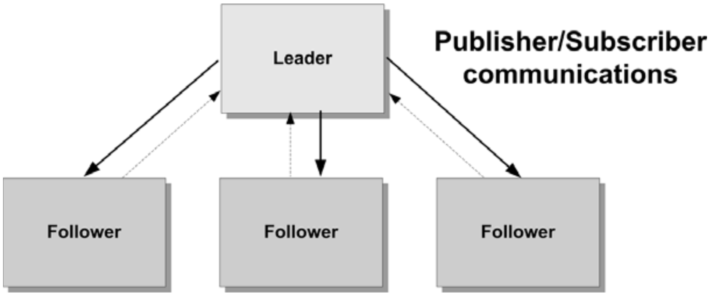


Figure 2.9
The dotted lines represent the initial registration of the followers with the leader. Event communication, indicated by the solid arrows, is one-way from the leader to its followers. In Laszlo, the leader is called an event object, and each follower is called a delegate.

Writing all the event-handling code to support event communication among objects can be laborious. Each delegate must be registered to receive an event; this requires the sending object and the event be specified along with an event-handling method. In addition, the sender must explicitly send each event. Requiring a developer to write all this code adds a significant amount of work to support communications. It would reduce a developer's workload if this event-handling overhead could be performed "behind the scenes," thus allowing the developer to concentrate on writing the event handler logic.

Since events require processing, declarative notation can't be used for event handling; declarative statements only specify things and don't actually perform any work. As in HTML, all event handlers are written in JavaScript. Since event handling occurs in the context of an object, each event handler is contained within an enclosing declarative statement.

To understand the requirements for such behind-the-scenes event handling, let's think about a jet setting its x property. Occasionally, the lead jet needs to move freely without communicating property changes to other jets; the lead is "offline," so to speak, since other jets receive no communications. In such an offline mode, a lead jet can set its x attribute with a simple LZX assignment, which generates no event:

```
this.x = 400;
```

At other times, the lead jet is in formation and so must communicate property changes to other jets; the lead is "online," since other jets do receive communications. In such an online mode, a lead jet can set its x attribute with the setAttribute method, which does generate an event:

```
this.setAttribute("x", 400);
```

This generates an onx event—all events have an on prefix—which is sent to all registered listeners and contains the updated property value.

Normally, a listener must register for any events it wants to receive. However, all LzNode-derived tags automatically have default event handlers registered for all its system and user attributes. A default event handler simply sets its attribute to the event argument value like this:

```
this.x = x;
```

However, any default event handler can be overridden. In this case, an overridden event handler would look like this:

```
<handler name="onx" args="x">
   this.x = x - 50;
</handler>
```

A tag object often uses an event handler for events it generates itself. However, it also needs to handle events from other objects. In that case, the event handler must reference the sending object. When the sender is itself, this reference can be omitted. In our case, handler `onx` responds to a property change event from the `LEAD` jet; since the event is being sent by `parent.LEAD`, it must be specified, as we see in listing 2.4.

Listing 2.4 The formation with event handlers

```
<jet … >
   <handler name="onx" args="x" reference="parent.LEAD">
      this.x = x - 50;
   </handler>
   …
   <handler name="onopacity" args="op" reference="parent.LEAD">
      this.opacity = op;
   </handler>
</jet>
```

Handles onx event from specified jet

Handles onopacity event from specified jet

To perform nondefault processing, overriding handlers must be provided for each attribute involved. As you can imagine, this can quickly grow cumbersome. What's really needed is a concise notation that is a mixture of tag notation and JavaScript.

2.3.5 Event handling with constraints

The LZX construct that blends tag notation with JavaScript is the *constraint*, where a property is assigned the value from a JavaScript expression contained within a character string. To protect the expression from being interpreted as character text, it's wrapped in curly braces with a leading dollar sign:

```
x="${parent.LEAD.x-50}"
```

A single data element can contain any number of constraints, as many as one per attribute.

In our formation example, all the constraints needed to maintain a consistent relationship between a leader and its followers can be expressed in a single declarative tag, as shown in listing 2.5.

Listing 2.5 The formation application with constraints

```
<canvas>
  <resource name="jet" src="F18_Hornet.png"/>
  <class name="jet" resource="jet"/>
  <jet name="LEAD" x="400" y="300"/>
```

```
    <jet x="${parent.LEAD.x-50}" y="${parent.LEAD.y+40}"/>
    <jet x="${parent.LEAD.x-100}" y="${parent.LEAD.y+80}"/>
    <jet x="${parent.LEAD.x+50}" y="${parent.LEAD.y+40}"/>
    <jet x="${parent.LEAD.x+100}" y="${parent.LEAD.y+80}"/>
</canvas>
```

Constraints position following jets

Now that we have identical jet formations created for hierarchical or flat structures, we're ready to get these formations flying across the screen. For either case, a steady stream of `ony` events is required to continually update the y coordinates to make the jets move.

2.3.6 Animating declarative tags

Every `LzNode`-derived object contains a built-in *animator* to incrementally vary its numerically based attributes. This animator can only be called through JavaScript, as it's not directly accessible to a constraint, although animators can be declared as tags.

An animator requires a starting and ending value, as well as a duration. Together, they determine the velocity of the animation. An attribute doesn't need visible properties to be animated. For example, it could be used as a timed `for` loop to iterate through a range of attribute values. Since every object possesses the ability to be animated and to communicate with other objects, this provides a fine-grained sense of animation with a level of interactivity bounded only by your imagination.

An animator generates a sequence of events, each containing an incremented property value. This stream of incrementing events is directed at either the parent or the leader node. In our case, since each jet is a visible object, it is registered to handle `onx` and `ony` events and has a default event handler to update these properties. When the animator begins emitting its stream of incrementing events, the `x` and `y` attributes of the parent node are continuously updated, resulting in a simulated movement. Since both the parent-child and the flat hierarchies use the same base-offset mechanism, updating the parent or leader node's attributes automatically causes the rest of the jet formation to move along with it.

Animation occurs asynchronously, so a second animator begins execution before a first animator finishes. Starting two animators sequentially for the `x` and `y` attributes actually executes them concurrently, which causes our formation to fly along a diagonal path.

But why stop there? Additional animators could easily be added to create an even more realistic simulation of flight. Adding rotation, opacity, height, and width animators would produce the illusion of a banked turn, with the aircraft images getting smaller and fainter as they recede into the distance. Any of the attributes listed in table 2.1 could be used to offset children.

We'll keep things relatively simple and fly the formation directly north as it fades into the distance. The parent or leader node is given an `oninit` initialization event handler to contain an animator for its `y` attribute from its initial position of 400 to a final position of 50 over two seconds (time is given in microseconds). Analogous animators are included to change the `opacity`, `width`, and `height` attributes. Although `width` and `height` are not listed in table 2.1, adding a `stretches` attribute to the parent node allows the resource image to be compressed.

So let's start with the nested parent-child jet formation shown in listing 2.6. Adding all these animation calls requires only that an initial event handler be attached to the lead jet, with its resource image *stretchable* in both directions, width and height.

Listing 2.6 Animating the formation model

```
<canvas>
   <resource name="jet" src="F18_Hornet.png"/>
   <class name="jet" resource="jet" stretches="both"/>
   <jet x="400" y="400">
      <handler name="oninit">
         this.animate("y", 50, 2000);
         this.animate("opacity", .3, 2000);
         this.animate("width", 90, 2000);
         this.animate("height", 90, 2000);
      </handler>
      <jet x="-50" y="40">
         <jet x="-50" y="40"/>
      </jet>
      <jet x="50" y="40">
         <jet x="50" y="40"/>
      </jet>
   </jet>
</canvas>
```

Animates lead jet

A flat hierarchy requires constraint-based communication for each of these attributes, so it's naturally a bit longer, as seen in listing 2.7.

Listing 2.7 The formation model using a flat hierarchy

```
  <canvas>
    <resource name="jet" src="./F18_Hornet.png"/>
    <class name="jet" resource="jet" stretches="both"/>
    <jet name="LEAD" x="400" y="400">
       <handler name="oninit">
          this.animate("y", 50, 2000);
          this.animate("opacity", .3, 2000);
          this.animate("width", 70, 2000);
          this.animate("height", 70, 2000);
       </handler>
    </jet>
    <jet x="${parent.LEAD.x-50}" y="${parent.LEAD.y+40}"
           width="${parent.LEAD.width}"
           height="${parent.LEAD.height}"
           opacity="${parent.LEAD.opacity}"/>
    <jet x="${parent.LEAD.x-100}" y="${parent.LEAD.y+80}"
           width="${parent.LEAD.width}"
           height="${parent.LEAD.height}"
           opacity="${parent.LEAD.opacity}"/>
    <jet x="${parent.LEAD.x+50}" y="${parent.LEAD.y+40}"
           width="${parent.LEAD.width}"
           height="${parent.LEAD.height}"
           opacity="${parent.LEAD.opacity}"/>
    <jet x="${parent.LEAD.x+100}" y="${parent.LEAD.y+80}"
           width="${parent.LEAD.width}"
           height="${parent.LEAD.height}"
           opacity="${parent.LEAD.opacity}"/>
 </canvas>
```

After updating and executing our application, the jet formation will take off from its origin and terminate its flight at the destination, as shown in figure 2.10.

This example illustrates the range of addressing capabilities of LZX's declarative tags. They are flexible enough to be used in situations ranging from a static parent-child configuration to a dynamic configuration requiring constraint notation to handle event-based communication. Best of all, these capabilities can be freely mixed; constraints can be added to declarative tags within a parent-child hierarchy to obtain the convenience and flexibility of both worlds.

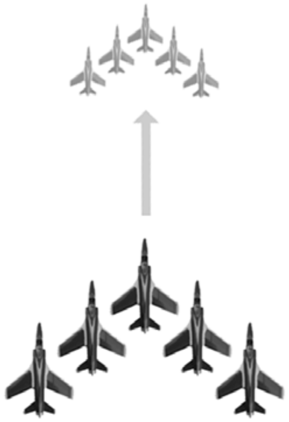


Figure 2.10 An animated sequence is created in which distance, size, and opacity are used to provide the impression that the aircraft formation has zoomed off into the distance.

Although our formation has flown off into the distance, we're not finished yet. Let's consider one of life's unpleasant realities—pilots get sick and can't fly. Since we can't just substitute any pilot, we are left with a missing aircraft. Our declarative notation needs a way to accommodate an outside event that removes an aircraft from a formation.

2.3.7 Binding tags to XML data

Although constraints are a powerful tool for expressing relationships among tags, these relationships are confined to the view and controller layers of MVC. Think of them as being horizontal relationships. An analogous tool is needed to express vertical relationships between the view and model layers. This allows changes to the model to be immediately reflected in a visual object, whose affected properties are then immediately propagated to other visual objects sharing a relationship with it. This cross-sectional relationship allows a single change to a data node within the model to initiate a chain reaction of events in the visual display.

Laszlo's data-binding communication system binds the view (the subscriber) to the model (the publisher). Since structured XML data describes "what" is to be seen—in this case, the number of jets—this binding is a good candidate for inclusion into LZX's declarative tags. An analogy may help to understand where we're heading here. We've seen how changes in an attribute value can generate an event that is handled by a registered receiving object. With structured XML data, changes in a data element value can generate an event that is handled by a registered receiving object.

Alternatively, think of extending the mechanism of a constraint that sends events from a leader to a follower object, to an analogous mechanism that sends events from an XML data repository to a dependent object.

Attribute-based constraints are implemented by embedding JavaScript into a tag. For the data-binding mechanism, we'll need a link between an XML data element and a tag. The missing piece we need for this link is the XPath expression.

Previously, we used an XML document to hold the application modeling the aircraft formation. Now we'll use an XML document to contain structured data. Although both are XML documents, they serve very different purposes. The document shown in listing 2.8, called jets.xml, contains a dataset listing the jets and their pilots.

Listing 2.8 An XML document jets.xml holding structured data

```
<dataset name="jets">
  <jet><pilot>Josh Sailor</pilot></jet>
  <jet/>
```

Contains missing text

```
   <jet><pilot>John MacNeille</pilot></jet>
   <jet><pilot>David Carpenter</pilot></jet>
   <jet><pilot>Peter Lindsay</pilot></jet>
</dataset>
```

The existence of a character string, a pilot's name, in a pilot text node of a jet data node governs the appearance of the corresponding jet. If there's no pilot, then obviously the jet can't be flown. If any change occurs within a text node—suddenly a pilot's name is entered—this generates an event causing the aircraft to appear within its formation. Of course, the converse also applies. If a change removes a pilot's name from a pilot text node, the corresponding jet disappears.

An XPath expression links a data element with a corresponding jet object. If a jet's associated XPath expression returns a matching pilot name from this data file, the aircraft flies and is displayed. The expression's location path specifies a path to a jet data element, while its predicate matches the pilot text node of the element.

Since the jet data elements are identical, each successive entry is specified with one-based array indexing. Here's what the path expressions look like:

```
jets:/jet[1]/pilot/text()
   …
jets:/jet[5]/pilot/text()
```

The initial jets token identifies the jets dataset. An array reference jet[i] selects the ith element in the file. And the predicate pilot/text() specifies the text data in the pilot node.

So how do we put an XPath expression into a tag? The analogy to the LZX constraint is the datapath attribute. Adding a datapath attribute to a declarative tag provides the mechanism to control the display of the tag, in our case a jet object and, of course, all its child jets. The XPath expression is given as the value of the data path. So let's compare how the different hierarchies react to the datapath attribute. Listing 2.9 shows the parent-child hierarchy with a data path for each jet object.

Listing 2.9 The parent-child hierarchy with data path binding

```
<jet x="400" y="400"
    datapath="jets:/jet[1]/pilot/text()">
  <jet x="-50" y="40"
       datapath="jets:/jet[2]/pilot/text()">
     <jet x="-50" y="40"
          datapath="jets:/jet[3]/pilot/text()"/>
  </jet>
  <jet x="50" y=40"
       datapath="jets:/jet[4]/pilot/text()">
```

Binds jet to missing text element

```
        <jet x="50" y="40"
             datapath="jets:/jet[5]/pilot/text()"/>
      </jet>
    </jet>
```

Next, the flat hierarchy is updated with data path attributes, as shown in listing 2.10.

Listing 2.10 The flat hierarchy with data path binding

```
 <jet name="LEAD" x="400" y="400"
      datapath="jets:/jet[1]/pilot/text()"/>
<jet x="${parent.LEAD.x-50}" y="${parent.LEAD.y+40}"
      datapath="jets:/jet[2]/pilot/text()"/>
<jet x="${parent.LEAD.x-100}" y="${parent.LEAD.y+80}"
      datapath="jets:/jet[3]/pilot/text()"/>
<jet x="${parent.LEAD.x+50}" y="${parent.LEAD.y+40}"
      datapath="jets:/jet[4]/pilot/text()"/>
<jet x="${parent.LEAD.x+100}" y="${parent.LEAD.y+80}"
      datapath="jets:/jet[5]/pilot/text()"/>
```

Binds jet to missing text element

Now each data path controls the display of its jet. The end result for either approach is that the display can be controlled by a single value in an XML document. But the displayed results are very different, depending on whether the node is represented in a parent-child or in a flat hierarchy.

The nature of the hierarchy determines the displayed results from a bound data path. When the bound node is a parent node, the parent node's state determines whether its child nodes are displayed. In other words, the ramifications of a missing plane depend on its position or on its dependencies. Since trailing aircraft depend on their lead (the parent node) to establish a base value, losing the lead grounds the entire formation. Similarly, missing lower-level nodes ground their trailing aircraft. Of course, losing a tail aircraft affects only that aircraft.

The XML pilot text node shows the pilots assigned to each jet. As you can see in listing 2.8, no pilot is assigned to the second jet. As a result, the left wing is grounded, leaving only the three aircraft shown in figure 2.11.

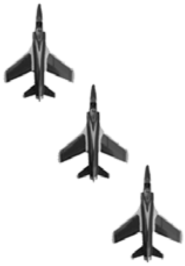


Figure 2.11 In a parent-child hierarchy, when a data path doesn't return any matching pilots, that aircraft and all its child aircraft are also grounded. Here, the left child of the lead aircraft is missing, grounding its trailing aircraft as well.

Since a flat hierarchy is fundamentally a relationship of equals, there is no cascading effect. If a node's data path doesn't return a matching pilot, then only that node is affected. This even pertains to the lead jet. This occurs because this lead jet still exists; it's just not visible, as we see in figure 2.12. So it can still send out events signaling changes to its other attributes: `y`, `opacity`, `width`, and `height`. This allows the jet formation to still fly across the screen, leaderless. Well, it seems that our formation analogy breaks down a bit here since those two lead jets have nothing to follow, but hopefully the main point has still been delivered.

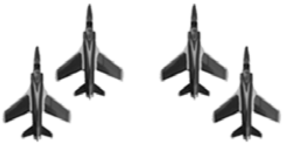


Figure 2.12 A flat hierarchy has no inherent dependencies. So when a data path doesn't return a matching pilot, it only results in the loss of that aircraft.

The purpose of this exercise was to provide a quick overview of Laszlo's declarative tags and to demonstrate how they model complex scenarios. We've covered a lot of ground and walked a fine line between introducing powerful features without getting into too much detail. If we have erred in either direction, we ask for your patience and for you to read on for more explanation. After some more reading, you may find it useful to come back to this chapter for a review that may be more illuminating at that time.

2.4 Summary

The Laszlo system embodies a three-tier structure, with a declarative upper layer, a procedural JavaScript middle layer, and a data repository lower layer. Since the declarative portion isn't Turing complete, it is combined with a general-purpose procedural language. This supports a development strategy whereby the declarative portion is used in areas that play to its strengths, while JavaScript is used for those portions that can't be expressed with declarative statements.

Laszlo LZX was designed to get the most from declarative specification. Since a declarative program describes *what* is to occur rather than *how* it is to occur, a significant increase in expressiveness results in large productivity gains. To achieve these gains, capabilities were added in these areas:

- Animation
- Built-in event handling
- Class-based inheritance
- Constraint notation support for event handling

- Data binding to XML elements
- Hierarchical parent-child addressing
- Parent-child attribute propagation

Each of these capabilities is supported by specific attributes in the `LzNode` class, which serves as the superclass for all declarative objects represented in the Laszlo DOM.

To make all this work, LZX declarative tags send and receive events generated by changes to attribute values. Since these communications are built into the declarative tags, they occur invisibly in the background. This allows a clear but minimal notation, which readily scales for complex cases.

A constraint-based relationship can be established between an LZX graphical element and an XML data element in a local cache. The LZX graphical element is bound to the XML data element through an XPath expression. This allows the visual presentation of the XML data to be controlled by the XML data itself. The absence or presence of XML data determines whether a graphical element is displayed. Furthermore, the bound graphical element can be a parent node to a large number of graphical child nodes, allowing a single XML data element to control a graphical display of any complexity.

The inheritance model for the `LzNode` class is based on the prototype-based capabilities of JavaScript. This provides a dynamic inheritance model, whereby object instances can be directly modified to add, delete, or update attributes and methods. New specialized objects can be built by tweaking the characteristics of existing objects, leading to the instance-first development strategy. The essence of this development strategy is to embellish instances with new functionality and, if these features prove to have wider applicability, to roll that back into the class definition. This provides a road-tested approach to class development and can prevent unnecessary abstractions.

In the next chapter, we continue to explore the relationship between declarative LZX and JavaScript.

3

Core LZX language rules

This chapter covers

- Debugging Laszlo LZX programs
- Commenting your code
- Declaring attributes
- Interfacing LZX tags to JavaScript
- Creating event handlers and constraints

A picture is worth a thousand words. An interface is worth a thousand pictures.

—Ben Shneiderman, University of Maryland

Becoming proficient in any language requires a mastery of its mechanics. This involves learning the "nuts and bolts" issues, such as how to declare comments and variables, call procedures, and functions, and other activities involved with an object-oriented environment. Since LZX uses XML and JavaScript, it leverages widely used programming technologies and is familiar to many developers. The remaining issue is how Laszlo integrates these differing technologies to produce LZX. In this chapter, we'll assume that you're already familiar with the basics of XML and JavaScript, and instead focus on identifying and understanding the operation of these declarative/JavaScript interface points. An interface point is the area where program control passes from the declarative XML to the imperative JavaScript domain. Mastering these concepts is the key to understanding the operation of a Laszlo application.

As you've seen, the strength of declarative XML notation is in describing the initial static layout of an application's interface elements with JavaScript used for handling its dynamic elements. Clearly, there has to be well-defined interface points, indicated by the white balloons in figure 3.1, to transfer processing from one domain to another.

What makes LZX challenging to learn is the fact that the Laszlo compiler manages and obscures many of these interactions. Its goal is to produce a more succinct programming interface to allow the development of larger and more sophisticated applications. But this results in a complex system whose operation isn't always immediately obvious. Hopefully by the end of this chapter, you'll have

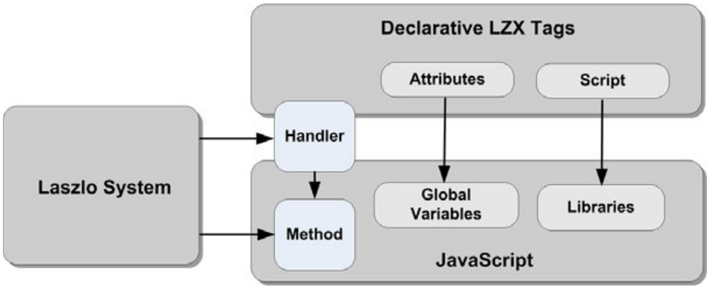


Figure 3.1 The lighter-colored areas indicate the interface points between the processing of LZX tags and JavaScript. In this chapter, we'll cover each of these topics and explain the role of the Laszlo compiler in managing them.

gained an intuitive understanding of the basic Laszlo operations and an appreciation for the abstractions performed by the Laszlo compiler. After mastering these concepts, you should find that your initial investment to learn LZX is substantially repaid with higher productivity.

However, before diving into interface issues, we'll first cover the basics of building, commenting, and debugging an application. This will provide the diagnostic tools needed for probing the interfaces. Rather than merely reading about how things work, you'll be able to run experiments to test a particular feature. Frequently, watching a feature in action is the easiest way to understand its operation.

3.1 Learning LZX Basics

Every Laszlo application is contained in an XML document that starts with a `canvas` tag. This satisfies the XML requirement that all documents contain a single root tag to contain all its other elements. The acceptable forms for displaying the `canvas` tag, and any other declarative tag, are shown in table 3.1.

Table 3.1 All Laszlo applications start with a `canvas` tag, the root tag for every application. The three acceptable forms for displaying the `canvas` tag, or any other tag, are shown here.

Tag Construct	Typical Use
`<canvas> … </canvas>`	A form acceptable for small, one-line applications

`<canvas>`
…
`</canvas>`
	The general case for building an application

`<canvas/>`	A legal statement expressing an empty application

A canvas serves as a container for *child* declarative tags, which appear between its opening and closing tags. These child declarative tags can contain JavaScript code. All Laszlo applications are structured as a parent-child hierarchical tree with the canvas as the root.

Of course, every application requires debugging and testing. One of the first questions of every programmer facing a new language is "How do I debug this?"

3.1.1 Debugging

Debugging information is displayed by adding the `debug` attribute to the `canvas` tag:

```
<canvas debug="true"/>
```

or adding a debug=true suffix to the URL query string like this:

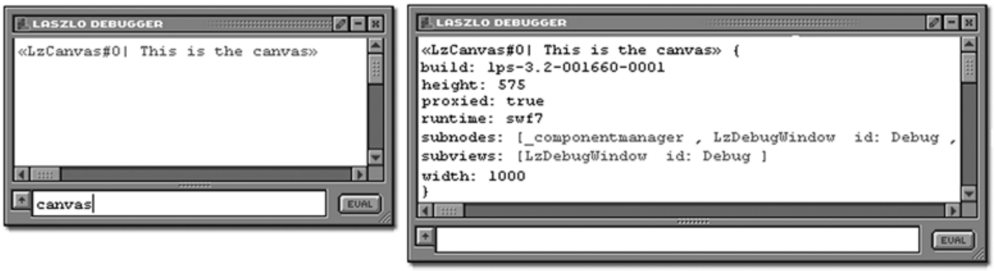
```
http://localhost:8080/lps/book/main.lzx?debug=true
```

This produces the debug window, shown in figure 3.2, which is both movable and resizable within the browser. The debugger should always be used during development, since it results in additional instrumentation being compiled to perform runtime error checking. You enter commands into the debug window through its text input field along the bottom, and then press the Enter key. The left screen in figure 3.2 shows the display of an LzCanvas object in compacted format. All debug output displayed in blue can be clicked to produce an expanded mode, where individual attributes are displayed.

The debugger is examining a running application, where tag names have been compiled into their corresponding JavaScript ORL objects. So the debugger output refers to the JavaScript LzCanvas object rather than the canvas tag.

To add a debug value within the code, it must be contained. An oninit event is generated when the canvas tag completes its initialization, indicating that the application is ready to receive input. An oninit event handler provides a convenient place to put this debug statement. The Debug object's write method is used to display its value:

```
<canvas debug="true">
   <handler name="oninit">
      Debug.write(canvas);
   </handler>
</canvas>
```



```
«LzCanvas#0| This is the canvas»
```

```
canvas
```

```
«LzCanvas#0| This is the canvas» {
build: lps-3.2-001660-0001
height: 575
proxied: true
runtime: swf7
subnodes: [_componentmanager , LzDebugWindow  id: Debug ,
subviews: [LzDebugWindow  id: Debug ]
width: 1000
}
```

(Compacted) **(Expanded)**

Figure 3.2 Adding the debug attribute to the canvas tag produces the debug window displayed within a browser. Entering an object name into the input field displays the object in compacted format. An object in compacted format is displayed in blue and can be expanded by clicking the mouse on it. The expanded display, shown in abridged form, displays its individual attributes.

This produces the compacted output seen in figure 3.2. Individual attributes can also be specified for display like this:

```
Debug.write(canvas.width);
```

To get the expanded output displaying all the attributes, use the `inspect` method:

```
<canvas debug="true">
   <handler name="oninit">
      Debug.inspect(canvas);
   </handler>
</canvas>
```

Laszlo is similar to other web technologies in that its internal DOM controls an application's appearance and operation. Although the debug window says Laszlo Debugger, it really is a DOM *property inspector* rather than a debugger, since there is no way to step through JavaScript code. In the next section, we'll explain how to perform a sanity check by verifying some of these property values.

Verifying debug values

Since JavaScript objects from the ORL correspond directly to LZX tags, individual fields of an object map to the attributes of the corresponding tag. For example, in figure 3.2 the displayed height and width values correspond to the `height` and `width` attributes of the canvas. These displayed values verify the physical dimensions of the browser window. Since the `canvas` tag specifies no width or height values, they default to 100 percent of the browser window size.

Because the browser window is in full-screen mode and the display settings for our monitor are set to 1024 by 768 pixels, some height is lost to the browser's chrome header and a smaller amount is lost to the chrome frame sides. A screen-based ruler can be used to check these values. Although it isn't open source, the screen ruler shown in figure 3.3 is freely available at http://www.mioplanet.com. With it, we can verify that our browser window has a width and height of 1000 by 575 pixels, which matches the debugger values.

Other general information is also available from the debugger. For example, we can find the build number of the OpenLaszlo Server, the version number for our runtime file, or the application mode (server or SOLO) of our application.

Figure 3.3 The pixel ruler, which is movable across your screen, is an invaluable tool for measuring distances. It's available free at http://www.mioplanet.com/products/pixelruler.

Updating with the debugger

The debugger operates on an application's internal DOM, which means we can use the dynamic properties of JavaScript to change an object's properties. This allows us to program the debugger and immediately see the results displayed on the screen. For example, entering the following JavaScript code into the debug window

```
canvas.setAttribute("bgcolor", 0xCCCCCC);
```

changes the background color of the canvas to a light shade of gray. This ability to dynamically update DOM properties is a powerful tool that allows you to test code fixes on a live, deployed application.

3.1.2 Commenting your code

LZX code contains declarative tags and JavaScript statements, so we need a syntax for comments that is specific to each of these. For example:

```
<canvas debug="true">
   <!-- Handler for initialization code -->
   <handler name="oninit">
      // Expanded display of the canvas
      Debug.inspect(canvas);
   </handler>
</canvas>
```

The comment for the tag, of the form `<!-- … -->`, conforms to XML rules. The comment for the debug statement, beginning with `//`, conforms to JavaScript rules. Let's review these rules.

XML comments

Comments that appear between, but not within, tags in LZX statements must be in the form of XML comments:

```
<!-- comment -->
```

A comment may spread across multiple lines and terminates only when it encounters its end delimiter.

JavaScript comments

JavaScript supports more comment styles than is necessary, but this is more a heritage issue than style. In all, JavaScript supports three different styles of comments, shown in table 3.2.

Notice that JavaScript ignores the closing characters, `-->`, of HTML-type comments, so they behave differently than similar-looking XML comments. It's easy to confuse these two types of comments, so you should probably avoid using HTML comments in a JavaScript method. Instead, use the C++ style for single-line comments.

Table 3.2 JavaScript supports three different styles of comments.

JavaScript Comment Form	Derivation of Form

`/*`
`Comment`
`*/`
	C-style multi-line comments, where everything between the opening `/*` and closing `*/`, including newline statements, is considered to be part of the comment

`// comment`	Single-line comments as found in C++
`<!-- comment`	Single-line HTML-type comments

LZX comments

Since neither HTML nor JavaScript allows nested comments, Laszlo added the `ignore` XML processing instruction to allow large sections of code to be commented out, without having to worry about encountering comment end delimiters:

```
<?ignore
…
?>
```

We're just about finished with the prerequisite material for creating and working with an example application. Just one more topic needs to be covered: protecting special characters within LZX code.

3.1.3 Well-formed XML files

Because an LZX application is a well-formed XML document, all its nodes, including those containing JavaScript methods, must be valid XML. JavaScript code is enclosed within tags and is thus protected from XML compliance requirements. Nevertheless, this JavaScript code might contain the special XML characters listed in table 3.3, resulting in a compile-time error. When these special characters are present, the entire method must be protected by enclosure within a CDATA declaration or the individual characters must be escaped by using the character sequence substitution shown in the second column of table 3.3.

Table 3.3 Certain XML special characters require protection in JavaScript code to avoid a compile error. The second column gives the substitution for escaping an individual character.

Character	LZX Substitute	Reason for the Conflict
`<`	`&lt;`	Begins an XML tag
`>`	`&gt;`	Ends an XML tag
`&`	`&amp;`	Begins an XML character code

To escape an entire event handler or method, you can wrap the entire JavaScript section of code in opening `<![CDATA[` and closing `]]>` tags, as shown in the following example:

```
<canvas debug="true">
   <handler name="oninit">
      <![CDATA[
      Debug.write("<&>");
      ]]>
   </handler>
</canvas>
```

These awkward and easily forgettable escape sequences are an unfortunate byproduct of using XML and can't be blamed on LZX. We suggest that you sidestep the issue by using the *abbreviations* feature available in most text editors. That way, you don't waste time and neurons memorizing these sequences.

We are now proficient at creating a blank canvas for an application, commenting this short piece of code with both XML and JavaScript comments, and examining its properties in the debugger. Our toolkit is now sufficiently equipped to examine the infrastructure beneath an LZX application.

3.2 Creating object hierarchies

This section builds on the previous section by adding declarative tags to our minimal application. We'll examine the effects of declaratively versus dynamically creating objects. Afterward, we'll look at the different hierarchies for containing objects. Finally, we'll see how these hierarchies are stored internally by LZX objects.

In chapter 2, you saw that a core feature provided by the `LzNode` class is support for global and local namespaces; now we'll use this support in a program.

3.2.1 Naming objects declaratively

Declaratively instantiating an `LzNode` object is simple: just use it. Although not every node needs a name, sometimes it's necessary to reference a specific node. For these cases, an `id` attribute serves as a *global identifier*, as shown in listing 3.1.

Listing 3.1 Declaratively instantiating an `LzNode` object

```
<canvas debug="true">
   <node id="first"/>
</canvas>
```

An id is an identifier within the global namespace and can be referenced using the name global.first or simply first. However, it can't be accessed through the parent-child hierarchy as canvas.first.

If a node needs to be addressable through the parent-child hierarchy, then the name attribute provides a local identifier within its parent node (or container). A local identifier can be used in either an absolute or relative reference. An absolute reference is a complete path starting at the canvas, while a relative reference starts from the current location within the parent-child hierarchy. Listing 3.2 shows both forms.

Listing 3.2 Relative and absolute addressing

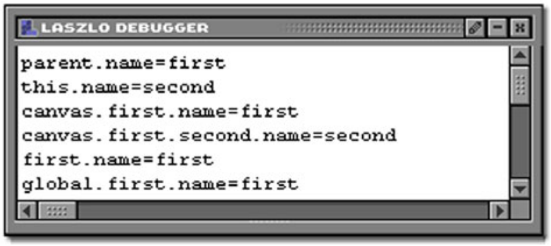
```
<canvas debug="true">
   <node name="first">
      <node name="second">
         <handler name="oninit">
            Debug.write("parent.name=" + parent.name);
            Debug.write("this.name=" + this.name);
            Debug.write("canvas.first.name=" + canvas.first.name);
            Debug.write("canvas.first.second.name=" +
                                       canvas.first.second.name);
            Debug.write("first.name=" + first.name);
            Debug.write("global.first.name=" + global.first.name);
         </handler>
      </node>
   </node>
</canvas>
```

❶ Specifies relative reference

❷ Specifies absolute reference

❸ Specifies top-level names referenced either way

In listing 3.2, ❶ and ❷ demonstrate relative and absolute addressing, but ❸ is a special case. As a general rule, all top-level declarative tags have an entry in the global table. So the first node has both a global and a local identifier and, as seen in figure 3.4, is accessible through either relative or global references.



```
parent.name=first
this.name=second
canvas.first.name=first
canvas.first.second.name=second
first.name=first
global.first.name=first
```

Figure 3.4
Using relative, absolute and top-level references

NOTE *A word about absolute and relative referencing* The structure of an absolute or a relative reference is somewhat complex. If you are a bit puzzled by the string `first.name` in the previous example, then this discussion is for you; saying `first.name` is like saying "Mary's name" rather than "Mary." The key to interpreting a reference is to understand that it has three parts: a *qualifier*, a *string of name values*, and the *name of a value*. The qualifier can be empty, the keyword `canvas`, or a possibly empty dotted string of `parent` keywords. The string of name values is a possibly empty dotted string of valid values of name attributes. The final value name is a valid attribute name.

For example, in the string `canvas.first.name` each component is a different sort of animal. The keyword `canvas` is the qualifier. The string `first` is the value of the `name` attribute. And finally, `name` references an attribute in the named node. If all this seems twisted, don't worry; it's actually quite intuitive once you have seen a few references. It's even more complex since the final value `name` can be a function—but we won't go there right now.

Laszlo currently only supports a single namespace, so the number of meaningful names is limited. Although a prefix can be added to a name to simulate a sense of namespace, the programmer is responsible for maintaining such a disciplined approach. LZX is case sensitive, so the names `gid`, `Gid`, and `GID` are all distinct. The cardinal rules of naming are

- Be consistent with case.
- Use global names sparingly.

So let's now see how to create named nodes dynamically.

3.2.2 Creating nodes dynamically with JavaScript

A node object or any other LZX object can also be dynamically instantiated. This allows its existence to be dependent on an outside occurrence. In contrast, a declarative object is always instantiated. The general form for instantiating an `LzNode` object looks like

```
var newnode = new LzNode(parent,args);
```

- Where `newnode` is a local name for the new `LzNode` object.
- `parent` is the parent node for the new `LzNode` object. If the parent is set to `null`, then the new object is placed under the canvas.
- `args` is an associative array—i.e., a dictionary—whose name/value pairs specify attributes of the new node.

Each of these arguments is optional; omitting them produces a node with no attributes whose parent is the canvas. In this example, a new node named `main` is instantiated only if the button is clicked to generate an `onclick` event, as shown in listing 3.3.

Listing 3.3 Dynamic instantiation of node `main`

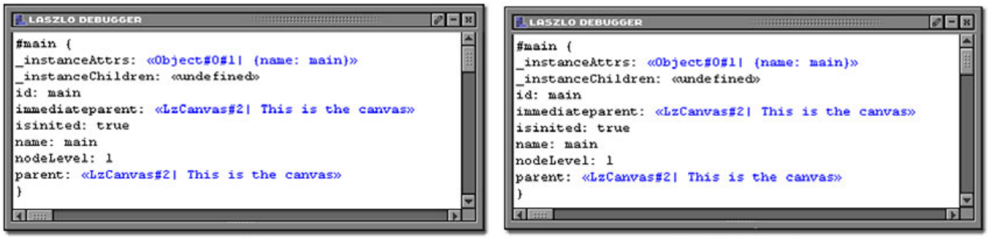
```
<canvas debug="true">
   <button text="Push">
      <handler name="onclick">
         var mynode = new LzNode(canvas, {name: "main"});
         Debug.inspect(canvas.main);
      </handler>
   </button>
</canvas>
```

The variable `mynode` is a local JavaScript variable, so its scope is limited to the method. JavaScript variables, whether local or global, won't be displayed within the debugger, since only the DOM contents are displayed. For comparison, listing 3.4 shows a declarative instantiation of a node `main`.

Listing 3.4 Declarative instantiation of node `main`

```
<canvas debug="true">
   <node id="main"/>
</canvas>
```

Comparing the debugger display from listings 3.3 and 3.4 (see figure 3.5), we see Laszlo's DOM is identically updated for both declarative and dynamic node instantiation.



```
#main {
_instanceAttrs: «Object#0#1| {name: main}»
_instanceChildren: «undefined»
id: main
immediateparent: «LzCanvas#2| This is the canvas»
isinited: true
name: main
nodeLevel: 1
parent: «LzCanvas#2| This is the canvas»
}
```

Declarative Version

```
#main {
_instanceAttrs: «Object#0#1| {name: main}»
_instanceChildren: «undefined»
id: main
immediateparent: «LzCanvas#2| This is the canvas»
isinited: true
name: main
nodeLevel: 1
parent: «LzCanvas#2| This is the canvas»
}
```

JavaScript Version

Figure 3.5 Both the declarative and the JavaScript versions of creating a node produce the same debug output, indicating that the DOM is updated identically.

Although tags and objects are loosely equivalent, certain LZX tags perform functions that don't have a corresponding JavaScript method. For example, every LZX application begins with `<canvas>` and ends with `</canvas>`; however, there is no way to instantiate a canvas object using script. Conversely, many LZX objects, such as services, cannot be created with declarative tag notation.

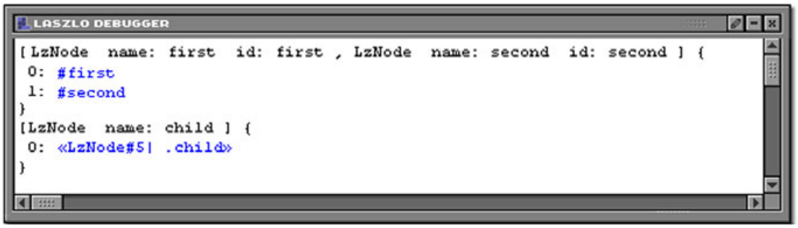
3.2.3 The subnodes array

One important attribute of the `LzNode` class is its *subnodes* array, which contains a set of references for its child nodes. Since the canvas object is derived from the `LzNode` class, it has a subnodes array. In the example in listing 3.5, a sibling relationship exists between the `first` and `second` nodes, and a parent-child relationship exists between the `second` and `child` nodes. Whenever a new node-based object is declaratively or dynamically instantiated, it is added to the parent node's subnodes array.

Listing 3.5 Declarative instantiation of parent-child and sibling relationships

```
<canvas debug="true">
   <node name="first"/>
   <node name="second">
      <node name="child"/>
   </node>
   <handler name="oninit">
      Debug.inspect(canvas.subnodes);
      Debug.inspect(canvas.second.subnodes);
   </handler>
</canvas>
```

The canvas's subnodes array, displayed in figure 3.6, contains two `LzNode` objects called `first` and `second`. Since the contained child node `second` is also a parent node, it too has a subnodes array containing its child node, called `child`.



```
[LzNode  name: first  id: first , LzNode  name: second  id: second ] {
 0: #first
 1: #second
}
[LzNode  name: child ] {
 0: «LzNode#5| .child»
}
```

Figure 3.6 The `subnodes` attribute defines the parent-child hierarchy in LZX. Here, the subnodes array for the canvas is displayed, followed by the contents of the subnodes array for the second node.

The subnodes array is used to handle references for relative addressing using the `parent` prefix with a parent-child hierarchy. Since each node maintains its own subnodes array, there is no central table to be updated.

The next section deals with the properties that define an object: its attributes, methods, and events.

3.3 Storing values in attributes

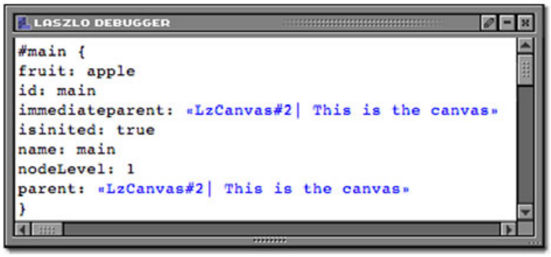
Being `LzNode`-derived blesses an object with the ability to store values in its attributes. Attributes are local data that define the operating characteristics of an object. We have already seen one attribute, `name`, that is used to store the local name of an object.

Laszlo supports two types of attributes: *base attributes,* which are class-specific and defined by Laszlo, and *user-specified attributes,* which are supplied by the user and are instance-specific. The `attribute` tag is used to create a user-specified attribute, as shown in listing 3.6. Attributes are declared as child tags within their parent element. While attributes can be dynamically created, accessing an undeclared attribute results in an "unknown attribute" compile-time error for a declarative tag and a runtime error in JavaScript. The debugger displays all attributes in an object.

Listing 3.6 Displaying attribute values

```
<canvas debug="true">
   <node name="main">
      <attribute name="fruit" value="apple" type="string"/>
   </node>
   <handler name="oninit">
      Debug.inspect(main);
   </handler>
</canvas>
```

Figure 3.7 shows the `fruit` attribute within the `main` node.



```
#main {
fruit: apple
id: main
immediateparent: «LzCanvas#2| This is the canvas»
isinited: true
name: main
nodeLevel: 1
parent: «LzCanvas#2| This is the canvas»
}
```

Figure 3.7 This debug output shows an abridged display for the `main` node containing a user-specified `fruit` attribute.

Attributes provide a way to communicate between declarative tags and JavaScript. When an attribute is created in a declarative tag, a new property is added to the object, which is accessible from JavaScript. Although the `this` prefix can be assumed, using `this` is considered a good programming practice as it indicates that the variable is a member of the enclosing object and not a local JavaScript variable. Although LZX tags support some class-based semantics, they don't support all the class-based access methods found in other object-oriented languages such as Java. All attributes are `public`, and there are no "private" or "protected" access modifiers to provide data privacy. The LZX compiler does, however, offer some access restrictions that we'll see in the next section.

3.3.1 Attribute types

LZX attributes come in several different types. All user-specified attributes are *settable*, since users create attributes to contain values and obviously need to set these values. The built-in base attributes can have any of the types read-only, final, and settable.

Read-only attributes

Read-only attributes, such as `parent` or `subnodes`, are set by the Laszlo system and, once set, can't be modified. They aren't accessible within a declarative tag; attempting to access one generates a compiler error. JavaScript can be used to initialize a read-only attribute but it can't later be modified. An example of its usage is when the parent-child hierarchy is extended with a dynamically instantiated node-based object. Here the node's `parent` attribute is set by the `new` constructor. Once a `parent` attribute is set, it can't be changed.

Final attributes

A final attribute is the inverse of a read-only attribute. It is set at compile time with a declarative tag and can't be modified by JavaScript. Once again, JavaScript can be used to set an uninitialized final attribute, but can't modify a previously set attribute. Two examples of final attributes are `name` and `id`, which are final to protect the DOM from corruption at runtime. Any attempt to modify a final attribute generates a runtime error.

Read-only and final attributes work together to provide a level of protection between the compile-time nature of declarative tags and the runtime behavior of JavaScript.

Settable attribute types

The set of data types supported by a programming language is one of its fundamental characteristics. JavaScript supports three primitive data types—`number`, `string`, and `boolean`—along with two composite data types—`object` and `array`. LZX extends the `string` type to support three additional types: `text`, `html`, and `css` (Cascading Style Sheets), resulting in a total of eight data types in LZX displayed in table 3.4.

Table 3.4 Attribute types

Type	Description
boolean	boolean (true or false)
color	CSS color (red, blue, yellow and other common color names)
css	CSS style
expression	JavaScript expression (the default type)
number	number
string	character text
text	character text
html	character text containing embedded HTML tags

LZX can leverage the dynamic features of JavaScript to postpone the setting of attribute types until runtime. In many cases, JavaScript can determine the type without having it explicitly stated. The example in listing 3.7, with output shown in figure 3.8, contains several such instances.

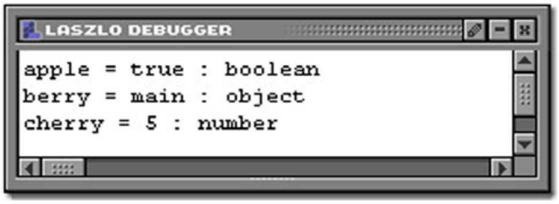
Listing 3.7 Dynamic typing

```
<canvas debug="true">
   <node>
      <attribute name="apple" value="true"/>
      <attribute name="berry" value="{name:'main'}"/>
      <attribute name="cherry" value="5"/>
      <handler name="oninit">
         Debug.write("apple = " + this.apple +
                     " : " + typeof(this.apple));
         Debug.write("berry = " + this.berry.name +
                     " : " + typeof(this.berry));
         Debug.write("cherry = " + this.cherry +
                     " : " + typeof(this.cherry));
      </handler>
   </node>
</canvas>
```

Sets type to boolean

Sets type to object

Sets type to number



```
apple = true : boolean
berry = main : object
cherry = 5 : number
```

Figure 3.8
JavaScript can determine type without having it explicitly stated. This dynamic typing capability is illustrated here in a case in which the correct type is determined because the value is only applicable to a particular type.

The attributes in listing 3.7 are set to their respective types because the value is only applicable to a particular type.

The `type` option can also be used to coerce a value to a particular type. In listing 3.8, the plus operator is overloaded to work differently depending on the type of its operands.

Listing 3.8 Coercing a value to a specific type

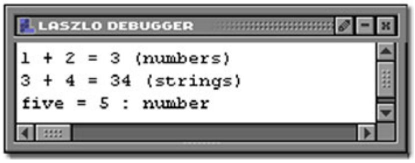
```
<canvas debug="true">
   <node>
      <attribute name="one"   value="1" type="number"/>
      <attribute name="two"   value="2" type="number"/>
      <attribute name="three" value="3" type="string"/>
      <attribute name="four"  value="4" type="string"/>
      <attribute name="five"  value="5" type="boolean"/>
      <handler name="oninit">
         var sum = this.one + this.two;
         Debug.write("1 + 2 = " + sum + " (numbers)");
         var sum = this.three + this.four;
         Debug.write("3 + 4 = " + sum + " (strings)");
         Debug.write("five = " + this.five +
                     " : " + typeof(this.five));
      </handler>
   </node>
</canvas>
```

First plus adds

First plus concatenates

Value supersedes type

As shown in figure 3.9, numeric operands are added and strings are concatenated. But in the final value, there is a conflict between its value five and its `boolean` type; the value supersedes the specified type, with the end result that the type is set to `number`.



```
1 + 2 = 3 (numbers)
3 + 4 = 34 (strings)
five = 5 : number
```

Figure 3.9
In general, numeric operands are added, while strings are concatenated. When there is a conflict between a value's natural type and its specified type, the value's type is set to its specified type. In this example, the value was set to five and the type was set to `boolean`, so the resulting type is `number`.

Specifying the type is required for string-valued attributes. If the `type` option is omitted, JavaScript defaults to interpreting a string value as a JavaScript object. This usually produces a runtime error, since a correspondingly named object can't be found. In the following section, we demonstrate how to work properly with JavaScript type expressions.

3.3.2 JavaScript type expressions

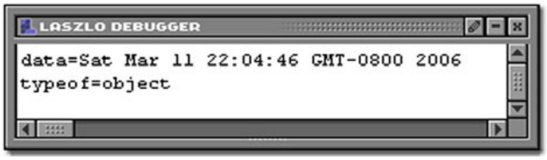
In addition to the primitive types, attributes can work with JavaScript expressions, allowing any data type to be dynamically set. By default, an attribute's type field is set to `expression`, allowing attributes to be initialized by the value returned from a JavaScript expression. To initialize an attribute named `date` to the current date, we would write

```
<attribute name="date" value="new Date()" type="expression"/>
```

The type field can even be omitted since it's the default:

```
<canvas debug="true">
   <attribute name="date" value="new Date()"/>
   <handler name="oninit">
      Debug.write("data=" + canvas.date);
      Debug.write("typeof=" + typeof(canvas.date));
   </handler>
</canvas>
```

In either case, the attribute named `date` contains an object with the current date expressible as a string, as seen in figure 3.10.



```
data=Sat Mar 11 22:04:46 GMT-0800 2006
typeof=object
```

Figure 3.10
The default type of a JavaScript expression is that of a value returned from a JavaScript operation.

An attribute's value can even be expressed as a constraint, as shown in listing 3.9, with the type determined by the return value of the JavaScript expression.

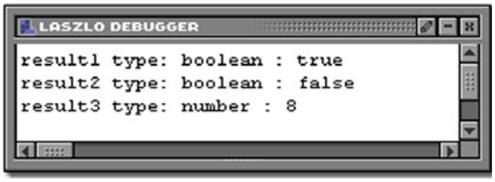
Listing 3.9 Using a constraint to assign to and dynamically type an attribute

```
<canvas debug="true">
   <attribute name="num1" value="32" type="number"/>
   <attribute name="num2" value="24" type="number"/>
   <node>
      <attribute name="result1"
                 value="${parent.num1 > parent.num2}"/>
      <attribute name="result2"
                 value="${parent.num1 &lt; parent.num2}"/>
```

Returns boolean

```
      <attribute name="result3"
                 value="${parent.num1 - parent.num2}"/>    <— Returns number
      <handler name="oninit">
         Debug.write("result1 type: " + typeof(result1) + " : " + result1);
         Debug.write("result2 type: " + typeof(result2) + " : " + result2);
         Debug.write("result3 type: " + typeof(result3) + " : " + result3);
      </handler>
   </node>
</canvas>
```

The return types, shown in figure 3.11, demonstrate how the attribute type is dependent on the operator used in the constraint.



```
result1 type: boolean : true
result2 type: boolean : false
result3 type: number : 8
```

Figure 3.11 This output demonstrates how the attribute type leverages the dynamic interpretative features of JavaScript.

Dynamic typing provides flexibility not available with statically typed languages—where all type information must be available at compile time. This flexibility allows LZX to be more easily programmed since it can dynamically respond to application changes. In a sense, its attributes are alive, while in a statically typed language, they are set in stone.

However, this raises the potential for type-related problems. Since JavaScript is a loosely typed language, it has flexibility at the expense of rigor. JavaScript converts a value to its correct type based on context. For example, if a string is used in a numeric context, it is automatically converted to a number. But in many cases there is insufficient information to determine type. For example, since the `setAttribute` method accepts numbers, strings, or objects as an argument, it relies on correct specification of the argument. This leads to common typing problems such as

```
this.setAttribute("width", "500");
```

instead of

```
this.setAttribute("width", 500);
```

Although JavaScript correctly converts values in most cases, it can't be relied on in all cases. Incorrect conversion can easily result in errors that are difficult to track down. Since the compiler allows loose typing, it is the responsibility of the developer to ensure that the correct type is passed.

Attributes represent the sending portion of the event-delegate communication system. In the next section, we'll cover the receiving portion represented by methods and event handlers. Whereas this section covered the general properties of attributes, the next section covers the communication-related aspects of attributes.

3.4 Methods and event handlers

Methods and event handlers work in tandem to handle all the imperative processing required in a Laszlo application. This processing encompasses all the supporting tasks that can't be performed with declarative notation. Methods and event handlers consist of a declarative LZX wrapper around JavaScript code to protect it from being parsed as XML statements. Because JavaScript can interact with declarative statements, we can leverage the descriptive capabilities of declarative statements to address new situations by updating their attributes.

Methods operate like the subprograms of other procedural languages, and their arguments can be passed by value or reference. Methods can be called by other methods, event handlers, or Laszlo itself. An event handler is a method that has been registered to receive events from a particular object. All the basic operations of methods are applicable to events. However, when we deal with classes, we will encounter some differences in how methods and event handlers are overridden and overloaded.

Although there are many surface similarities between methods and event handlers, there's a big difference in how they're used. A method is called and executed in the context of its object. Events can be sent by multiple objects in an indeterminate order, so event handlers need to be *idempotent*. This means the work performed by an event handler must produce the same effect regardless of whether it's called once or multiple times. This stipulation mandates that methods should be used to initiate events and event handlers should only process events.

Let's start with methods before we cover writing event handlers. We'll examine the relative timing of event handlers and the different types of events—base or attribute—that specify particular event senders. Finally, we'll show how objects communicate through events. This event communication serves as the basis for constraints, which are covered in the next section.

3.4.1 Writing methods

An LZX method has the following form:

```
<canvas>
   <method name="method_name" args="a,…,z"><![CDATA[
      var val_names;
      global_name = null;
      // JavaScript code
      return;
   ]]></method>
</canvas>
```

Contains local variable

Contains global variable

A method is called by its name method_name, is supplied with a number of arguments a,…, z, and can return a result. It is considered good practice to always protect the special characters within a method with a CDATA declaration. Local or global JavaScript variables can be declared. Now let's examine what is *not* shown in the previous example. First, there is no access specifier, so all LZX methods have public access. LZX provides no support for private or protected access modifiers, so all methods can be accessed by any element.

Second, there is no argument or return data type declaration, so the data types for arguments and return types can dynamically change. Once again, LZX takes advantage of the dynamic nature of JavaScript to provide the flexibility to handle different data types depending on runtime conditions. Depending on its type, an argument is passed by value or by reference.

We can pass fewer arguments than are specified in the args attribute. This allows some arguments to be optional, with default values set within the method. However, passing too many arguments generates a compile-time error.

The dynamic nature of JavaScript makes writing methods easy, since there is no need to specify method types, argument types, or return types. However, loose type checking can be a dangerous thing. The responsibility lies with the developer to ensure that the correct data types are used.

Let's now look at value and reference arguments.

Passing arguments by value

In our previous example, the variable val_names is local to the method because its definition begins with the keyword var. A local variable's scope is confined to the method. JavaScript does not support static variables, so there is no way to retain a local variable's value across invocations.

Any variable defined without the var keyword is global, and so is universally accessible. Since this can lead to unintended side effects, we recommend using local variables whenever possible. Listing 3.10 shows that variables containing primitive data types are always passed by value.

Listing 3.10 Passing primitive types by value

```
<canvas debug="true">
   <node name="node">
      <method name="test_method" args="val">
         val = 100;
         Debug.write("inside test_method, val: " + val);
         return val;
      </method>
   </node>
   <handler name="oninit">
```

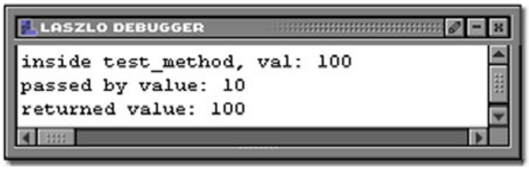
```
        var sum = 10;
        var ret = node.test_method(sum);
        Debug.write("passed by value: " + sum);
        Debug.write("returned value: " + ret);
    </handler>
</canvas>
```

Argument passed by value

Local value unchanged

Return value updated

The variable sum is initialized to 10 and its data type is set to number, a primitive data type. This value is passed into test_method, where the argument val is updated to 100. Since the argument is passed by value, after the call sum retains its original value of 10, as seen in figure 3.12.



```
inside test_method, val: 100
passed by value: 10
returned value: 100
```

Figure 3.12
Primitive data types are passed by value. This is demonstrated in this debug output, with a value that was initialized to 10 and then updated in the method to a value of 100. After the call, it still contains the value 10.

In the next section, you'll see that passing arguments by reference is just as simple as passing them by value.

Passing arguments by reference

Because there are no declared argument types, only objects are passed by reference. Since an array in JavaScript is considered to be an object, this supports both regular and associative arrays. Objects passed by reference can have individual attribute values updated. The example in listing 3.11 demonstrates passing an argument by reference.

Listing 3.11 Passing by reference

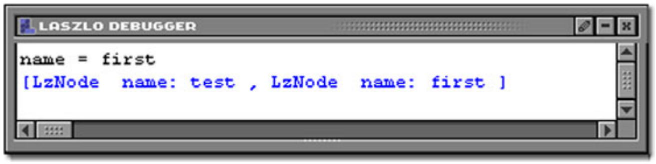
```
<canvas debug="true">
    <node name="top">
        <node name="test">
            <method name="updateName" args="node">
                node.setName("first");
            </method>
        </node>
        <handler name="oninit">
            var new_node = new LzNode(canvas.top);
            test.updateName(new_node);
            Debug.write("name =", new_node.name);
            Debug.write(canvas.top.subnodes);
        </handler>
    </node>
</canvas>
```

Statically creates node

Updates reference argument

Dynamically creates child of canvas

Passes argument by reference



```
name = first
[LzNode  name: test , LzNode  name: first ]
```

Figure 3.13 The updated subnodes array contains a new node named `first`.

On application initialization, a new node is dynamically instantiated as a child of the canvas and a sibling to the node `test`. Since this node doesn't have a name, the method `updateName` is used to update its name. After the call, the updated value of the `name` attribute, displayed in the subnodes array, can be seen in figure 3.13. JavaScript is able to update this read-only attribute, `name`, because it is uninitialized.

JavaScript passes objects and arrays by reference and primitive data types by value, as is common in other languages. Consequently, most developers should feel comfortable using LZX's methods.

An event handler operates just like a method except that its execution is triggered by an event. So let's see the implications of this difference.

3.4.2 Writing event-handler methods

An event handler operates as a subscriber in publisher-subscriber-based communications. An event handler is a method, whose `event` tag alerts the Laszlo compiler that it should be automatically registered as a subscriber to receive a particular event from a particular publisher. Its `name` attribute specifies the type of events that it receives. In its simplest form, it only receives events from its enclosing object. But a `reference` attribute can be added to specify other publishers.

Two types of events can be handled: system events and attribute events. A system event is defined by the enclosing object type, while an attribute event is generated by a change to an attribute of an object. All `LzNode`-derived objects can receive the base system events listed in table 3.5.

Table 3.5 The system events registered to all `LzNode`-derived objects are `onconstruct`, `ondata`, and `oninit`.

Event	Description
`onconstruct`	Sent at the end of the instantiation process, but before any subnodes have been created
`ondata`	Sent when the data selected by the node's data path changes
`oninit`	Sent immediately before the node becomes active

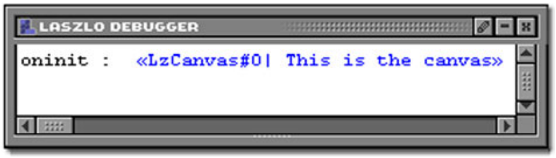
We'll defer looking at the ondata and onconstruct events until later chapters that deal with datasets and optimization and start by examining the oninit event.

Handling system events

The oninit event is sent by all LzNode-derived objects when they have completed their initialization. These events have the form on+method. For example, the oninit event is associated with the completion of the init method. All events have a single argument; the oninit event's argument contains a reference to the object that sent the event. Since the canvas is LzNode-derived, it is automatically registered to receive oninit events:

```
<canvas debug="true">
   <handler name="oninit" args="s">
      Debug.write("oninit : " , s);
   </handler>
</canvas>
```

This event handler is idempotent, since it can be executed multiple times without affecting application state. The canvas object generates and handles the event, passing a reference to itself as an argument. It is sent when the canvas completes its initialization, resulting in the display of the debug message, shown in figure 3.14.



```
oninit :  «LzCanvas#0| This is the canvas»
```

Figure 3.14
All events provide a single argument containing a reference to the object that sent the event.

Every event handler is associated with a declarative tag. Event handlers at the top level of an application have no special properties and are associated with the canvas.

Handling attribute events

An attribute event, of the form on+attribute, is associated with each of an object's attributes. For example, whenever the setAttribute method is used to change the value of an attribute width, it generates an onwidth event.

Attribute-based events provide a single argument containing the value of the attribute. Alternatively, the value can be accessed directly from the sending object, as shown in listing 3.12.

Listing 3.12 Event arguments

```
<canvas debug="true">
   <attribute name="fruit" value="orange" type="string"/>
   <handler name="oninit" args="s">
```

Declares and initializes attribute ❶

```
      this.fruit = "pear";
      this.setAttribute("fruit", "apple");
   </handler>
   <handler name="onfruit" args="s">
      Debug.write("onfruit : " + s);
      Debug.write("onfruit : " + this.fruit);
   </handler>
</canvas>
```

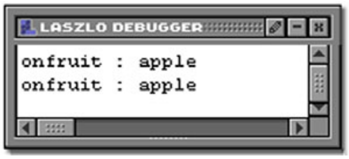
❷ Updates attribute

❸ Generates onfruit event

❹ Specifies onfruit event handler

In listing 3.12, when initializing ❶ or directly updating ❷, the attribute doesn't generate an event. An event is only sent when the `setAttribute` method is used ❸ to change its value. In figure 3.15, we see that the updated value can be accessed either through the event handler's argument ❹ or directly through the attribute itself.

Let's now take a look at how event generation affects processing.



```
onfruit : apple
onfruit : apple
```

Figure 3.15 Attribute event handling occurs when an attribute's value is updated with the `setAttribute` method. The updated value can be accessed either through an argument or through the object's attribute itself.

Event-handler timing

When an event is generated, current processing pauses until all scheduled event handlers have completed. This works like a stack; if an event handler generates an event, that handler pauses until all subsequent events have been handled. Listing 3.13 shows an example.

Listing 3.13 Generating and handling user events

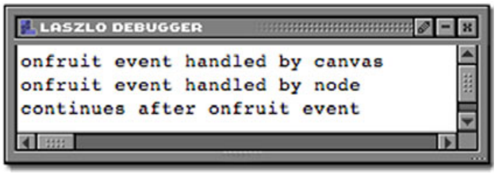
```
<canvas debug="true">
   <node name="first">
      <attribute name="fruit" value="apple" type="string"/>
      <handler name="onfruit" args="f">
         Debug.write("onfruit event occurs first : "
                                              + this.fruit);
      </handler>
   </node>
   <handler name="oninit">
      first.setAttribute("fruit", "orange");
      Debug.write("continues after onfruit event");
   </handler>
</canvas>
```

❶ Executes immediately when fruit changes

❷ Generates event onfruit

❸ Executes after onfruit event handler

When the attribute `fruit` is updated ❷ with `setAttribute`, processing immediately continues ❶ in the handler. After the event handler completes, the debug statement at ❸ is executed. Figure 3.16 shows the debugger output for listing 3.13.



```
onfruit event handled by canvas
onfruit event handled by node
continues after onfruit event
```

Figure 3.16 A user-specified event called fruit is handled by the onfruit event handler.

Up to now, we've only looked at situations where an object handles self-generated events. Naturally enough, we'd like to expand our use of event handlers to handle events sent from other objects.

Handling events from other tags

For an event handler to receive an event from an external object, it's only necessary to add a `reference` attribute to specify the object sending the event. Listing 3.14 shows an updated version of an earlier example.

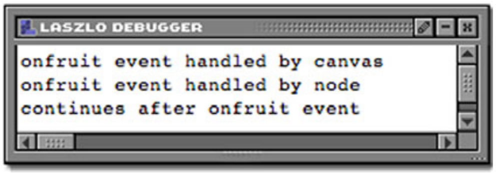
Listing 3.14 The `reference` attribute in handlers

```
<canvas debug="true">
   <node name="first">
      <attribute name="fruit" value="apple" type="string"/>
      <handler name="onfruit" args="s">
         Debug.write("onfruit event handled by node");
      </handler>
   </node>
   <handler name="oninit">
      first.setAttribute("fruit", "orange");
      Debug.write("continues after onfruit event");
   </handler>
   <handler name="onfruit" reference="first" args="s">
      Debug.write("onfruit event handled by canvas");
   </handler>
</canvas>
```

Handle onfruit internally

Handle onfruit externally

The event is processed by both the `canvas` tag's and the `first` tag's `onfruit` event handlers. Since two event handlers receive this event, processing is paused until both have completed. Although, as figure 3.17 shows, the canvas's event handler is triggered first, there is no guarantee concerning the order of event handling.



```
onfruit event handled by canvas
onfruit event handled by node
continues after onfruit event
```

Figure 3.17 Processing is interrupted until all event handlers receiving an event have completed.

As you saw in the jet formation exercise in chapter 2, while built-in event handling is a valuable resource in many situations it is still too cumbersome. These situations require an even more succinct notation, a notation that can be directly embedded within declarative tags. A constraint is nothing more than an embedded event handler.

3.5 Declarative constraints

In chapter 2, you saw that within a parent-child hierarchy a limited number of visual attributes, listed in table 2.1, are propagated to their children. But in general, objects use events to communicate attribute value changes to other objects. When attribute-based event processing is simple, it is often more readable to embed these event handlers within the declarative tags themselves. Normally, declarative tags are static, unable to react to changes in their environment, but constraints allow certain dynamic capabilities to be expressed directly within declarative tags. This solution leads to LZX *constraints*.

Constraints are the "duct tape" of Laszlo, allowing disparate parts of an application to interconnect. Constraints can be used with either event-delegate or data-binding communications. They are used to set an attribute to the value returned from a JavaScript expression. In later sections, dealing with data paths and datasets, we'll also see how constraints are used to set an attribute to the value returned from a data path's XPath expression.

In this section, we'll explore different types of constraints and discuss when it is appropriate to use each type. Then, we'll examine the limitations of constraints.

3.5.1 The basics of constraints

A constraint is applied to an attribute so that its value is set by a function's return value. This function is expressed as a JavaScript expression. If this expression contains any attribute values, any changes to these attribute values causes the JavaScript expression to be executed and the constrained attribute to be updated. A constraint is declared within an attribute tag like this:

```
<attribute name="fruit" value="${expression}"/>
```

or as an inline attribute like this:

```
<window attribute="${expression}"/>
```

Let's look at an example to explore the features of constraints. Suppose that we have two nested windows, an inner and an outer window, and we want the inner window to automatically resize itself to reflect resizing of the outer window. To automatically reflect size changes to the outer window, the inner window needs to be constrained to the outer window's height and width (see listing 3.15).

Listing 3.15 Using constraints to maintain window size relationship

```
<canvas>
   <window resizable="true" height="150" width="150">
      <window width="${parent.width-25}"
              height="${parent.height-50}"/>
   </window>
</canvas>
```

In listing 3.15, the `width` and `height` attributes of the inner window are constrained to the corresponding attributes of its parent outer window. When the outer window is manually resized, its `width` and `height` attributes change to reflect this resizing, causing the `onwidth` and `onheight` events to be sent to the inner window. Now there is a relationship between these elements supporting a consistent margin, as seen in figure 3.18.

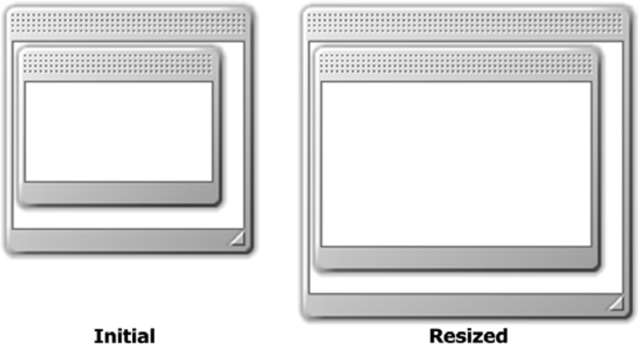


Figure 3.18 The `width` and `height` attributes of the inner window are constrained to the values of the corresponding attributes of its parent outer window. The inner window resizes itself to reflect any resizing of the outer window.

Now suppose that you aren't interested in maintaining a relationship between these windows, but only need to obtain the outer window's height to initially set the height of the inner window. This is where the `once` *modifier* comes into the picture.

3.5.2 The once modifier

The `once` modifier, applied to a constraint, indicates that the constrained value of the attribute is to be used only once, to initialize it. Listing 3.16 contains a short example.

Listing 3.16 Using a `once` constraint for initialization

```
<canvas>
   <window resizable="true" width="200">
      <attribute name="height" value="150"/>
      <window width="100" height="$once{parent.height-50}"/>
   </window>
</canvas>
```

The height of the inner window is initialized to a value based on the height of the outer window. The `once` modifier limits the effect of this constraint to initialization. Later during execution, the `height` attribute of the inner window is not affected by changes to the size of the outer window. Figure 3.19 shows this effect. As a result, a window margin is set but not maintained.

Using the `once` modifier in an attribute imposes a syntactic restriction. Since attribute values default to type expression, a string must be quoted; otherwise, it would be interpreted as a JavaScript variable. For example:

```
<attribute name="url" value="$once{parent.name + '.com'}"/>
```

Constraint-based relationships can also be applied dynamically with JavaScript. While not as elegant as using declarative tags, it is still a relatively simple matter to apply constraints with JavaScript.

There is still one interface point left to be covered. It is isolated here in its own section because it doesn't share the same initialization sequence as the other tags. As a result, it has only niche uses.

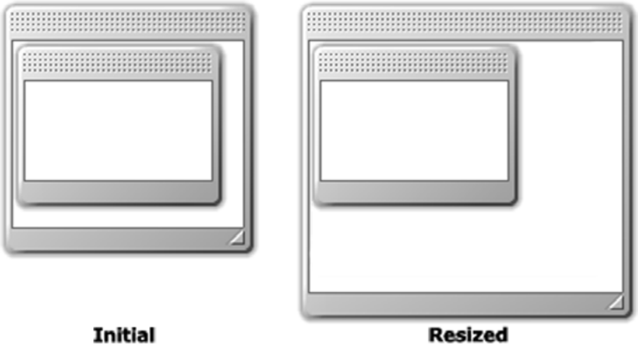


Figure 3.19 The `height` attribute of the inner window is declared with the `once` modifier, based on the value of its parent's height. After initialization, it doesn't respond to any change to its parent's height.

3.6 JavaScript and the script tag

The last of the interface points is the script tag, which provides another way to enclose JavaScript code within an application. But, unlike the method and handler tags, it isn't intended for general-purpose use. This much misunderstood tag is frequently the bane of novice Laszlo developers. At first glance, it appears to be a general-purpose tag for executing JavaScript code. However, since the contents of the script tag are evaluated early during the initialization sequence, the rest of the application hasn't completed its initialization and isn't ready to be accessed. If it isn't handled correctly, it can result in timing issues and produce unexpected results.

The purpose of the script tag is to serve as a repository for JavaScript libraries and global variables. Appropriate uses of script include containing an auxiliary sort function for declarative tags, setting global JavaScript variables, or containing legacy JavaScript libraries.

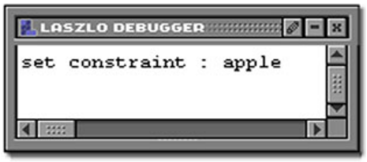
However, these uses don't preclude the contents of a script tag from interconnecting with an application. Once again, constraints can be used to communicate between declarative tags and JavaScript variables contained within a script tag. Listing 3.17 demonstrates this communication.

Listing 3.17 Using constraints to communicate between attributes and JavaScript variables

```
<canvas debug="true">
   <script>fruit = "apple"</script>
   <node>
      <attribute name="result" value="$once{fruit}"/>
      <handler name="oninit">
         Debug.write("set constraint : " + this.result);
      </handler>
   </node>
</canvas>
```

Executing this application produces the result shown in figure 3.20.

We've now completed reviewing how declarative tags and JavaScript can interface. In later chapters, we'll see additional ways that constraints can be used to interconnect with XPath statements to further enrich declarative statements.



```
set constraint : apple
```

Figure 3.20 Constraints allows local JavaScript variables to be accessible within declarative LZX tags.

Now that we've achieved a level of proficiency working with the "nuts and bolts" of LZX, we're ready to proceed toward working with visible objects derived from the `LzView` object.

3.7 Summary

This chapter focused on the declarative JavaScript interface of LZX and examined the connections that allow processing to pass from declarative tags into procedural JavaScript code.

We demonstrated the relative ease with which a small application can be created, debugged, and executed. We presented a basic set of tools that can be used to examine the properties of Laszlo's DOM. With these tools, you can begin to develop insights into the interactions between various program elements.

We introduced the main elements of the language interface: attributes, methods, and events. Since Laszlo is designed for building event-driven user interfaces, all processing originates with an event. Regardless of where an event originates from—user input or system related—the Laszlo system invokes an event handler to receive it. All event processing is performed in JavaScript methods, which can invoke other methods to handle common tasks.

Event handling also occurs in succinctly expressed event handlers known as constraints. The underlying mechanism is the same as for normal event handling, but the constraint notation allows a snippet of JavaScript code to be directly embedded within a declarative tag. This concise notation allows event-handling situations to be clearly specified without cluttering the overall logic of an application.

Constraints can also be used in a more general form of interconnection to allow declarative tags to reference other declarative tags, or even JavaScript variables. Although limited, this mechanism provides a convenient way to reference any language construct of LZX.

At this point, you've seen how to build a small application and have begun to appreciate the structural skeleton of LZX. In the next chapter, we'll introduce the `LzView` class. The `LzView` and `LzNode` classes serve as the fundamental classes of LZX.

4 A grand tour of views and user classes

This chapter covers

- Covering visual-based object basics
- Interacting with visual-based objects
- Creating user classes
- Dynamically instantiating JavaScript and LZX objects

I'm a great believer in luck and I find the harder I work, the more
I have of it.

—Stephen Leacock,
Canadian humorist (1869–1944)

Up to this point, we've focused on the general workings of LZX rather than the characteristics of any specific tag. Now that we have some familiarity with LZX's skeletal framework, let's flesh out this skeleton with an exploration of user interaction. Rather than split user interaction across a number of classes, the Laszlo language designers decided to concentrate it in a single JavaScript class called `LzView`. This class is derived from the `LzNode` class, so it can also be used as a `view` declarative tag. It serves as the superclass to support all visual objects within Laszlo.

In its simplest form, a view object is just an invisible container. But adding dimensions and a background color turns it into a rectangle. Since the view has full access to multimedia, a view can also be used to display an image, play a video or audio track, or specify fonts for text. Of course, it also supports positioning elements to control the placement of this display. A view also supports different ways for users to interact with it. This supports user peripherals such as keyboards, mice, and scroll wheels, and the different ways to control their access.

A strong command of the `LzView` API is essential for creating applications. Let's start tackling this complex API by splitting it into these major groupings:

- Physical dimensions
- Background and foreground colors
- Multimedia resources—images, audio and video, mouse cursor
- CSS mouse events
- Focus
- Font specification
- Selection of frames within a resource
- The playing of embedded audio and video
- Placement
- along the z-axis
- Relative and absolute positioning
- Rotational placement

Because this class has so many attributes and methods, we'll conduct a "grand tour," visiting each of these groupings. To help you digest all this information, we'll organize the features in each group around central themes such as "controlling visibility." These themes address high-level behaviors and will be discussed in subsequent chapters as well. Hopefully, this provides a more interesting presentation and a better sense of how these features are used. We'll conclude the tour by revisiting dynamic view instantiation and creating user-defined classes.

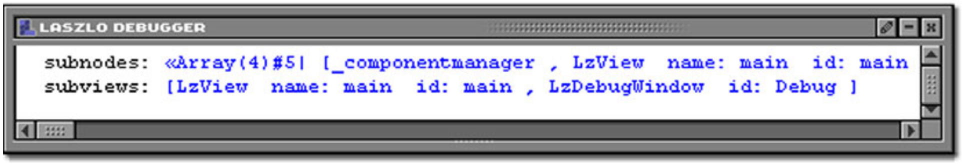
4.1 Introducing the basic features of views

While the `LzNode` class defines common behavior for all LZX declarative tags, the `LzView` class defines their visual and user interaction characteristics. The `LzView` class adds a *subviews array* to `LzNode`'s subnodes array. This array contains only view-based child nodes and is used to manipulate the arrangement of visual objects to produce spatial layouts and different visual effects. Listing 4.1 produces the debug output displaying both of the arrays in figure 4.1.

Listing 4.1 Displays both the subnodes and subviews arrays

```
<canvas debug="true">
   <view name="main">
      <handler name="oninit">
         Debug.write("subnodes: " + canvas.subnodes + "\n" +
                     "subviews : " + canvas.subviews);
      </handler>
   </view>
</canvas>
```

The canvas contains a `subviews` array containing the child view `main`. Since a view is also a node, `main` also appears in the `subnodes` array. The debug window is being displayed, so it also appears in these array listings.



```
subnodes: «Array(4)#5| [_componentmanager , LzView  name: main  id: main
subviews: [LzView  name: main  id: main , LzDebugWindow  id: Debug ]
```

Figure 4.1 Nodes and views are maintained within their respective parent-child hierarchies. These hierarchies are traversed through the `subnodes` array for nodes and through the `subviews` array for views. Since the debug window is currently displayed, it appears in the `subviews` array.

Now that we've seen the structure used to contain `LzView` objects, let's take a look at manipulating it to control visibility.

4.1.1 Controlling view visibility

You might think that declaring a view in a canvas should produce a visible result. But if we write

```
<canvas>
   <view name="top"/>
</canvas>
```

this only displays a blank screen. Although the view exists, it's not visible since it has nothing to display. For a view to be visible, it needs a minimum of a background color with positive `height` and `width` attributes or an attached media resource. Table 4.1 shows the requisite attributes to display a view. The minimal code to make a view visible looks like this:

```
<canvas>
   <view width="75" height="75" bgcolor="blue"/>
</canvas>
```

or—assuming that a logo.gif file exists in your local resources directory—something like:

```
<canvas>
   <view resource="resources/logo.gif"/>
</canvas>
```

In the second case, `height` and `width` attributes don't need to be specified since image resources have an implicit width and height. An important rule in Laszlo is that a parent view resizes to accommodate the size of its child views. This occurs even when the parent view has explicit dimensions smaller than its child.

TABLE KEY The tables used throughout this book provide an abridged overview of LZX attributes and methods. For complete information about methods, refer to the Laszlo system documentation available at www.openlaszlo.com.

Attribute tables consist of a name and data type, shown in the first two columns. The third column states whether the attribute can be accessed only within a tag, only within JavaScript, or both. The attribute type, in the fourth column, can be *settable* (it's writable), *read-only* (the obvious meaning), or *final* (it can't be changed at runtime). Some tables also provide a default value column, preceding the brief description. Method tables consist of the method call with its arguments followed by a short description.

Table 4.1 View attributes that establish visibility properties

Name	Data Type	Tag or Script	Attribute Type	Description
`bgcolor`	`number`	Both	Settable	The background color of the view; a number between 0 and 0xFFFFFF; also supports CSS color names
`height`	`number`	Both	Settable	The height of the view
`width`	`number`	Both	Settable	The width of the view
`resource`	`string`	Both	Settable	The name of the view's resource, or the URL from which the resource was loaded

Specifying an attribute from table 4.1 doesn't necessarily guarantee that the view will be visible. Let's look at some other attributes that affect view visibility.

Controlling visibility with clipping

The `clip` attribute controls rather than establishes visibility. Normally, a parent resizes itself to the sizes of its children, even when a child is partially or completely outside its dimensions. The `boolean` attribute `clip` turns off this behavior; see table 4.2.

Table 4.2 A view's `clip` attribute controls the visibility of its children.

Name	Data Type	Tag or Script	Attribute Type	Description
`clip`	`boolean`	`Tag`	`Final`	If true, any attached resource or child of the view is clipped to the parental width and height.

With `clip` set, a parent view rigidly imposes its dimensions on a child, cutting off any portion of the child view that lies outside its boundaries. The left display in figure 4.2 shows an unclipped view produced by the following:

```
<canvas>
   <view bgcolor="0xCCCCCC" width="100" height="100">
     <view bgcolor="0xDDDDDD" x="100" y="100"
        width="100" height="100"/>
   </view>
</canvas>
```

However, if the `clip` attribute is added to the parent view

```
<view bgcolor="0xCCCCCC" width="100" height="100" clip="true">
```

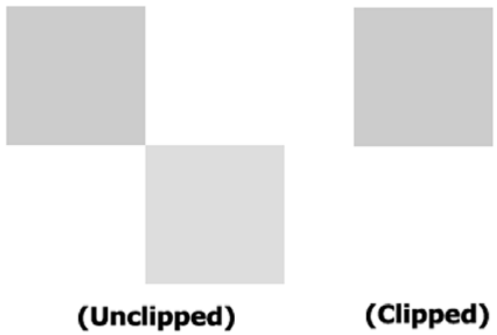


Figure 4.2
When a parent sets its `clip` attribute, the child views are clipped to the parent's dimensions. In this example, the child view is entirely clipped and loses visibility.

the child view loses visibility, as seen in the right display in figure 4.2. The child loses visibility since it is located entirely outside the bounds of its parent view.

Controlling visibility with the z-axis

If we think of the screen as a stack of translucent views, whose visibility is controlled through a third dimension, the z-axis, then views with a common parent can be reordered with the `bringToFront`, `sendBehind`, `sendInFront`, and `sendToBack` methods, shown in table 4.3. These methods can change a view's visibility by changing its order in the stack.

Table 4.3 A parent's subviews can be reordered on the z-axis using the methods listed here. Such a reordering can cause a subview to lose or gain visibility.

Name	Description
`bringToFront()`	Makes the invoking subview the foremost subview of the parent
`sendBehind(LzView)`	Puts the invoking subview behind the specified sibling
`sendInFrontOf(LzView)`	Puts the invoking subview in front of the specified sibling
`sendToBack()`	Makes the invoking subview the rearmost subview of the parent

In listing 4.2, clicking on any of the views causes the z-axis stack to be reordered. This results in some views becoming visible and hiding others.

Listing 4.2 Controlling visibility through placement

```
<canvas>
   <view bgcolor="red" width="100" height="100"
         onclick="bringToFront()"/>
   <view bgcolor="blue" x="50" y="50"
         width="100" height="100"
```

```
              onclick="sendBehind()"/>
        <view bgcolor="green" x="100" y="100"
              width="100" height="100"
              onclick="sendInFront()"/>
        <view bgcolor="yellow" x="150" y="150"
              width="100" height="100"
              onclick="sendToBack()"/>
     </canvas>
```

Specifying visibility directly

The visible and opacity attributes directly control a view's visibility. The visible attribute works like a light switch, causing a view's display to turn on and off. The opacity attribute works like a dimming light switch by specifying an incremental value from fully opaque (1.0) to transparent (0). A similar effect occurs if we set the x and y attributes to positions outside the screen. You'll see examples of these in the next section.

4.1.2 Controlling visibility with animation

Since the LzView class extends LzNode, which has a built-in animator, we can animate views. Any of the visibility-controlling attributes listed in table 4.4 can be animated to provide a gradual effect.

For example, suppose a square is to fade into nothingness over a period of 3 seconds. Listing 4.3 accomplishes this fading effect, whose results are shown in figure 4.3.

Table 4.4 The attributes listed here can be animated to control a view's visibility.

Name	Data Type	Tag or Script	Attribute Type	Description
height	number	Both	Settable	The height of the view
width	number	Both	Settable	The width of the view
x	number	Both	Settable	The horizontal position of the view relative to the upper-left corner of the parent view, measuring positive to the right
y	number	Both	Settable	The vertical position of the view relative to the upper-left corner of the parent view, measuring positive downward

Listing 4.3 Controlling visibility through opacity

```
<canvas>
   <view name="main" height="100" width="100" bgcolor="#dddddd">
      <handler name="oninit">
         main.animate("opacity", 0, 3000);
      </handler>
   </view>
</canvas>
```

❶ Fades to transparent

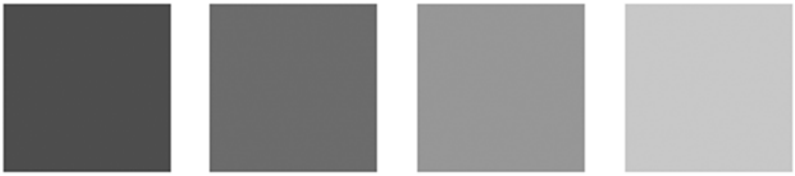


Figure 4.3 With time increasing from left to right, the opacity of a view is continuously reduced by an animator.

The animation ❶ changes the value of `opacity` from full opacity to transparency, 1.0 to 0, over a period of 3 seconds.

Animation allows us to vary the `x` and `y` attributes of a view so it moves like an actor on a stage to exit slowly to the left, right, or even up or down. The box example can be modified to animate the `x` attribute to an off-screen value of –100, thus producing an exit-stage-left effect:

```
<canvas>
   <view name="main" x="400" y="20"
         height="100" width="100" bgcolor="blue">
      <handler name="oninit">
         main.animate('x', -100, 3000);
      </handler>
   </view>
</canvas>
```

Changing the final value of `x` to a value greater than the screen dimensions causes a "stage right" exit.

The `width` and `height` attributes can be used to produce an "Alice in Wonderland" effect, whereby a view suddenly springs into view like a balloon or shrinks down into nothingness:

```
<canvas>
   <view name="main" x="400" y="20" bgcolor="blue">
      <handler name="oninit">
```

```
            main.animate('height', 100, 3000);
            main.animate('width', 100, 3000);
        </handler>
    </view>
</canvas>
```

Some view attributes are so frequently used that *convenience methods*, listed in table 4.5, are supplied for them. These convenience methods operate identically to `setAttribute`, but offer slightly better performance.

Table 4.5 Convenience methods are intended for manipulating a view's visibility.

Name	Description
`setX(number)`	Assigns a value to the x position of the view
`setY(number)`	Assigns a value to the y position of the view
`setHeight(number)`	Assigns a value to the height of the view
`setWidth(number)`	Assigns a value to the width of the view
`setOpacity(number)`	Assigns a value to the opacity of the view; a number between 0.0 (transparent) and 1.0 (opaque)
`setVisible(boolean)`	Sets, or resets, the visible attribute of the view and enables, or disables, any associated `clickregion` (see later in this chapter for more info on `clickregion`)

Many more transformations for views are available. In the next section, we'll see various ways in which a view can be rotated.

4.1.3 Animating with rotations

The `rotation` attribute controls rotation about an origin point, which defaults to the upper-left corner of the screen. Listing 4.4 rotates a box twice through 720 degrees over 3 seconds.

Listing 4.4 Rotating a box

```
<canvas>
    <view name="main" x="400" y="200"
            height="100" width="100" bgcolor="#DDDDDD">
        <handler name="oninit">
            main.animate('rotation', 720, 3000);
        </handler>
    </view>
</canvas>
```

Figure 4.4 shows the square rotating clockwise about its upper-left corner.

Specifying a negative angle for the rotation produces counterclockwise motion:

```
<handler name="oninit">
   main.animate('rotation', -720, 3000);
</handler>
```

Now that you've seen how to rotate a view around the default axis, let's discuss how to specify an axis.

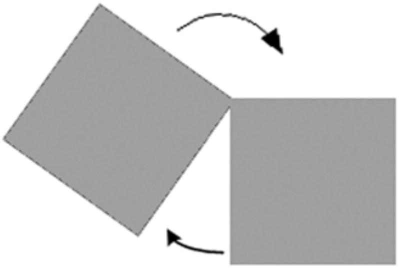


Figure 4.4 A view can be continuously rotated by animating its rotation attribute around an origin point, shown here as the default upper-left corner.

Rotating on a selected axis

The origin point for a view can be modified from its default upper-left corner setting with the xoffset and yoffset attributes. With a new origin set, a view rotates around the selected axis. Since, in the rotating box example, the dimensions of the view are 100 pixels on each side, assigning an xoffset of 100 causes the box to rotate around its upper-right corner:

```
<canvas>
   <view name="main" x="400" y="200"
         height="100" width="100"
         xoffset="100" bgcolor="blue">
      <handler name="oninit">
         main.animate('rotation', 720, 3000);
      </handler>
   </view>
</canvas>
```

Various combinations of xoffset and yoffset values can establish the origin point at any one of the four corners of the box, as seen in figure 4.5. Different effects can be achieved by specifying an origin other than a corner.

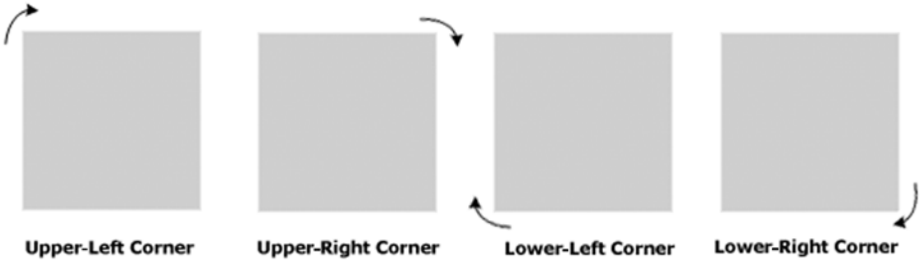


Figure 4.5 Changing the origin point of a rotation by assigning to the xoffset and yoffset attributes one of the combinations (0, 0), (0, width), (width, 0), or (width, width) causes the square to rotate around the specified corner.

Table 4.6 summarizes the attributes concerned with rotation. The pixellock attribute is applicable only to vector-based images (Flash or SVG) and turns on subpixel positioning, providing smoother animation along diagonal arcs. It is not applicable to DHTML.

Table 4.6 Four attributes support view rotation.

Name	Data Type	Tag or Script	Attribute Type	Description
`pixellock`	`boolean`	Both	Final	When set, supports subpixel positioning to smooth animation. When unset, makes the view snap to a pixel boundary.
`rotation`	`number`	Both	Settable	The rotation value for the view, in degrees; not restricted to the range 0 to 360.
`xoffset`	`number`	Both	Settable	The value to be added to the x position of the view to establish the point of origin before rotation.
`yoffset`	`number`	Both	Settable	The value to be added to the y position of the view to establish the point of origin before rotation.

We have shown you only some basic animation here. In chapter 13, we'll demonstrate how Laszlo adds features to this base to support more complex animation effects using sequential and simultaneous animators.

4.1.4 Adding multimedia resources

Although a view is technically always rectangular, if its attached image has a transparent background, a view assumes the shape of the image. A resource can contain a graphical image using any of the media types shown in table 4.7. These resources can be loaded at runtime or compiled into an application.

Table 4.7 Laszlo supports image, audio, font, and video (Flash) resource files.

Media Type	Description
Audio	MP3 at sample rates: 11.025 kHz, 22.5 kHz, and 44.1 kHz
Fonts	Embedded and native TrueType (TTF) files
Images	PNG, JPEG, and GIF files
Vector art	SWF

Table 4.7 Laszlo supports image, audio, font, and video (Flash) resource files. ***(continued)***

Media Type	Description
Vector animation	SWF
Video	SWF and FLV

A resource can be declared in one of two ways: as a `resource` tag or as an attribute of a view. Resources aren't derived from `LzNode`, so they must reside at the top level. The only settable attributes for a `resource` tag are `name` and `src`; `name` is a logical name for use by a `view` tag and `src` specifies a path to any of the media resource types given in table 4.7. Since it's only a path, it has no visible qualities of its own, such as position, size, or color. So simply declaring a resource doesn't result in its display.

For a resource to be displayed, it must be attached to a view through the view's `resource` attribute. If the resource is declared externally, then the resource's `name` attribute is referenced by the view. Here are two examples, first using an external resource declaration:

```
<canvas>
   <resource name="logo" src="images/logo.png"/>
   <view resource="logo"/>
</canvas>
```

and next only using the `resource` attribute:

```
<canvas>
   <view resource="resources/logo.png"/>
</canvas>
```

Notice that in both cases, the image file serving as the resource's `src` must be resident in the file system. This results in the resource being compiled into the application, which also increases the download size and time. However, this resource is now available to be displayed or played immediately at an application's start. Attempts to load an image file represented by an HTTP URL into a resource result in a compile-time error.

When a resource contains an audio file, there is no visible display associated with the resource. Despite this, the resource still works by being attached to a view. The pleasant result is that all media types operate similarly.

A view's resource-related attributes control loading, visual display, frame selection, and stretching, as well as provide information on the progress of loading. Table 4.8 summarizes these resource-related attributes.

Table 4.8 A view's resource-related attributes

Name	Data Type	Tag or Script	Attribute Type	Description
`resourceheight`	`number`	Script	Read-only	The height of the resource attached to the view.
`resourcewidth`	`number`	Script	Read-only	The width of the resource attached to the view.
`source`	`string`	Script	Settable	The URL from which to load the resource for the view; similar to the resource attribute but callable only from script.
`stretches`	`string`	Both	Settable	Determines a view's coordinate space so that contained resources and views fit exactly into the view's width and height; used to resize a view's contents by setting its width and/or height; valid values include height, width, and both; the default value is none.
`unstretchedheight`	`number`	Script	Read-only	If `stretches` is set to height or both, indicates the height for the view if it were not stretched; can be used to scale a view by a percentage of its original size or to determine the aspect ratio for a view.
`unstretchedwidth`	`number`	Script	Read-only	If `stretches` is set to width or both, indicates the width for the view if it were not stretched; can be used to scale a view by a percentage of its original size or to determine the aspect ratio for a view

Regardless of how a resource is loaded, an extensive set of view attributes is available to manipulate its display. For example, the height and width of a view's resource can be accessed through the `resourceheight` and `resourcewidth` attributes.

When dealing with image resources, it is frequently necessary to resize them to fit within a view. The `stretches` attribute causes an image to stretch or compress to conform to the view's dimensions. The image can be stretched by its height, its width, or both. But if an image is not uniformly stretched, then it will distort. To be uniformly stretched, the resized image's dimensions must be corrected with an aspect ratio. The `unstretchedwidth` and `unstretchedheight` attributes are available to determine this ratio:

```
ratio = unstretchedwidth/unstretchedheight
```

In chapter 6, you'll see an example of resizing an image without distorting it.

Finally, when resources need to be obtained over the Web, they can either be accessed through their URL by the resources attribute:

```
<canvas>
   <view resource="http://www.google.com/intl/en/images/logo.gif"/>
</canvas>
```

or the setSource method:

```
<canvas>
   <view>
      <handler name="oninit">
       this.setSource("http://www.google.com/intl/en/images/logo.gif");
      </handler>
   </view>
</canvas>
```

There is also a setResource method, which is the runtime equivalent of the resource attribute. But it can only access resident resources. Table 4.9 summarizes these resource-related methods.

Table 4.9 A view's resource-related methods

Name	Description
`setResource(src, cache, headers)`	Loads a resource at runtime using a resource name
`setSource(src, cache, headers)`	Loads a resource at runtime using a URL

The resource-related events, shown in table 4.10, provide opportunities for processing during resource loading; the three most important events are onload, onerror, and ontimeout. The onload event is generated when a resource has been successfully downloaded. The onerror event is generated if the network server returns an error. An instance of this would be the 404 error returned by a server unable to find a resource. An ontimeout event occurs when the network server times out. Finally, after a resource has been loaded by a child view, an onaddsubresource event is generated to allow postprocessing of that resource.

Table 4.10 A view's resource-related events provide opportunities for processing during resource loading.

Name	Description
`onaddsubresource`	Sent when a child view adds a resource.
`onerror`	Sent when there is an error loading the view's resource; the argument sent with the event is the error string sent by the server.

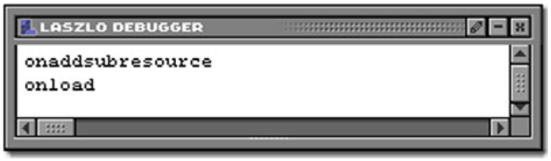
Table 4.10 A view's resource-related events provide opportunities for processing during resource loading. *(continued)*

Name	Description
`onload`	Sent when the view attaches its resource.
`ontimeout`	Sent when a request to load media for the view times out.

When local resources are compiled into the application, the resource-related events listed in table 4.10 won't be triggered. Listing 4.5 illustrates the generated event sequence when a resource is dynamically loaded, while figure 4.6 shows its event sequence.

Listing 4.5 Events generated by dynamic loading of resources

```
<canvas debug="true">
   <view>
      <view resource=
         "http://www.google.com/intl/en/images/logo.gif">
         <handler name="onload">
            Debug.write("onload");
         </handler>
      </view>
      <handler name="onaddsubresource">
         Debug.write("onaddsubresource");
      </handler>
   </view>
</canvas>
```



```
onaddsubresource
onload
```

Figure 4.6 The `onload` and `onaddsubresource` events are sent for resources accessed through HTTP.

A resource can have a single frame—an image—or multiple frames—a video or audio clip. For the latter, we need a way of selecting frames.

Selecting frames

For multiframe resources, the view's `totalframes` attribute provides the total number of frames. A multiple-frame resource is represented as an array, indexed

Table 4.11 Four view attributes describe a downloaded multiframe video or audio file resource.

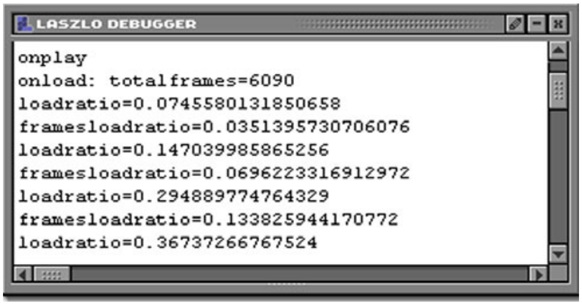
Name	Data Type	Tag or Script	Attribute Type	Description
`frame`	`number`	Both	Settable	Selects the frame displayed by the resource associated with the view, a number between 0 and `totalframes`
`framesloadratio`	`number`	Script	Read-only	For a view whose resource is loaded at runtime, the ratio of loaded frames to total frames; a number between 0 and 1
`loadratio`	`number`	Script	Read-only	For a view whose resource is loaded at runtime, the ratio of loaded bytes to total bytes; a number between 0 and 1
`totalframes`	`number`	Script	Read-only	The total number of frames for the view's resource

in the range 0 to `totalframes-1`. Notice in table 4.11 that the only settable attribute is `frame`. The others are read-only, since they specify physical characteristics of a multiframe resource.

The `frame` attribute is set to reference a particular frame; since Flash restricts a resource to 16,000 frames, that establishes the range of `frame`. Here's an example to demonstrate frames; its output is shown in figure 4.7:

```
<canvas debug="true">
   <view id="player" play="true"
      resource="http://www.themeatrix1.com/meatrix.swf">
      <handler name="onframe">
         Debug.write("loadratio=" + this.loadratio);
         Debug.write("framesloadratio=" + this.framesloadratio);
      </handler>
      <handler name="onload">
         Debug.write("onload: totalframes=" + this.totalframes);
      </handler>
      <handler name="onplay">
         Debug.write("onplay");
      </handler>
   </view>
</canvas>
```

Since the `play` attribute is set, this causes the video to immediately start playing after it has completed downloading. The `onplay` event occurs before the `onload` event, because it needs to determine whether the resource is local or needs to be



```
onplay
onload: totalframes=6090
loadratio=0.0745580131850658
framesloadratio=0.0351395730706076
loadratio=0.147039985865256
framesloadratio=0.0696223316912972
loadratio=0.294889774764329
framesloadratio=0.133825944170772
loadratio=0.36737266767524
```

Figure 4.7 The `loadratio` and `framesloadratio` attributes display the downloading progress for a video, expressed as a ratio. When the video has completed downloading, the ratio is 1.0.

loaded over HTTP. The `onload` event handler can be used for postprocessing functions, such as displaying the total number of frames. The `loadratio` and `framesloadratio` attributes keep track of the ratio of downloaded bytes and frames, respectively, to the total in the video—the total number of frames is available through the `totalframes` attribute. These ratios, as counters from 0 to 1.0, can be used to drive a progress bar.

Once an audio or video resource has been loaded, you'll naturally want to control it from within its view. The `play` attribute, shown in table 4.12, is used to control automatic playing upon loading for both audio and video.

Table 4.12 A view has only one audio- and video-related attribute.

Name	Data Type	Tag or Script	Attribute Type	Description
`play`	`boolean`	Both	Settable	If true, the resource begins playing when it is loaded.

Embedded Flash videos

Although progressive and streaming video is preferred over embedded video for viewing, it is available for niche cases. Laszlo comes with a short Flash SWF video clip, found in the laszlo-explorer/basics/jfk.swf directory of the OpenLaszlo server installation, which is used as the video resource in listing 4.6. This resource can be compiled into the application or downloaded, but both cases require that it be loaded in its entirety before playback begins. In all the examples in this book, we default to using port 8080 with the latest Laszlo version. You can change these settings to reflect your environment.

Listing 4.6 Play and Stop buttons in a simple video player

```
<canvas>
   <view id="player" play="false"
      resource="http://localhost:
               8080/lps/book/jfk.swf">
      <handler name="onload">
         play_btn.setVisible(true);
         stop_btn.setVisible(true);
      </handler>
   </view>
   <button name="play_btn" x="10" y="170" visible="false"
           onclick="player.play()">Play</button>
   <button name="stop_btn" x="70" y="170" visible="false"
           onclick="player.stop()">Stop</button>
</canvas>
```

Resets play; pauses clip

Accesses local file

Starts clip

Stops clip

A dynamically loaded video clip requires that the `onload` event be sent before playback begins. To ensure that the user doesn't get trigger-happy and try to play a clip before it has been loaded, the default visible states of the Play and Stop buttons are set to false. They become visible only when the video clip has completed loading. Figure 4.8 shows the video and control button display for the player in listing 4.6.

Table 4.13 summarizes the events for controlling audio and video resources. The `onframe` event is generated continuously while a multiframe resource is in the play state. On the last frame, both the `onframe` and `onlastframe` events are generated. The remaining events, `onplay` and `onstop`, denote when the player begins and ends playing.

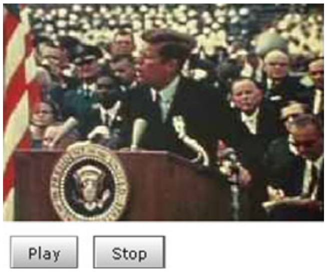


Figure 4.8 A simple video player like this can be produced with six lines of code.

Table 4.13 A view's audio- and video-related events

Name	Description
`onframe`	Sent from each frame during playback; this corresponds to the issuing of `onidle` events.
`onlastframe`	Sent when the view sets its fram—for example, resource number—to the last frame; can be used to determine when a streaming media clip is completed.

Table 4.13 A view's audio- and video-related events *(continued)*

Name	Description
`onplay`	Sent when a view begins playing its resource.
`onstop`	Sent when a view resource, capable of playing, is stopped; sent only if stop is called directly; when a resource hits its last frame, the `LzView` event `onlastframe` is sent.

Most of the methods related to controlling audio and video resources—see table 4.14—correspond to the operations start, stop, pause, volume, and others, available on a standard video or audio player.

At this point, don't worry too much about the details of these attributes, events, and methods. Instead, attempt to focus on the groupings of facilities that we're tossing out—animation, rotations, resources, and multimedia. Soon we'll start using them within a meaningful example.

Table 4.14 A view's audio- and video-related methods

Name	Description
`getCurrentTime()`	Returns the elapsed play time for the view's resource
`getPan()`	Returns the audio pan of the attached resource
`getTotalTime()`	Returns the total amount of time required to play the resource
`getVolume()`	Returns the volume of the attached resource
`play(frame, relative)`	Starts playing the attached resource
`seek(n)`	Skips forward or backward `n` seconds
`setPan(number)`	Sets the audio pan of the attached resource
`setPlay(boolean)`	Starts or stops playing the attached resource
`setResourceNumber(number)`	Selects the frame to be displayed for a multiframe resource
`setVolume(number)`	Sets the volume of the attached resource
`stop(frame, relative)`	Stops playing the attached resource
`unload()`	Unloads any media loaded with `setSource` or with `source=` attribute

4.1.5 Handling font specifications

Although fonts are just another type of resource, their settings are controlled through the view class. This may seem odd, since a view doesn't have a `text` attribute and can't directly display text. To display text, a view requires a subview that is capable of displaying text:

```
<canvas>
   <view fontsize="16" fontstyle="bold">
      <text text="Hello World"/>
   </view>
</canvas>
```

Table 4.15 describes the font-related attributes of views.

Table 4.15 A view's font-related attributes

Name	Data Type	Tag or Script	Attribute Type	Default	Description
`font`	`string`	Tag	Settable	`verity1`	Font to use for any text or `inputtext` elements appearing inside the view.
`fontsize`	`number`	Tag	Settable	`8`	Pixel size to use in rendering text appearing inside the view.
`fontstyle`	`string`	Tag	Settable	`plain`	Style to use in rendering text fields appearing inside the view; valid values are `plain`, `bold` , `italic`, or `bolditalic`.
`fgcolor`	`number`	Both	Settable	`black`	Color to use to render objects appearing inside the view.

Fonts were built into the view class to allow font attributes to be inherited by its subviews. This design decision provides a convenient way to assign font settings in a nested hierarchy. Subviews can override font attributes to make local changes.

4.1.6 Controlling the cursor

The mouse cursor icon is controlled through the `cursor` attribute and the `setCursor` method; see tables 4.15 and 4.17. This encapsulation allows the cursor to display any image particular to a view. Any supported graphic image—GIF, JPEG, PNG, or SWF—can be used as an image resource for the cursor. Listing 4.7 shows how a view—the first one in the listing—can specify a particular cursor image rather than the default image. The second view uses the default image.

Listing 4.7 Changing the mouse cursor

```
<canvas>
   <resource name="waitcursor"
             src="resources/lzwaitcursor_rsc.swf"/>
   <view width="100" height="100"
         bgcolor="#DDDDDD" cursor="waitcursor"/>
   <view x="130" width="100" height="100"
         bgcolor="#CCCCCC" clickable="true"/>
</canvas>
```

Although Laszlo supplies only the busy icon, others can be easily added. The results from listing 4.7 are shown in figure 4.9.

A meaningful cursor image is a useful technique for supplying the user with information in an unobtrusive way (although using tool tips is even better). Tables 4.16 and 4.17 show the cursor-related attributes and methods.

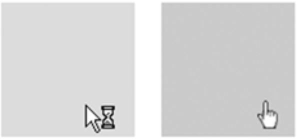


Figure 4.9 Two cursor images supplied with Laszlo are the busy icon and the hand icon, shown here for comparison.

Table 4.16 A view's cursor attributes

Name	Data Type	Tag or Script	Attribute Type	Description
`cursor`	`string`	Both	Settable	Sets the cursor icon to display when the cursor is over the view

Table 4.17 A view's cursor-related methods

Name	Description
`setCursor(resource)`	Sets the cursor to the given resource when the cursor is over the view

This section has presented some view basics: controlling visibility, attaching multimedia resources, specifying fonts, and controlling cursor behavior. Let's next examine different ways that users can interact with a view.

4.2 Interacting with a view

Since the view is the superclass for all visible objects, it must define the base attributes and methods for all user interaction. Later in chapter 6, which deals with components, we'll see how these base features are extended by user interface components to provide higher-level services such as default keyboard processing.

4.2.1 Receiving user events

User input in Laszlo was purposely designed to extend the familiar CSS mouse events that web programmers know from working with JavaScript. The complete list of mouse-related JavaScript events supported by the view object is given in table 4.18. Laszlo allows user input events to be handled inline within a view, rather than requiring an event handler. This provides some consistency with the naming scheme used on HTML-based websites. The following code shows a useful exercise in input event handling:

```
<canvas debug="true">
   <view x="10" y="10" width="100"
      height="100" bgcolor="green"
      onclick="Debug.write('onclick')"
      ondblclick="Debug.write('ondblclick')"
      onmousedown="Debug.write('onmousedown')"
      onmouseout="Debug.write('onmouseout')"
      onmouseover="Debug.write('onmouseover')"
      onmouseup="Debug.write('onmouseup')"/>
</canvas>
```

You are encouraged to run this example as a hands-on exercise. The goal is to get each debug message to display.

Table 4.18 A view's mouse-related events

Name	Description
onclick	Sent when the mouse button is clicked over a view.
ondblclick	Sent when the button is double-clicked over a view; a view needs to be registered to received this event; otherwise two click events are sent. A view's double-click time can be adjusted by setting its double_click_time attribute.
onmousedown	Sent when the mouse button is pressed over a view.
onmousedragin	Sent when the mouse moves over a view with the button down, after having been already pressed over this view.
onmousedragout	Sent when the mouse moves out of the view with the button down, after having been pressed while within the view.
onmouseout	Sent when the mouse moves out of a view with the button up.
onmouseover	Sent when the mouse moves over a view with the button up.
onmouseup	Sent when the button is released over a view.
onmouseupoutside	Sent when the button is released outside a view, after having been pressed over a view.

Practice with these events until you are comfortable with their operation. When dealing with mouse events, there is no substitute for the tactile experience.

Clickable and focusable settings

In order to interact with a view by selecting it with a mouse, it must be `clickable` or have a `clickregion`. As a convenience, specifying any mouse event handler for a view sets its `clickable` attribute. Resetting `clickable` to false overrides this default setting.

Setting the `focusable` attribute allows field navigation to be controlled by the keyboard; a view with focus accepts keyboard input. Similarly, specifying any `onfocus` event handler sets the `focusable` attribute for that view. The Tab key can be used to traverse through the focusable views. Focus can be *trapped* within a view or its child views using the `focustrap` attribute; this prevents other views from obtaining focus until the `focustrap` attribute is reset to false. Table 4.19 contains all of the user-input related attributes.

Table 4.19 A view's user input-related attributes

Name	Data Type	Tag or Script	Attribute Type	Description
`clickable`	`boolean`	Both	Settable	Must be set to true for the view to receive mouse events; automatically set to true if a mouse event script is specified in the tag.
`clickregion`	`string`	Both	Settable	Allows an irregular region to be established as the clickable area within a view.
`focusable`	`boolean`	Both	Settable	If true, the view participates in keyboard focus and receives focus events and keyboard events when it has the focus.
`focustrap`	`boolean`	Both	Settable	If true, the view traps the keyboard focus within itself or its children; useful to restrict keyboard focus to a specific area—for example, within a window or dialog.

The user-input-related methods, shown in table 4.20, manage mouse and keyboard interaction. The `getMouse` method returns the mouse coordinates relative to its view.

Table 4.21 summarizes a view's keyboard-related events. For the `onfocus` event, and consequently the `onblur` event to fire, a view must be clickable. For a view to lose focus, another view must be available to gain it, requiring the creation of another clickable view. Of course, both views must be `focusable`.

Table 4.20 A view's user-input-related methods

Name	Description
`getMouse('x' or 'y')`	Returns the x or y position of the mouse relative to this view; `x` or `y` must be specified.
`setClickable(boolean)`	Makes a view clickable or not clickable.

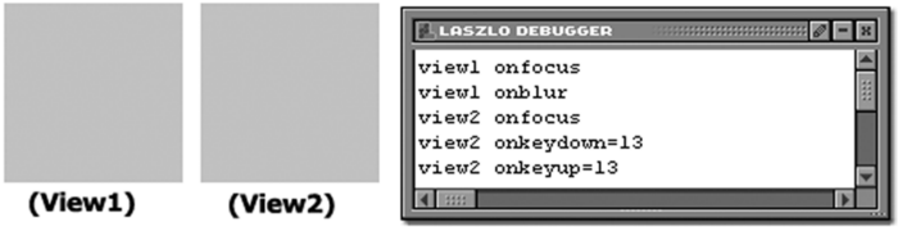
Table 4.21 A view's keyboard-related events

Name	Description
`onblur`	Sent immediately before a node becomes active, before the view displays, or before a layout affects its subviews.
`onfocus`	Sent when the view gets the focus; the parameter for the event is the new focus.
`onkeydown`	The `onkeydown` script is executed when the view has the focus and a key is pressed. Sent with the keycode for the pressed key.
`onkeyup`	The `onkeyup` script is executed when the view has the focus and a key is released. Sent with the keycode for the released key.

The `onblur` event occurs when a view loses focus to another view. The `onkeydown` and `onkeyup` events receive an event when a key is pressed down and later released. Listing 4.8 and its output in figure 4.10 demonstrate the effects of the `onfocus`, `onblur`, `onkeydown`, and `onkeyup` events.

Listing 4.8 The `clickable` and `focusable` attributes

```
<canvas debug="true">
   <view width="100" height="100" bgcolor="0xCCCCCCC"
         clickable="true" focusable="true"
         onblur="Debug.write('view1 onblur')"
         onfocus="Debug.write('view1 onfocus')">
      <handler name="onkeydown" args="key">
         Debug.write("view2 onkeydown=" + key);
      </handler>
      <handler name="onkeyup" args="key">
         Debug.write("view2 onkeyup=" + key);
      </handler>
   </view>
   <view x="120" width="100"
         height="100" bgcolor="0xCCCCCCC"
         clickable="true" focusable="true"
         onblur="Debug.write('view2 onblur')"
         onfocus="Debug.write('view2 onfocus')">
      <handler name="onkeydown" args="key">
         Debug.write("view2 onkeydown=" + key);
      </handler>
      <handler name="onkeyup" args="key">
         Debug.write("view2 onkeyup=" + key);
      </handler>
   </view>
</canvas>
```



```
view1 onfocus
view1 onblur
view2 onfocus
view2 onkeydown=13
view2 onkeyup=13
```

Figure 4.10 This example demonstrates the effect of the `onfocus` and `onblur` events along with the result of pressing the Enter key. The mouse is first clicked on the left view and then on the right view. Debugger output shows the first view acquiring focus and then losing focus to the second view, causing the first view to generate the `onblur` event. Finally, within the second view, the Enter key is pressed, causing the ASCII code for carriage return to be displayed in decimal as 13.

We once again encourage you to run this example. In particular, you might experiment with changing the `focusable` and `clickable` attribute settings and checking how the number of catchable events changes.

Figure 4.10 illustrates the result of clicking on a first view, and then on a second view. The debugger output shows the first view acquiring focus and then losing focus to the second view, which causes the `onblur` event. Finally, in the second view, the Enter key is pressed, causing the ASCII code for Carriage Return, 13 decimal, to be displayed.

4.3 Locating views

This section discusses the view methods that assist in locating your position within an application. In the first subsection, this involves locating an absolute or relative position within the screen coordinates. The second subsection deals with locating a view within the subviews hierarchy of an LZX application.

4.3.1 Locating absolute and relative screen position

You have already seen, with the jet formation exercise, how the x and y attributes of a nested view are offset from its parent's position. For those situations when a view needs to determine its absolute position or position relative to another reference frame, the getAttributeRelative method can be used to determine a position relative to another view. If the other view selected is the canvas, the position returned is absolute. This is demonstrated in listing 4.9, which shows three nested views, each offset by 20 pixels along the x- and y-axes. When it is necessary to place an object from one view context onto another, the getAttributeRelative method is useful for determining the x and y coordinates in another view's context.

Figure 4.11 illustrates how views are positioned based on relative offset values.

Listing 4.9 The getAttributeRelative method returning coordinates in another view

```
<canvas debug="true">
   <view name="first" x="20" y="20" width="200"
         height="200" bgcolor="0xCCCCCC">
      <handler name="oninit">
          Debug.write("first x=" + this.x);
          Debug.write("first y=" + this.y);
          Debug.write("first relative to third x=" +
             this.getAttributeRelative("x", first.second.third));
          Debug.write("first relative to third y=" +
             this.getAttributeRelative("y", first.second.third));
      </handler>
      <view name="second" x="20" y="20" width="100"
            height="100" bgcolor="0xDDDDDD">
         <view name="third" x="20" y="20" width="50"
               height="50" bgcolor="0xBBBBBB">
            <handler name="oninit">
               Debug.write("third x=" + this.x);
               Debug.write("third y=" + this.y);
               Debug.write("third relative to canvas x=" +
                  this.getAttributeRelative("x", canvas));
               Debug.write("third relative to canvas y=" +
                  this.getAttributeRelative("y", canvas));
            </handler>
         </view>
      </view>
   </view>
</canvas>
```

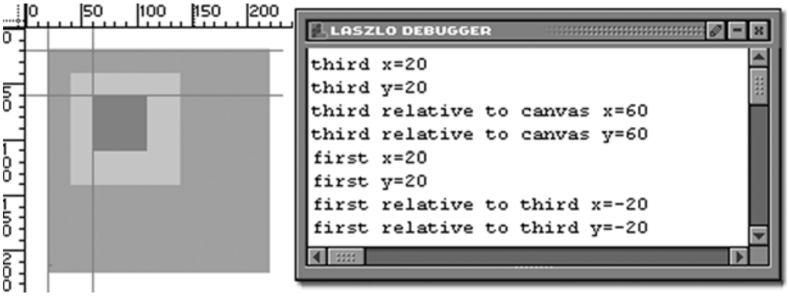
Offsets first by (20,20)

Offsets first by (20,20) from third

Offsets second by (20,20)

Offsets third by (20,20)

Displays third at (60,60) absolute



```
third x=20
third y=20
third relative to canvas x=60
third relative to canvas y=60
first x=20
first y=20
first relative to third x=-20
first relative to third y=-20
```

Figure 4.11 Here views are positioned with relative offset values. The `getAttributeRelative` method is used to determine a position relative to some other view.

Since each nested view is offset by 20 pixels, the x and y coordinates for each view display as 20 pixels. Displaying the third view's coordinates relative to the canvas shows the absolute value of 60 pixels. Displaying the first view's coordinates relative to the third view moves in a negative direction, to 20 pixels.

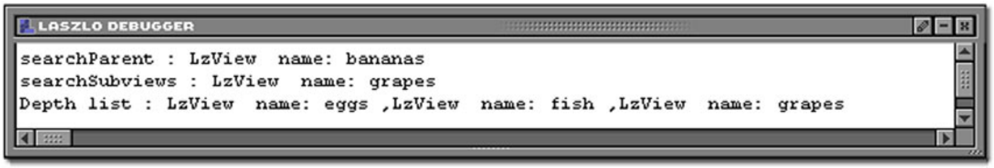
4.3.2 Locating a view

Every view, other than the canvas, has an entry in the subviews array of its parent. For situations in which a view needs to locate or to check for the existence of another view, Laszlo provides several ways to search through the parent-child hierarchy. The `searchSubviews` method searches down through the child views, searching for the specified attribute containing a matching value. The `searchParents` method searches up through the parent views. When a match is found, an `LzView` object is returned. While these methods allow any attribute to be searched for a matching value, searching for a particular named view is the most useful. In addition, the `getDepthList` method returns an array listing all sibling views. Listing 4.10 shows an example.

Listing 4.10 Searching for a view

```
<canvas debug="true">
  <view name="apples">
     <view name="bananas">
        <view name="carrots">
           <handler name="oninit">
              Debug.write("searchParent : " +
                 this.searchParents("name", "bananas"));
              Debug.write("searchSubviews : "  +
                  this.searchSubviews("name", "grapes"));
               Debug.write("Depth list : " +
                  this.getDepthList());
            </handler>
            <view name="eggs"/>
            <view name="fish">
                <view name="grapes"/>
            </view>
         </view>
      </view>
   </view>
</canvas>
```

Figure 4.12 shows the searchParents method returning a parent node bananas, while the searchSubviews method returns a child node grapes. The getDepthList method returns an array of sibling views in their declarative order, so the grapes subview isn't returned.



```
searchParent : LzView  name: bananas
searchSubviews : LzView  name: grapes
Depth list : LzView  name: eggs ,LzView  name: fish ,LzView  name: grapes
```

Figure 4.12 This debugger output shows two views, named bananas and eggs, found by searching with the searchParents and searchSubviews methods. The last line shows a listing of the subviews of the carrots view returned by the getDepthList method.

Table 4.22 summarizes the node search–related methods of the view class.

As you saw in the previous chapter, most objects can be instantiated with either a declarative tag or with JavaScript. We look next at dynamic instantiation of an LzView object.

Table 4.22 A view's node search–related methods

Name	Description
getDepthList()	Returns an array of subviews for the current level
searchParents(prop, val)	Searches parent views for a specified property
searchSubviews(prop, val)	Searches subviews for a specified property

4.4 Instantiating LFC-based objects

We'll complete our discussion of views by showing how to dynamically instantiate an LzView object. All declarative tags contained in the Laszlo Foundation Class (LFC) library are written in JavaScript. These declarative tags are easy to identify, because they are referenced within the debugger and the Laszlo reference document with a leading Lz prefix. When a JavaScript-based tag is dynamically instantiated, its JavaScript constructor is used. Every JavaScript constructor is supplied with a parent and a directory of associative name-value pairs for its initial set of attribute values. The JavaScript constructor for the LzView class looks like this:

```
var name = new LzView(parent, attributes)
```

where

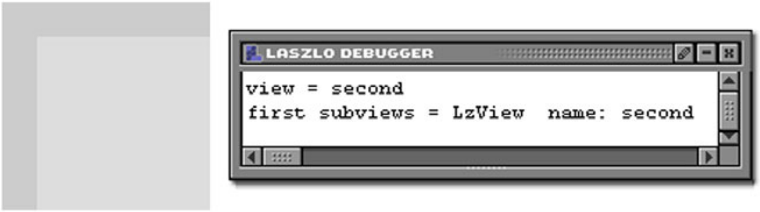
- name is the name of the new LzView object.
- parent is the parent node for the new LzView object. If the parent is set to null, then the new object is placed under the canvas.
- attributes is an array of attribute values to initialize the new object.

Listing 4.11 shows an example of dynamic instantiation.

Listing 4.11 Dynamic instantiation of JavaScript-based object

```
<canvas debug="true">
   <view name="first" bgcolor="0xCCCCCC">
      <handler name="oninit">
         var parentview =
             new LzView(first, {name: "second", width: 100,
             height: 100, bgcolor: 0xDDDDDD, x: 20, y: 20});
      </handler>
      <handler name="onaddsubview" args="view">
         Debug.write("view = " + view.name);
         Debug.write("first subviews = " + first.subviews);
      </handler>
   </view>
</canvas>
```

Creation of a new view generates an onaddsubview event. The parent view is resized to accommodate the dimensions of its child. In figure 4.13, the newly created subview second can be seen in its parent's subviews array.



```
view = second
first subviews = LzView  name: second
```

Figure 4.13 Adding a new entry to the canvas's subviews array generates an `onaddsubview` event.

The goal of these preceding sections was to scale the mountain known as the `LzView` API. Since the `LzView` object is such a key part of LZX, it's critically important to have a strong command of this API in particular.

Now that you have a basic understanding of views, it's time to begin customizing them with user-defined classes and class-based inheritance.

4.5 User-defined classes

LZX tags use a class inheritance model similar to that of Java. Both are class-based languages that maintain a distinction between classes and instances. Although JavaScript's inheritance is prototype based, its main purpose is to add a dynamic quality to the tag inheritance model. When writing a Laszlo application, you'll have almost no need to create new JavaScript classes. Rather, you'll do most development with LZX classes.

LZX classes implement both *inheritance*, where additional details are added to a general data type to create more specific data types, and *object composition*, where simple objects are combined to produce more complex ones. This means that it supports both "is-a" and "has-a" relationships. So for instance, a `Ford` *is a* specialized instance of a `car` class, while a `car` class *has an* `engine` class.

We'll further examine inheritance-related features, such as overriding methods to change behavior and finding information about an ancestor class. LZX also provides a class root shortcut for accessing nodes within a class. Depending on its placement, this class root can be used to either access the root node within a class or to traverse up the hierarchy tree to access a superclass. Finally, we'll discuss how to create new JavaScript classes for those times when it's needed.

4.5.1 Overriding a method in a subclass

Although LZX doesn't support overloading a method, a subclass can override inherited methods to change their behavior. This is accomplished by creating a new method with an identical name to the superclass's method. Although it is straightforward to override user-defined methods, things get more complicated when overriding system-defined methods.

Listing 4.12 shows how the box class's setWidth method is overridden to set its opacity proportional to its width, so that a larger width results in a lighter box.

Listing 4.12 Overriding a method

```
<canvas>
   <class name="box" width="100" height="100" bgcolor="#BBBBBB"/>
   <class name="FadedBox" extends="box">
      <method name="setWidth" args="w">
         setAttribute("opacity", 1 - w/canvas.width);
      </method>
   </class>
   <FadedBox width="150"/>
</canvas>
```

But there are several problems with this approach; the width of the view never gets set because the original behavior of setWidth never gets executed. The display ends up empty, since a view needs both its width and height attributes to be greater than zero. But the setAttribute method can't be used to set the width:

```
setAttribute("width", w);
```

This would cause an endless loop, because setAttribute calls the setWidth method. Instead, it is necessary to set the width and then perform all of Laszlo's system responsibilities. Rather than to attempt this, a better solution is to call the superclass's overridden method to perform this default behavior. The super keyword is used to call a method in the superclass:

```
<canvas>
   <class name="Box" width="100" height="100" bgcolor="#BBBBBB"/>
   <class name="FadedBox" extends="Box">
      <method name="setWidth" args="w">
         setAttribute("opacity", 1 - w/canvas.width);
         super.setWidth(w);
      </method>
   </class>
   <FadedBox width="150"/>
</canvas>
```

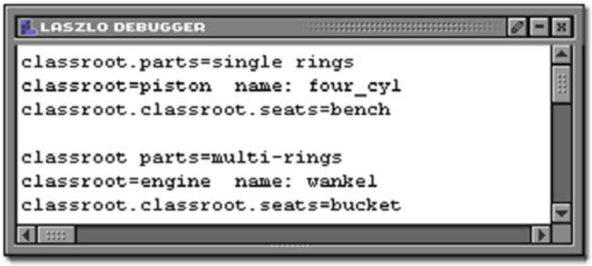
Now when the `width` attribute is set to 150 pixels in the `FadedBox` instance, our `setWidth` method is called first, setting `opacity` to 70 percent and calling the superclass's `setWidth` method to ensure that the width attribute is correctly set.

Now that you've seen how to use LZX's class-based inheritance to override a superclass's methods, let's take a look at using its composition features with the `classroot` qualifier.

4.5.2 Using the classroot qualifier with classes

Although the `parent` qualifier is used to reference nodes in the XML node hierarchy, a `classroot` qualifier is also supplied to reference parent superclasses within the *class composition hierarchy*. The `classroot` qualifier can only be used within a class and, depending on the `classroot` qualifier's placement level, it operates in two slightly different ways. When used within a nested node in a class definition, it references that class definition. This allows a deeply nested object to easily reference top-level attributes without having to use a long string of `parent. … .parent` qualifiers. When used at the top level of a class definition, it references the parent class in the class composition hierarchy. When multiple `classroot` qualifiers are chained together, either at the top level or within a nested view, they traverse an equal number of levels up through the class composition hierarchy.

Listing 4.13 illustrates these class relationships; its output is shown in figure 4.14. The listing shows an example of class composition; a car has an engine, which has a piston. There is a single instance of a car, which calls the two `fire` methods of its piston instance; one at the top level and the other within nested views, causing their `classroot` references to be different. The code contained in each `fire` method is identical; the only difference is their placement.



```
classroot.parts=single rings
classroot=piston  name: four_cyl
classroot.classroot.seats=bench

classroot parts=multi-rings
classroot=engine  name: wankel
classroot.classroot.seats=bucket
```

Figure 4.14 A `classroot` reference works differently depending on its placement. A top-level `classroot` references its parent class in the class composition hierarchy. A nested `classroot` is a shortcut to the root of its class definition. No matter their placement, chained `classroot` specifiers always reference an equal number of parent steps upward in the class composition hierarchy.

The top-level classroot references attributes from the engine class, while the nested classroot references attributes from its current piston class. This corresponds to the composition structure:

- car (has a)
- engine (has a)
- piston

A top-level classroot references attributes one class level higher than a classroot specified in a nested node. As seen in the debugged output of figure 4.14, the top level needs a classroot.classroot qualifier to reference the seats attribute in the engine class, while a nested node needs a classroot.classroot.classroot qualifier to get there.

Within the nested view, we also see that a classroot is equivalent to the longer parent.parent qualifier for accessing attributes located at the root level of the class definition.

Listing 4.13 Accessing the root node of a class instance using classroot

```
<canvas debug="true">
   <class name="car">
      <attribute name="seats" value="bucket" type="string"/>
      <engine name="wankel"/>
      <handler name="oninit">
         wankel.four_cyl.a.b.fire();
         wankel.four_cyl.fire();
      </handler>
   </class>

   <class name="engine">
      <attribute name="rings" value="multi-rings" type="string"/>
      <attribute name="seats" value="bench" type="string"/>
      <piston name="four_cyl"/>
   </class>

   <class name="piston">
      <attribute name="rings" value="single rings" type="string"/>
      <view name="a">
         <view name="b">
            <method name="fire">
               Debug.write("classroot=" + classroot);
               Debug.write("parent.parent=" + parent.parent.rings);
               Debug.write("classroot.rings=" + classroot.rings);
               Debug.write("classroot.classroot.classroot.seats=" +
                         classroot.classroot.classroot.seats + "\n");
            </method>
```

Composes car with engine

Composes engine with piston

Accesses class root

```
         </view>
      </view>
      <method name="fire">
         Debug.write("classroot=" + classroot);
         Debug.write("classroot.rings=" + classroot.rings);
         Debug.write("classroot.classroot.seats=" +
                      classroot.classroot.seats + "\n");
      </method>
   </class>
   <car name="mazda"/>
</canvas>
```

Accesses calling class

Now that you have seen how both inheritance and composition are implemented in LZX, it's time to wrap things up by taking a look at dynamic class instantiation.

4.5.3 Instantiating LZX-based objects

The only thing left is to see how to dynamically instantiate LZX-based class objects within JavaScript. The full syntax to instantiate a user-defined class is a bit unwieldy, because it supports creating a tree of child nodes. However, since the last two arguments are optional, we discourage their use. This results in the identical syntax as for JavaScript-based instantiation. LZX-based class instantiation looks like this:

```
var name = new myclass( parent, attributes,[ children, instcall ])
```

where

- `name` is the name of the new object.
- `myclass` is the class of the new object.
- `parent` is the object that is to contain the new object; i.e., where the new object is to be placed in the object hierarchy. The new object's superclass is, of course, the class of its parent object. If the parent is set to `null`, the newly created object is a subclass of the view and exists as a top-level object under the canvas.
- `attributes` is an array of attribute values for initializing the new object.
- `children` is an array of child views to be encapsulated by the new object.
- `instcall` is a boolean that determines whether the new object is immediately instantiated or instantiated normally with the other views.

The following example reuses the previous example shown in figure 4.14 to illustrate how to dynamically instantiate an instance of `new_class`.

```
<canvas debug="true">
   <class name="new_class" width="100" height="100"/>

   <view name="first" bgcolor="0xCCCCCC">
      <handler name="onaddsubview" args="view">
         Debug.write("view = " + view.name);
         Debug.write("first.subviews = " + first.subviews);
      </handler>
   </view>

   <method name="init">
      var parentview = new new_class(first,
            {name: "second", bgcolor: 0xDDDDDD,
             x: 20, y: 20, width: 100, height: 100});
   </method>
</canvas>
```

This output matches the results from figure 4.14.

This completes all the functionality associated with the `LzView` object. The `LzView` object is a fundamental part of Laszlo, so allow some time to work through the examples and to digest all the material presented here. In the upcoming chapters, we'll revisit many of these features, so if you encountered any difficulties, they will hopefully clear up when viewed within an application context.

4.6 Summary

The `LzView` object is a microcosm for much of the functionality in LZX. Since it's the superclass of all visible objects, familiarity with it carries across to these derived objects. For example, it defines all the mechanisms for controlling visibility. Visibility isn't defined by just a single attribute; a large set controls visibility in different ways. Animation can be used with the visibility attributes to control visibility gradually rather than in abrupt steps.

An `LzView` object controls all access to attached multimedia resources: images, audio, video, the cursor, and fonts. This releases a view from a restrictive rectangular shape allowing it to take on the irregular shape of an attached image. An `LzView` object provides various ways to stretch and shrink images to fit into confined areas.

A full set of controls is provided for managing the playback of both audio and video media. An application has all the functionality found in a hardware audio or video player: fast forward, reverse, volume control, and stereo panning. These multimedia resources can be downloaded in their entirety or streamed from a networked website. Attributes allow the current progress of a downloaded medium to be monitored in either bytes or frames.

Fonts are managed within the `LzView` object, making use of both resources and the parent-child hierarchy to allow a font specification to cascade through subviews. This provides a hierarchy of fonts, enabling subviews to inherit font settings.

The `LzView` object defines the basic interactions of a user with a visible object, including the entire mouse- and keyboard-based set of events. It also defines the `clickable` and `focusable` attributes, which manage the views that will receive events.

The goal of this ambitious chapter was to show how the fundamental elements of LZX fit together and to allow the big picture to emerge. Now that this foundation has been established, we can begin working with an online store application, called the Laszlo Market, to demonstrate how to build an RIA application. We hope that you'll find it easier to understand many of Laszlo's concepts in an application context.

5

Designing the Laszlo Market

This chapter covers

- Prototyping an application
- Designing screen layouts and transitions
- Refactoring code
- Testing

Art is making something out of nothing and selling it.

—Frank Zappa,
composer, guitarist, and singer

You now have the requisite Laszlo LZX skills to embark on the development of our online Laszlo Market store, which sells action videos. But rather than jumping in and beginning to code, let's take some time to discuss development strategy.

Building an RIA requires a wide range of skills, including design, client-side and server-side programming, and content development. The required skills are too diverse to be covered by a single individual. As a result, most RIAs are built by a team with widely varying backgrounds. As with any team project, good communication and coordination are critical. There are many different approaches to resolving the communication issue. One is to write requirements and design specification documents as a basis for coordinating activities.

A complementary approach is prototype development, whereby the team builds a series of continually refined prototypes to clarify both the requirements and the design. In effect, the prototype becomes a living specification. This approach fits well with *agile development*, where the goal is to satisfy the customer through continuous delivery of operational software.

This chapter demonstrates how to apply prototyping to LZX development for the Laszlo Market.

5.1 Prototyping our application

A strong dichotomy exists between the worlds of designers and developers. These differences are large enough to have resulted in separate working methodologies. Developers generally prefer a *system-centered* approach, using modeling tools such as Unified Modeling Language (UML), and use case statements to capture requirements. On the other hand, designers prefer a *user-centered* approach, relying on users to express their requirements.

One reason for these differences is they need to accomplish different goals. Designers are concerned with identifying *what* needs to be developed, while developers are concerned with *how* to implement it. The user-centered design (UCD) approach is based on the golden rule of "Know thy user." This requires a collaborative design process between designers and customers to ensure that designers understand customer needs. Conversely, a system-centered approach complements UCD by providing a structural model. Combining these two approaches produces two models: a sequence of prototypes for the client side and a traditional series of UML models for the server side. Figure 5.1 illustrates the UCD methodology top-down movement from wireframe sketches of screen layouts, storyboards of screen transitions, and prototyping.

We'll start our wireframes with rough wireframe pencil sketches to illustrate the placement of information on the screen.

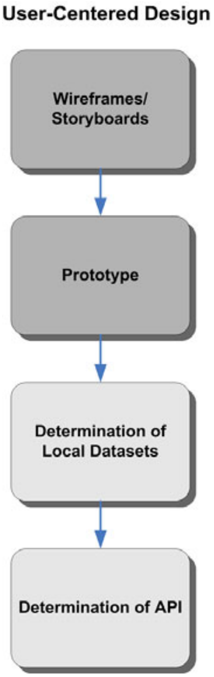


Figure 5.1 The user-centered design for our graphical user interface proceeds top down beginning with wireframe sketches and moving through storyboards and prototyping. This chapter covers the first three stages, up to prototyping.

5.1.1 Creating wireframes

The process of creating wireframes has few rules. The main idea is to get thoughts down on paper without worrying about pretty or elaborate diagrams. If a wireframe is too detailed or complicated to be easily jotted down on paper, the basic ideas probably need to be simplified. Remember that a wireframe shows only what needs to be accomplished, not how it is to be done.

Start with a list of the required functions for your application. Laszlo Market's main screen must display the following functional areas:

- Browse Search
- Product List
- Product Details
- Shopping Cart
- Media Player
- Checkout

Figure 5.2 shows our first wireframe sketch for the main screen. The goal of this design is to make all relevant functions visible.

Now we can begin answering the *what* questions. What is the function for each window? These scenarios are also known as an application's business processes and are directly related to the *use cases*. It's often easier to conceptualize business processes using visual aids rather than text. Wireframes frequently provide a simpler method for generating use case statements.

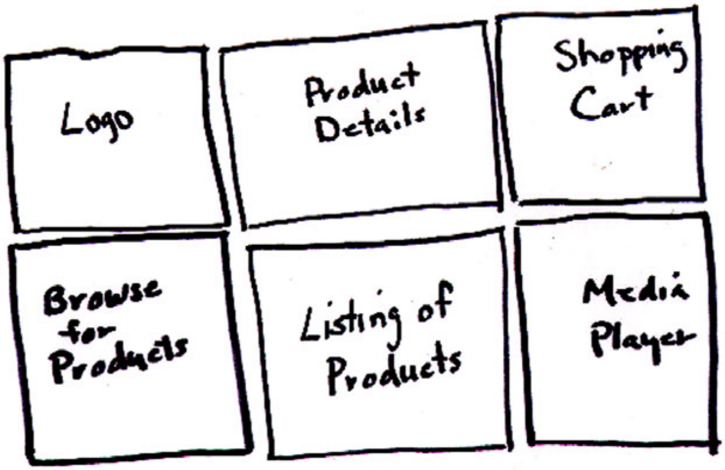


Figure 5.2 The initial wireframe for the Laszlo Market's main screen divides it into five functional areas, along with a window for the store logo.

The question that needs to be answered is "How does a user purchase an item?" In other words, what steps need to be performed to complete a transaction? We'll need to collect the user's shipping and billing information, along with the purchased items, and then summarize all this information into a purchase confirmation. Let's assume that each of these activities requires a separate form. This produces three forms needing to be completed for a purchase: shipping, billing, and order confirmation. The next question is "Where will these forms be displayed?"

At this point we're not sure, but we'll assume there will be a Checkout window to contain these forms or pages. We'll capture these requirements in a series of wireframes, shown in figure 5.3.

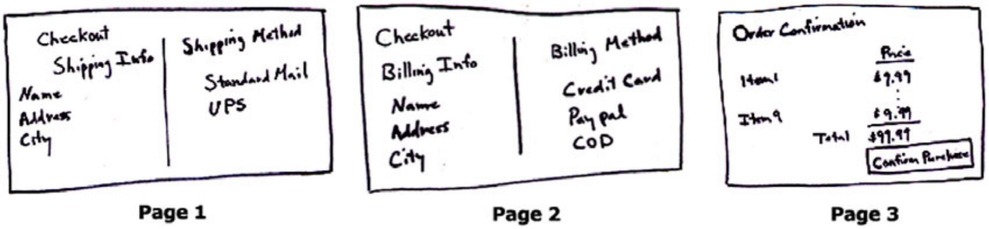


Figure 5.3 This wireframe for the Checkout window shows the checkout process for a user to purchase an item. It is assumed that three separate forms are to be completed for a purchase: a shipping form, a billing form, and an order confirmation.

Although the three pages can't be displayed simultaneously, they need to be easily accessible. There are several paradigms available, such as tabs or tabsliders, for displaying a number of pages in a window. For the moment, we'll just wireframe the Checkout window as a stack of pages, as shown in figure 5.4.

Let's pause and summarize the proposed windows in our design; we have the six windows on the main screen, shown in figure 5.2, along with the Checkout window with its three pages, shown in figure 5.4. We can now design the screen transitions for maintaining visual continuity.

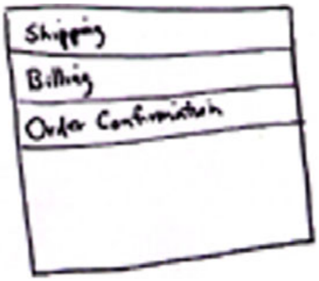


Figure 5.4 The Checkout window is represented as a low-fidelity wireframe with three checkout pages stacked inside it.

At this point, we don't know if it will be handled within a single physical screen or as an *exterior* screen requiring a transition. However, the first transition is from the main screen to a screen containing the Checkout window. Transitions are designed using storyboards, so let's look at that technique next.

5.1.2 Storyboard transitions

Storyboarding is a technique borrowed from the animation industry to display the states of a visual sequence. We'll start with the state transitions shown in figure 5.5. Even though this initial sequence is trivial, creating a storyboard is still worthwhile, since transitions can later grow more complex.

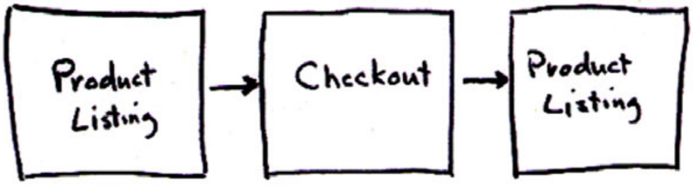


Figure 5.5 An initial storyboard sequence for the main screen and the checkout screen shows transitions from main to checkout and back to main.

We're now ready to move to a more detailed design. We'll start by making an initial guess at the physical dimensions of the windows that appear on the screen. We can get a rough idea of our model's workability by examining the wireframe displayed in figure 5.6.

There is no substitute for seeing a live display of our design. Armed with our wireframes, we are ready to build a prototype. This is useful for evaluating proportions, and other issues such as color and font. Prototyping must be fast and adaptable to allow different approaches and solutions to be investigated quickly. LZX contains a number of features that makes it an excellent tool for rapid prototyping.

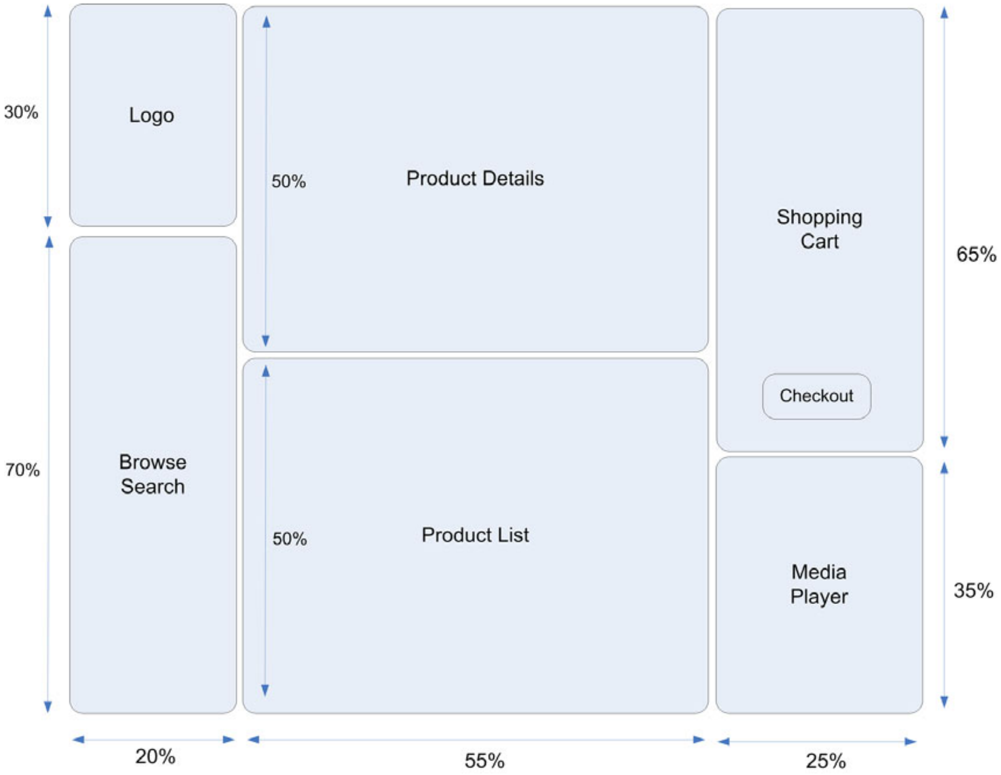


Figure 5.6 The first try at a detailed wireframe for the main screen of the Laszlo Market, mainly to investigate proportions, contains six windows: a Logo header, Browse Search, Product List, Product Details, a Shopping Cart, and Media Player.

5.2 Coding the prototype

LZX's built-in layouts and constraints facilitate a straightforward information transfer from wireframes into code. We can almost code directly from the wireframe measurements in figure 5.6. The canvas defaults to a width and a height setting of 100 percent, allowing the application to resize when the size of the browser window changes. We'll create a container view named `main` to hold the storefront modules, represented as views, to fill the available space:

```
<canvas>
   <view name="main" width="100%" height="100%">
      <!—- Storefront modules -->
   </view>
</canvas>
```

So far, we have an empty canvas with a nonvisible view container in it. So let's add a labeled header module. The header module uses percentage values for its width and height. The `align` and `valign` attributes position the label within this view. The `valign` attribute sets the vertical alignment to `top`, `middle`, or `bottom`, while the `align` attribute sets the horizontal alignment to `left`, `center`, or `right`. Since the label needs to be centered, it is set to `center` and `middle`:

```
<canvas width="100%" height="100%" bgcolor="0xDDDDDD">
   <view name="header" width="20%" height="30%"
         bgcolor="0xCCCCCC">
      <attribute name="label" type="string" value="Logo"/>
      <text align="center" valign="middle"
            fontsize="12" fontstyle="bold"
            text="${parent.label}"/>
   </view>
</canvas>
```

Because we have many views and each needs to be labeled, we'll define a class for them. To get a quick start on creating this class, let's take one of the views and tweak it with some additional tags. After it has the desired appearance, we only need to change the `text` attribute constraint setting from `parent` to `classroot` to convert it into the `lview` class shown in listing 5.1. Our labeled views are ready to be instantiated. Welcome to your first example of *instance-first* development.

Listing 5.1 Using a class to add labels to the display

```
<canvas width="100%" height="100%" bgcolor="0xDDDDDD">
   <class name="lview">
      <attribute name="label" value="default" type="string"/>
      <text align="center" valign="middle"
            fontsize="12" fontstyle="bold"
            text="${classroot.label}"/>
   </class>

   <view name="main" width="100%" height="100%">
      <lview name="header" label="Logo"
             width="20%" height="30%"/>
      <lview name="details" label="Product Details"
             bgcolor="0xCCCCCC"
             x="${parent.browse.width}"
             width="55%" height="50%"/>
      <lview label="Shopping Cart"
             bgcolor="0xCCCFFF"
             x="75%"
             width="25%" height="65%"/>
      <lview name="browse" label="Browse Search"
             bgcolor="0xBBBFFF"
             y="${parent.header.height}"
```

Defines labeled view class

Contains header Logo view

Contains Product Details view

Contains Shopping Cart view

Contains Browse Search view

```
              width="20%" height="70%"/>
       <lview label="Product List"            Contains Product List view
              bgcolor="0xDDDFFF"
              x="${parent.browse.width}"
              y="${parent.details.height}"
              width="55%" height="50%"/>
       <lview label="Media Player"            Contains Media Player view
              bgcolor="0xBBBBBB"
              x="75%" y="65%"
              width="25%" height="35%"/>
    </view>
</canvas>
```

Let's move clockwise starting with the `header` view. This view contains no `x` or `y` attributes, so it's placed at the default (0, 0) position at the top-left corner of the canvas. The horizontally adjacent view is the Product Details view. To account for the `header` view, its `x` attribute is offset by the value of the header's `width` attribute. The vertically adjacent view is the Browse Search view. To account for the `header` view, its `y` attribute is offset to the value of the header's `height` attribute. Afterward, these views are used as the offsets for the remaining view. The test output seen in figure 5.7 looks fine and verifies our layout.

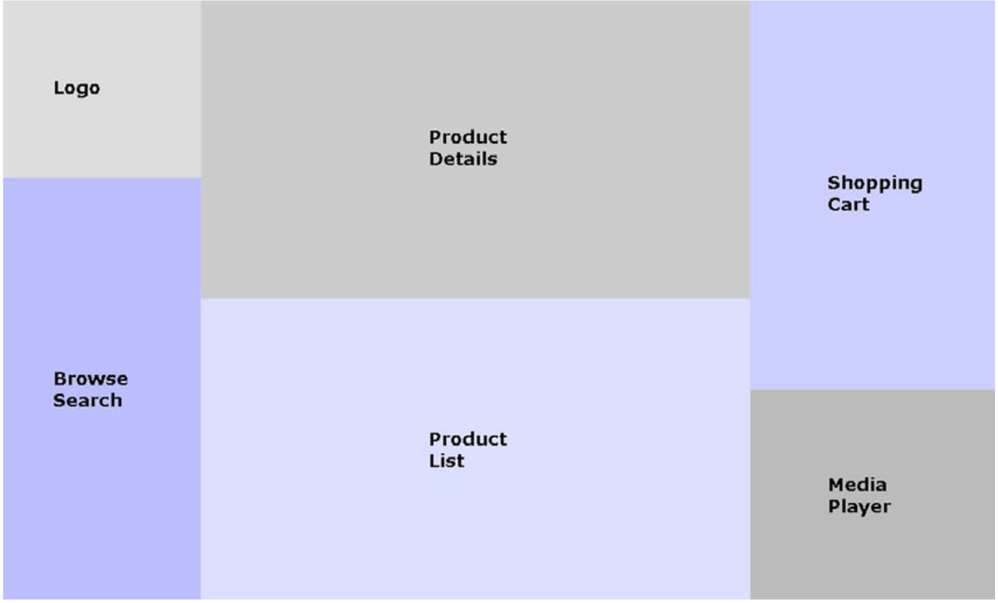


Figure 5.7 The views are labeled and colored to help identify them.

Labeled views are useful when the screen needs to be divided into abstract areas. The next step requires a more interactive interface. The window component has many properties that make it the ideal prototyping tool for such a job.

5.2.1 The window as a prototyping tool

The window component is self-labeling, resizable, and movable, and it conforms to the Laszlo component interface, allowing it to easily interact with other Laszlo components and letting you add sophisticated features such as scrollbars and grids later. Listing 5.2 shows how easily our code is updated to use windows. At this stage of design, we're primarily focused on layout issues and aren't concerned with visual design issues. Nevertheless, it's useful to add default artwork to check for sizing issues.

Art assets can be compiled or dynamically loaded into an application. We'll add a logo and placeholder image for the Media Player. Since we're just roughing things out, we'll use a `stretches` attribute to stretch the image to fit the Media Player window; although this results in some image degradation, it suffices for now. In chapter 10 we'll show you how to resize images so they don't become distorted.

Listing 5.2 Creating the Laszlo Market prototype with windows

```
<canvas bgcolor="0xDDDDDD">
   <resource name="logo"
             src="resources/laszlo_store_header.png"/>
   <resource name="video" src="resources/thematrix.jpg"/>

   <view name="main" width="100%" height="100%">
      <view name="header" width="20%" height="30%"
            resource="logo" x="30" y="15"/>
      <window name="details" title="Product Details"
            x="${main.header.width}"
            width="55%" height="50%" resizable="true"/>
      <window name="shopcart" title="Shopping Cart"
            x="75%"
            width="25%" height="65%" resizable="true"/>
      <window name="browse" title="Browse Search"
            y="${main.header.height}"
            width="20%" height="70%" resizable="true"/>
      <window name="productlist" title="Product List"
            x="${main.browse.width}"
            y="${main.details.height}"
            width="55%" height="50%" resizable="true"/>
            <window name="mediaplayer" title="Media Player"
                  x="${main.browse.width+main.productlist.width}"
                  y="${main.shopcart.height}"
                  width="25%" height="35%" resizable="true">
               <view stretches="both"
                     width="100%" height="100%" resource="video"/>
            </window>
         </view>
      </canvas>
```

The main view is now populated with windows, as shown in figure 5.8. Since windows can be dragged and resized with the mouse, we can manually tweak them if we don't like the initial layout.

So far, the construction of the prototype has been as easy as programming in HTML, and the resulting LZX program has fewer than 20 lines of code. Nevertheless, this is an appropriate stage to think about partitioning our code base into libraries.

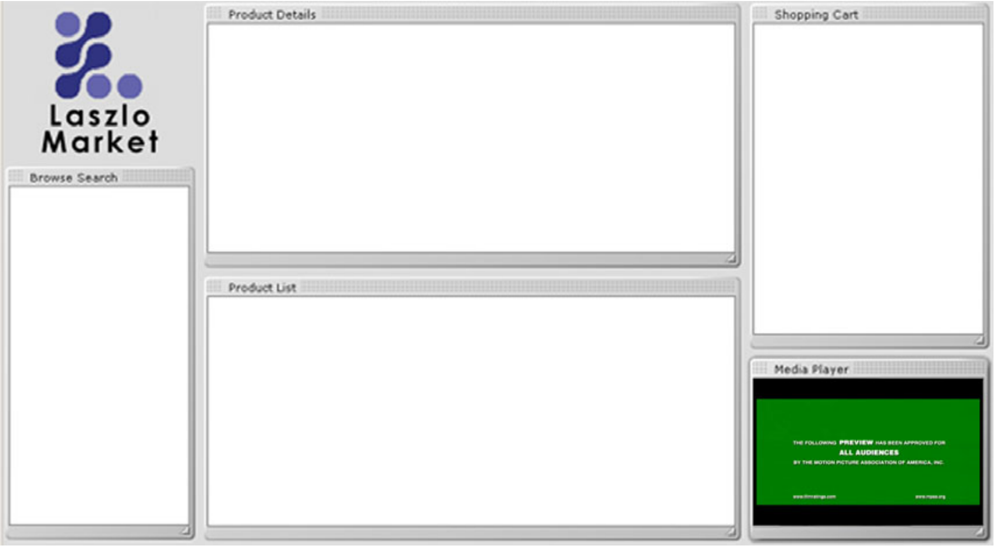


Figure 5.8 The main screen layout for the Laszlo Market prototype, with image resources added, starts to take form. The windows can be adjusted by dragging and resizing with the mouse. Afterward, a pixel ruler can be used to measure any adjusted values.

5.2.2 Organizing with libraries

Every application of significant size should be partitioned into library files. The `include` tag contains an `href` attribute to include a file. Included files can be library, text, XML, or directory files. Included files can be nested to build hierarchies of included files. Laszlo supports the following types of include files:

- Library files
- Text files
- XML files
- Directories

A library file is an XML file whose root element is a `library`. The contents of a library are included only once; this is true even if the library file is included more than once. All declarative tags specified in a library are considered by Laszlo to be top-level tags. This makes a library a good place to declare resources.

When the root element of an included file is an XML tag, other than a `library` tag, the contained XML code segment is inserted at the point of the `include` tag. This occurs for each instance encountered in the source code. Similarly, the contents of a text file are included at the spot of an `include` tag. It is identified by a `type` attribute that is set to a value of `text`. It is also included for each instance of the tag.

Finally, a file system pathname can be used to reference a file; e.g., `pathname/library.lzx`. The examples in listing 5.3 illustrate the various `include` forms.

Listing 5.3 Four forms of the include tag: a library, text, XML, and directory file

```
<canvas>
   <include href="resources.lzx"/>     <— Include library file

   <text>
      <include href="text.txt" type="text"/>     <— Include text file
   </text>

   <window>
      <include href="code.xml"/>     <— Include XML file
   </window>

   <include href="resources"/>     <— Include directory
</canvas>
```

We'll use two library files within our Laszlo Market application: library.lzx, a general-purpose library file; and resources.lzx, a list of media resources. To assist these libraries with their organizational duties, we'll also create standardized directories to store files. All media resources are stored in the resources directory and listed in resources.lzx:

```
<library>
   <resource name="logo"
             src="resources/laszlo_store_header.png"/>
   <resource name="video" src="resources/thematrix.jpg"/>
</library>
```

LZX source files are stored in the lzx directory and are listed within our library.lzx file.

We're ready now to move on to the checkout process. So we need to pull back out our wireframe sketches.

5.3 Configuring the checkout screen

The design goal for our application is to maintain visual continuity throughout the shopping experience, but this requires more information than can fit within a single screen. What's worse, the checkout process involves information that spreads across the three pages of the Checkout window—shipping information, billing information, and order confirmation. Supporting visual continuity requires that we address these problems:

- Handling the transitions between the main and checkout screens
- Displaying the checkout-related shopping information

5.3.1 Sliding a virtual screen

First imagine one large virtual screen containing all the windows. Then visualize the browser window sliding across this background of windows. By keeping some windows from vanishing, we ensure that a user retains situational awareness and won't get lost. This is a useful design tool for configuring an application, since it provides a way to lay out a flat listing of windows. This allows us to identify common elements and helps us find a design that minimizes the size of this listing. Afterwards, areas containing similar functionality can be identified and placed within a hierarchy of windows and sub-windows.

Because our window configuration fits within two screens, it is an ideal candidate for a single virtual screen. With only two states of the browser window, most users won't perceive a large virtual screen but will see the application as having

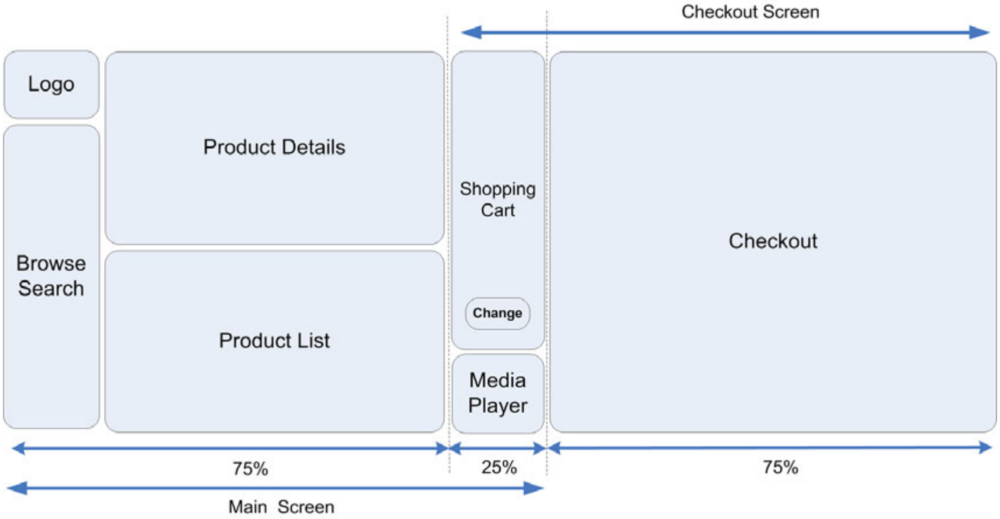


Figure 5.9 This wireframe demonstrates a proposal for transitioning from the main screen to the checkout screen while maintaining visual continuity. The Change button scrolls the display horizontally to expose either the main screen or the checkout screen.

two modes. To make this convincing, let's use the Shopping Cart and Media Player windows as the common unifying elements across the main and checkout screens, as shown in figure 5.9.

The Shopping Cart contains the most area, so it's a natural spot for a Change button to switch between screen configurations. For now, don't worry about how this is accomplished; just be aware that the Shopping Cart and Media Player windows must also be included in the checkout screen's wireframe.

5.3.2 Stacking pages

The next problem we'll address is the display of the checkout pages. Laszlo supplies numerous components to manage the display of multiple-element screens. A *tabslider* stacks pages represented as *tabelement* elements within the tabslider. This works well for our checkout pages.

Using a tabslider

A "high-fidelity" wireframe for the checkout screen is shown in figure 5.10. The Shopping Cart window on the left displays the list of items to be purchased. The Checkout window tabslider supports three tabelements for the three checkout pages. The Media Player provides customers with one last opportunity to view their electronic media.

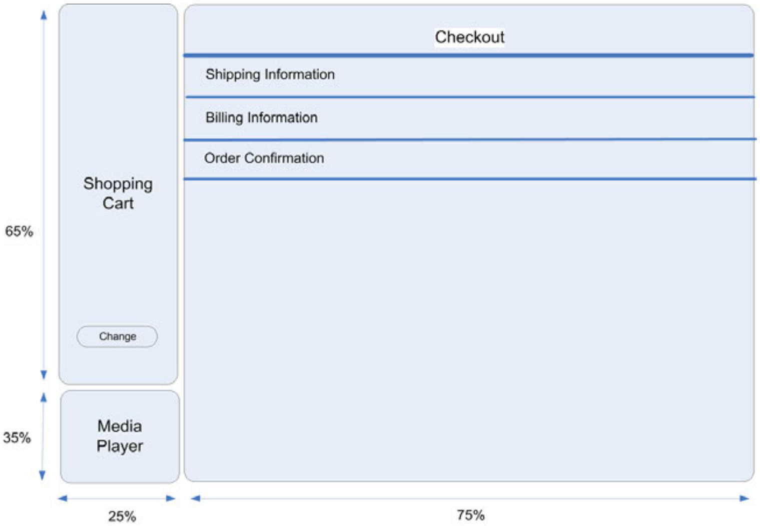


Figure 5.10 This high-fidelity wireframe shows the checkout screen. The Checkout window comprises three tabbed pages. The Shopping Cart and Media Player windows are shared with the main screen. The Change button triggers a transition to the main screen.

The LZX code that implements the wireframe of figure 5.10 is no more complex or lengthy than that for our initial main screen. Once again, we can plug many of the values directly from the high-fidelity wireframe into the `height` and `width` attributes of the windows. Figure 5.11 shows the results.

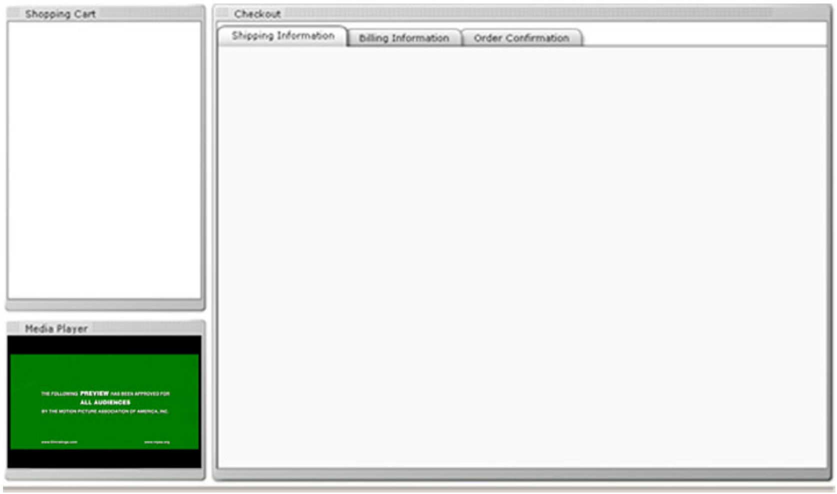


Figure 5.11 The screen display for the checkout process. The Checkout window consists of a tabslider with three tab elements for shipping information, billing information, and order confirmation. The Change button reconfigures the display back to the main screen.

At this point, we'll create the checkout screen as a separate Laszlo application. Listing 5.4 shows the code corresponding to the display shown in figure 5.11. We'll worry later about having to integrate these applications together again.

Listing 5.4 LZX code to create the checkout screen in figure 5.11

```
<canvas width="100%" height="100%">
   <include href="resources.lzx"/>

   <view name="main" width="100%" height="100%">
      <window name="shoppingcart" title="Shopping Cart"
              width="25%" height="65%">
         <button id="checkoutbtn" text="Change"
                 y="80%" align="center"/>
      </window>
      <window title="Media Player"
              y="${main.shoppingcart.height}"
              width="25%" height="35%">
         <view width="100%" height="100%"
               resource="video" stretches="both"/>
      </window>
      <window title="Checkout"
              x="${parent.shoppingcart.width}"
              width="75%" height="100%">
         <tabslider name="checkoutsteps"
                    width="100%" height="100%"
                    spacing="2" slideduration="300">
            <tabelement text="Shipping Information"/>
            <tabelement text="Billing Information"/>
            <tabelement text="Order Confirmation"/>
         </tabslider>
      </window>
   </view>
</canvas>
```

❶ Display a tabslider

❷ Set spacing

❸ Display three tabelements

The tabslider ❶ contains two attributes ❷, `spacing` and `slideduration`, for controlling its appearance. The `spacing` attribute specifies the width of the gap between each closed tabelement. The `slideduration` attribute specifies the amount of time in milliseconds for a particular tab entry to slide open. The three pages within the Checkout window are specified with individual tabelements ❸.

After we show this prototype to some users, receive feedback and, more importantly, approval on the overall design, we can begin planning how to integrate the two screen prototypes and handle their transition.

TIP *Debugging: What to do when components or views don't display* One of the most perplexing problems for new Laszlo developers is how to handle visual objects that won't display. The first step is to add an id value to the declarative tag and then turn on the debugger. This allows the object to be easily referenced from the debugger input window. Ensure that the object is in expanded format by double-clicking on it. Every view-based object's display is governed by the same critical attributes: height, width, x, y, and visibility. Check these values in the debugger. Ensure that a bgcolor or resource is associated with this object and that this view-based object isn't hiding behind another object. You might want to return to chapter 4 to review the properties controlling a view's visibility.

If the values appear to be valid, experiment by setting different values in the debugger. Remember that direct assignments don't trigger constraints or event handlers and don't update the display. To generate a visible change, the attribute value must be changed with setAttribute. For example:

```
XX.x=30                  // Does change display characteristics
XX.setAttribute('x', 30) // Sets the attribute value
```

When an appropriate value for an attribute is supplied, the object will suddenly appear. This allows a developer to program within the debugger window.

5.4 Central control of screen display

A first attempt at transitioning from one screen to another could be performed by directly controlling the view's visibility. This causes one screen to disappear and another screen to suddenly appear and would look something like this:

```
<view id="page1" width="100%" height="100%" bgcolor="blue"/>
<view id="page2" width="100%" height="100%" bgcolor="red"/>
<button text="Change Screen Elements">
   <handler name="onclick">
            page1.setVisible(true);
            page2.setVisible(false);
   </handler>
</button>
<button x="100" text="Change back">
   <handler name="onclick">
            page1.setVisible(false);
            page2.setVisible(true);
   </handler>
</button>
```

Even though this example is relatively simple, it does involve a visual state transition that must be captured within a central state table. As an application becomes more complex, it helps to view it as a set of states with state transition events leading from one state to another. This is a many-to-many relationship since there are many ways to transition back to a particular state and many ways to transition from it.

For example, moving from a login window to the main application window involves two states, `Login` and `Main`, a state transition `Login to Main`, and a state transition event (clicking the OK button). While local events should be handled by an object's event handler, events generating a state transition need to be handled by a global state controller. This controller maintains the application's state through a state transition table. We'll build this table by identifying each state and the paths leading from one state to another, the *state transitions.* Notice that a single state can have multiple state transitions, as there might be more than one way to get there. Since a state is static, we are most interested in the state transitions, as this shows where changes to the current declarative structure must occur. Table 5.1 shows each state transition listed in the controller for the Laszlo Market.

Table 5.1 Laszlo Market states and their state transitions

State	Incoming State Transition
`Login`	`Splash to Login`
`Main`	`Login to Main`
`Main`	`Checkout to Main`
`Checkout`	`Main to Checkout`

In its simplest form, a controller consists of a `node` tag with two attributes, `appstate` and `currstate`, which work together to control communications between this central controller and the rest of the application.

Since we are dealing with an event-driven GUI, all state transitions are initiated by an event requiring the current state of the application to be changed. In the previous example, the `onclick` event handler acted like a controller and directly changed the state of the declarative tags. With a central controller, the `onclick` event handler passes control to the central controller, `gController`, by updating its `appstate` attribute with an argument indicating a transition from the current state to a desired state:

```
  <handler name="onclick">
    gController.setAttribute("appstate", "Main to Checkout");
</handler>
```

The central controller is now responsible for updating the declarative statements to switch states. The `currstate` attribute, settable only within the central controller, contains the name of the new state. It is used to trigger constraint operations within the declarative statements to make the transition from the previous state to this new state. When the `appstate` attribute is set to `Main to Checkout`, this results in the `currstate` attribute being set to the `Checkout` state.

We start building our central controller by listing the state transitions from table 5.1. Although they all appear empty in listing 5.5, we'll update these state-transition case statements with logic in the upcoming sections.

Listing 5.5 A state-transition controller for the Market screen configurations

```
<node id="gController">
   <attribute name="appstate" type="string"/>
   <attribute name="currstate" type="string"/>

   <handler name="onappstate" args="state">
      switch (state) {
         case "Splash to Login":
            this.setAttribute("currstate", "Login");
            break;
         case "Login to Main":
            this.setAttribute("currstate", "Main");
            break;
         case "Main to Checkout":
            this.setAttribute("currstate", "Checkout");
            break;
         case "Checkout to Main":
            this.setAttribute("currstate", "Main");
            break;
         default:
            // Display an error message
            break;
      }
   </handler>
</node>
```

Although this is a simple prototype, you should make it a habit to always use a state controller when developing a Laszlo application. It's much easier to begin with a controller-based design than to deal with the problems of ripping apart an existing code base when it grows too large to support an ad hoc method of handling state.

5.4.1 Designing the screen transitions

Transitioning between the Market application states requires first that declarative statements be established to carry out the following:

- Integrate the main and checkout screens into a larger virtual screen
- Create a transition mechanism to traverse this virtual screen
- Create a trigger to initiate the traversal

Let's start by integrating the screens.

Integrating multiple screens

We'll start by converting our two prototype screens, main and checkout, into top-level views called, not surprisingly, `main` and `checkout`. Since the Shopping Cart and Media Player windows serve as a common element, they need to be removed from the `checkout` view. Since these windows had a width of 25 percent, the updated `checkout` view only has a width of 75 percent. These two views are placed directly after one another within the integrated screen:

```
<view name="main" width="100%" height="100%">
   …
</view>
<view name="checkout" x="${main.x+main.width}"
      width="75%" height="100%">
   …
</view>
```

Displays main view

Stores checkout view off-screen

To attach the checkout screen to the right side of the main screen, set the `checkout` view's `x` attribute to the current position of the `main` view (currently 0) plus its width. The width of the `main` view is equal to the size of the browser window (100 percent), ensuring that the `checkout` view is located off-screen to the right in the initial state.

Now we have a large virtual screen, with the main screen displayed in the browser and the Checkout window hidden to the right where users can access it by scrolling. Next we need to think about the transition to the checkout screen.

Scrolling the virtual screen

Let's walk through the changes that we must make to the declarative statements in order to transition from one screen to another within the virtual screen. Figure 5.12 illustrates these steps:

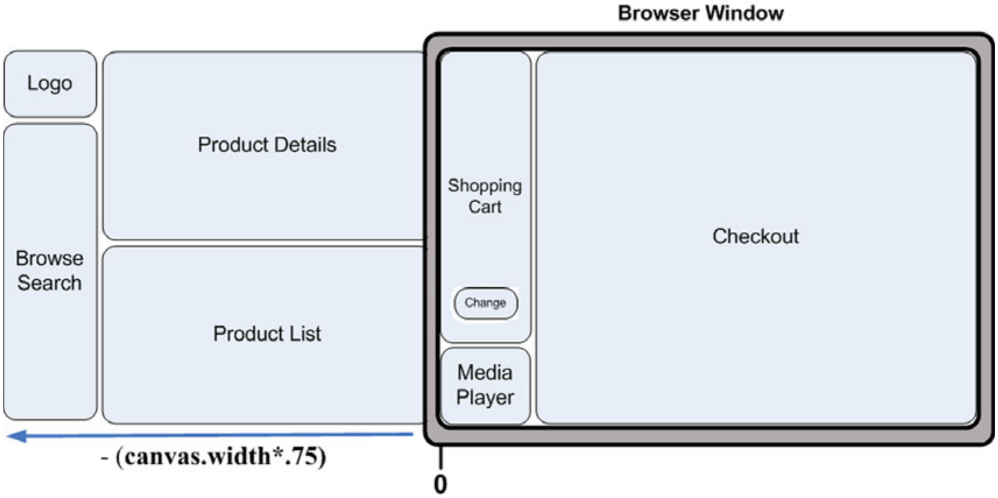


Figure 5.12 The browser window can display only a portion of the larger virtual screen (which consists of two separate views). The virtual screen is animated to the left for a distance of `-(canvas.width*.75)` pixels. After the animation, the upper-left corner of the Shopping Cart window is at position (0, 0).

- The main screen needs to be moved to the left just enough so the Shopping Cart and Media Player are located on the left browser border. This is accomplished by setting the `main` view's `x` attribute to a negative value. This results in 75 percent of the `main` view disappearing off-screen to the left.
- Since the `checkout` view's `x` attribute has a constraint attached, it receives an event when the `main` view's `x` attribute is changed, enabling it to update its value to reflect this change. Since 75 percent of the `main` view is off-screen, this provides enough room to display the entire `checkout` view, since its width is also 75 percent.
- Appropriate actions associated with the Change button are required to initiate the change between these two states, `Main` and `Checkout`.

To transition from the main to the checkout screen, we animate the main screen to move to the left. This is accomplished by animating the `main` view's `x` attribute to a destination value of `-(canvas.width * .75)` over a period of 700 milliseconds with this JavaScript call:

```
main.animate("x", -(canvas.width*.75), 700);
```

To transition from the checkout screen back to the main screen (the original position) we write

```
main.animate("x", 0, 700);
```

We can now update our central controller to represent these actions as states. When an input event requests that the screen configuration transition from main to checkout, this is interpreted internally as a `Main to Checkout` state transition request, resulting in the internal state being updated to `Checkout`:

```
case "Main to Checkout":
   main.animate('x', -(canvas.width*.75), 700);
   this.setAttribute("currstate", "Checkout");
   break;

case "Checkout to Main":
   main.animate('x', 0, 700);
   this.setAttribute("currstate", "Main");
   break;
```

All that remains is to find a declarative mechanism to initiate the state transitions of this central controller.

5.4.2 *Triggering screen transitions*

Clicking the Change button generates an `onclick` event signaling that the checkout screen should appear. The button's `onclick` event handler maintains the current state of the application in the `gController.currstate` attribute. Depending on the current state, the `gController`'s `appstate` attribute is set to a state transition request, causing the central controller to spring into action to update the application's state:

```
<button text="Change" y="80%" align="center">
   <handler name="onclick">
      if (gController.currstate == "Main") {
         gController.setAttribute("appstate",
                                   "Main to Checkout");
         return; }
      if (gController.currstate == "Checkout") {
         gController.setAttribute("appstate",
                                   "Checkout to Main");
         return; }
   </handler>
</button>
```

Signals change to Checkout

Signals change to Main

We can now transition between the `Main` and `Checkout` states, resulting in the screen moving back and forth between the main and checkout screens. But so far, the changes to the declarative structure to transition from one state to another are limited to updating declarative structure with JavaScript code.

The problem with this approach is that each change to the declarative structure has to be coded in JavaScript. This works all right with animating screens, because the animation is limited to changing the x attribute. But suppose there are a number of small visual details that also need updating. Using JavaScript to code individual changes would quickly become tedious. Instead, a method is needed for making wholesale changes to declarative structure.

In our previous example, a button labeled Change is used to initiate the state transition to move between the Main and Checkout states. Instead of a fixed label, this button should display “Checkout” while in the Main state and “Return to Store” while in the Checkout state. In the next section, we’ll see how to use the state tag to control this behavior.

The state tag

The state tag adds a conditional capability to LZX, which can be used to switch among sections of declarative code based on its apply attribute value. This adds an “if … then” behavior to declarative statements, creating *application states* that can be applied during execution. For example, an application could switch between constrained and unconstrained states that exhibit static and dynamic behavior. However, note that the state tag doesn’t result in any optimization, since all declarative tags need to be instantiated whether or not they are used.

In its simplest form, the state tag operates like a binary switch, *on* when its apply attribute is true and *off* when it is false. The apply attribute value can be controlled through its apply and remove methods. Each state tag needs to have a name. Since we already have abstract states known as Main and Checkout, it simplifies things to have these names match.

In listing 5.6, the state tag is used to control the display of the button’s text label. Initially, the button text is set to Checkout. Since the main state is applied, its onclick event handler is applied to the button. Looking at figure 5.13, we see that when the button is clicked, its label changes to the “Return to Store” message. The apply attribute is removed from the main state tag and applied to the Checkout state tag, resulting in a switch to the Checkout state tag’s onclick event handler. This event handler contains the reverse logic whereby the button label is updated to display Checkout, the Checkout state is removed, and the main state is applied, thus completing the button’s cycle of changes.

Listing 5.6 Maintaining declarative structure with the `state` tag

```
<canvas>
   <button text="Checkout">
      <state name="main" apply="true">
         <handler name="onclick">
            this.setText("Return to Store");
            main.remove();
            checkout.apply();
         </handler>
      </state>
      <state name="checkout">
         <handler name="onclick">
            this.setText("Checkout");
            checkout.remove();
            main.apply();
         </handler>
      </state>
   </button>
</canvas>
```

Removes main apply state

Applies Checkout state

Removes checkout apply state

Applies main state

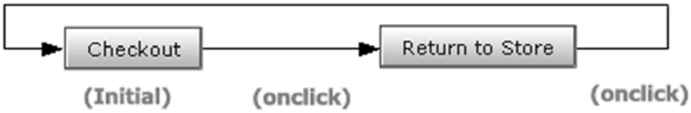


Figure 5.13 The button is initially in the `main` state, displaying a Checkout label. After the user clicks the button to generate an `onclick` event, the button's label is updated to Return to Store, the `main` state is removed, and the `Checkout` state is applied. A subsequent click of the button completes the cycle by updating the button label to Checkout, removing the `Checkout` state, and reapplying the `main` state.

The next step is to coordinate the button with the display of the main and checkout screens. However, while the `apply` and `remove` methods work well for a single state, they quickly become cumbersome with a large number of states. Since we don't want to preclude ourselves from increasing the number of states, we need a way to expand the capabilities of the `state` tag with constraints to allow it to operate like a case statement.

State-based case statements

Rather than explicitly managing the `apply` attribute with the `apply` and `remove` methods, we can use a constraint to support additional states, resulting in a case statement. We'll use a single variable, the `currstate` attribute from our global controller, to select from any number of states. In general, these features provide the `state` tag with the transparency required of a control statement and make it easier to use, since there is no need to remove the previous state. In listing 5.7, the constraint selector appears in bold.

Listing 5.7 Creating a case statement structure with constraints

```
<state apply="${gController.currstate == 'main'}">
   <button x="${(immediateparent.width-this.width)/2}"
           text="Checkout" y="80%" width="100">
      <handler name="onclick">
         this.setText("Return to Store");
         gController.setAttribute("appstate", "Main to Checkout");
      </handler>
   </button
</state>
<state apply="${gController.currstate == 'checkout'}">
   <button x="${(immediateparent.width-this.width)/2}"
           text="Return to Store" y="80%" width="100">
      <handler name="onclick">
         this.setText("Checkout");
         gController.setAttribute("appstate", "Checkout to Main");
      </handler>
   </button>
</state>
```

Switches to main screen

Switches to checkout screen

The state tag approach scales well, because it can be applied to any number of tags and can occur in multiple places throughout the declarative code. Listing 5.7 shows how the declarative structure can be easily updated to represent a new state. Since the current state is provided by the `gController.currstate` attribute, the application's declarative structure is automatically updated to represent a new state. Since constraint notation is used, all updates occur within the declarative structure itself.

We can use the `state` tag even further to refactor our central controller. Our goal is to move as much functionality as possible into the declarative structure, to leverage its greater flexibility and built-in features.

5.5 Refactoring our code

When refactoring code in LZX, look for opportunities to refactor from JavaScript into declarative tags. The most obvious targets are areas containing lots of JavaScript. Our central controller has such a place ripe for code refactoring—the animation scripting.

5.5.1 Replacing the animator

To refactor the central controller, we want to move as much as possible of the JavaScript data presentation code into the declarative structure representing the states, since that declarative structure is the focus of any modifications to the states. Rather than using the JavaScript `animation` method to update the `x` attribute of the `main` view, we'll declare an animator at the top level, targeted at the `main` view. Remember that since the two views are connected by a constraint on the `x` attribute, any changes to the `main` view affect the `checkout` view, so the declarative animators have the same target. A declarative animator benefits from these advantages:

- It can be contained within an `animatorgroup` tag, making possible later participation in concurrent or sequential actions with other animators.
- An animator object associated with an `animator` tag can be reused, while the `animate` method instantiates an animator object for each use.

As listing 5.8 shows, a `state` tag with a `currstate` constraint determines the state to which the animator is applied.

Listing 5.8 Using states to apply an animator

```
<canvas>
  <view name="main" width="100%" height="100%">
     …
  </view>
  <view name="checkout" x="${main.x+main.width}"
        width="75%" height="100%">
     …
  </view>

  <state apply="${gController.currstate == ‘main'}">
     <animator target="main" attribute="x"
               duration="700" to="0"/>
  </state>
  <state apply="${gController.currstate == 'checkout'}">
     <animator target="main" attribute="x"
               duration="700" to="${-(canvas.width*.75)}"/>
  </state>
</canvas>
```

Moves virtual screen right

Moves virtual screen left

An `animator` tag uses its `target` attribute to specify the view against which it executes. This provides a further level of decoupling; because a constraint can be used to set this field, the animator could potentially be targeted against any of the major views. We can now think in terms of *operators*—animators or other tags that are applied—and *entities*—states. This provides us with the flexibility to mix and match operators against entities.

5.5.2 A general-purpose architecture

Our current Market prototype design forms the basis for a general-purpose architecture for many Laszlo applications. This architecture uses a mirrored state transition table to allow the declarative statements to be defined as operators and entities. Figure 5.14 illustrates this diagrammatically.

The operators are controlled by state, so we can easily add any number of operators and any number of views. This allows different animator-based classes to be applied to manipulate these views in different ways. The manipulation can occur intra-screen, allowing an application to be composed of a series of overlapping interior layers. Or, as in the Laszlo Market, the layers can be represented as exterior screens within a larger virtual view. This architecture scales to support any number of additional entities and operators.

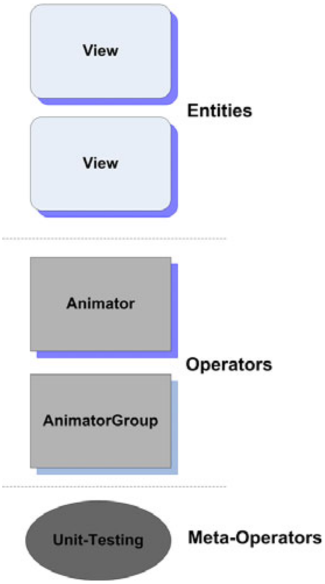


Figure 5.14 Our general-purpose architecture allows the declarative statements to be viewed as a set of entities along with operators that operate on them. A meta-operator operates on other operators.

We'll complete this architecture by adding the final piece, meta-operators, which are operators that operate on other operators. We'll use this last piece in our last section on unit testing.

5.6 Testing with LzUnit

Writing test cases, a cornerstone of *extreme programming*, is well supported in LZX. Unit tests are easy and natural to write, because they use declarative statements. A test suite consists of individual unit tests that specify the expected characteristics of the application. Unit tests easily map to and test specific tags. Once an application passes a test suite, it constitutes a *reference implementation* for further ongoing regression testing as development and refinement proceed.

After establishing our testing, we'll have the freedom to refactor our code or head off in a new design direction with the assurance that, if we break anything, the problems should immediately become apparent with regression testing.

5.6.1 Unit testing with LzUnit

Adding a unit-testing framework to a Laszlo application requires only that the LzUnit library be included:

```
<include href="lzunit"/>
```

The LzUnit framework provides two tags, TestSuite and TestCase. Each TestSuite element contains one or more instances of the TestCase element to perform a test suite. Test suites are differentiated by name to allow multiple suites in an application.

When a test suite is executed, it automatically runs its entire suite of child test cases, reporting the number of cases run, the number of failures, and the number of runtime errors. A large variety of assertions are available for testing almost any situation. Table 5.2 provides a complete list of the conditions that can be tested.

Table 5.2 Assertions for performing unit testing

Calling Method	Assertion Tested
assertEquals(expected, actual, message)	An actual value equals (==) an expected value.
assertFalse(condition, assertion)	A condition is false.
assertNotNull(object, message)	A value is not (!==) null.
assertNotSame(expected, actual, message)	An actual value is not the same as (!==) an expected value.
assertNotUndefined(object, message)	A value is not undefined.
assertNull(object, message)	A value is (===) null.
assertSame(expected, actual, message)	An actual value is the same as (===) an expected value.
assertTrue(condition, assertion)	A condition is true.
assertUndefined(object, message)	A value is undefined.

Optional setUp and tearDown methods exist for each test case. Since every test run must execute cleanly, test data is loaded with setUp and later removed with tearDown. By default, the order of execution of individual tests within a test case is not guaranteed. To have tests run in consecutive order, the global flag asynchronoustests must be set to false:

```
<script>
   asynchronoustests = false
</script>
```

Since tests are declarative, they are executed in sequential order at the next idle event. However, a test that takes a long time to execute may complete after a subsequent but shorter test.

Be careful to ensure that any condition being tested is ready for testing. In our case, since we are testing the animation of the storefront, which takes 700 milliseconds, we must ensure that our tests occur after the animation has completed. To accomplish this, we'll chain our test cases to ensure they run sequentially. To chain test cases, we'll wait for the onstop event, signaling completion of animation, and then invoke our next test case from within the onstop event handler.

5.6.2 Testing the Laszlo Market

To ensure our animation sequence doesn't accidentally break as a result of future code changes, let's add some unit testing. Our centralized controller framework is scalable, which means we can easily add a unit-testing state by adding another case to the state transition switch statement:

```
<node name="gController">
   ...
   <handler name="onappstate" args="state">
      switch (state) {
   case 4:
           title = "TestSuite";
           this.testsuite.apply();
           break;
      ...
   </handler>
</node>
```

Since unit testing tests the other operators, which depend on the currstate attribute accurately reflecting the current state, unit testing can't just be another state within the case statement. We need to use the binary apply and remove mechanisms for the state containing the unit testing.

Next, the declarative portion of the controller is updated with a meta-operator section containing TestSuite tags that exercise and test the other operators; see listing 5.9. The test cases are represented as methods chained to execute sequentially. Within the unit-testing state, the currstate attribute is manually updated to control state transitions. All test cases that begin with the test prefix automatically initiate a test; we have set up the methods with a _test suffix to check the results.

Listing 5.9 A Laszlo Market test suite

```
<state name="testsuite" apply="false">
   <TestSuite name="testsuite">
      <TestCase name="testcase">
         <method name="testCheckout">
            gController.setAttribute("currstate", "checkout");
         </method>
         <method name="checkout_test">
            assertEquals(-(canvas.width*.75)+3, main.x);
            gController.setAttribute("currstate", "main");
         </method>
         <method name="main_test">
            assertEquals(0, main.x);
            Debug.write(”test complete”);
         </method>
      </TestCase>
   </TestSuite>
</state>
```

Causes transition to Checkout state

Does checkout test and chains to main state

Does main test

The method testCheckout causes a transition to the Checkout state. We update the animator tags to handle the onstop event and invoke the checkout_test method to enable the next test within the chain; see listing 5.10.

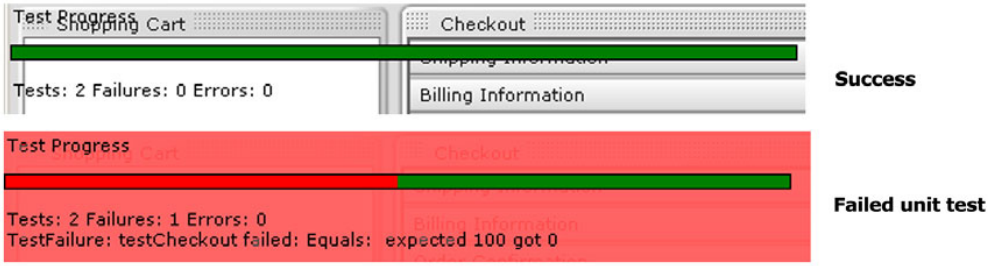
Listing 5.10 Invoking the Laszlo Market tests

```
<state name="main"
       apply="${gController.currstate == 'main'}">
   <animator name="anim" target="main" attribute="x"
             to="0" duration="700"
             onstop="testsuite.testcase.main_test()"/>
</state>
<state name="checkout"
        apply="${gController.currstate == 'checkout'}">
   <animator name="anim" target="main" attribute="x"
             to="${-(canvas.width*.75)+2.5}" duration="700"
             onstop="testsuite.testcase.checkout_test()"/>
</state>
```

Invokes main test

Invokes checkout test

Finally, we turn on unit testing by setting the transition state to the appropriate value in the oninit event handler. When you run unit tests, a progress bar displays. A green bar appears for each successful test, and a red bar for each failing test. If one or more tests fail, the background of the test status area turns red. Since the printed page can only display grayscale, you need to imagine figure 5.15 displaying the appropriate colors.



Test Progress

Tests: 2 Failures: 0 Errors: 0

Test Progress

Tests: 2 Failures: 1 Errors: 0
TestFailure: testCheckout failed: Equals: expected 100 got 0

Figure 5.15 A successful unit test is indicated by a green progress bar in the upper-left corner of the screen. An unsuccessful unit test causes the screen to turn red. Information about the failed test is displayed in the upper-left corner of the screen.

During testing, you can watch the animations occur sequentially, as shown in figure 5.16. It's a bit like watching a "ghost in the machine."

The last step involves setting up unit testing so that it is resident within your application to be invoked with a URL query string. This removes the need to maintain several different versions, thus allowing a deployed application to have unit-test facilities to assist in tracking down problems.

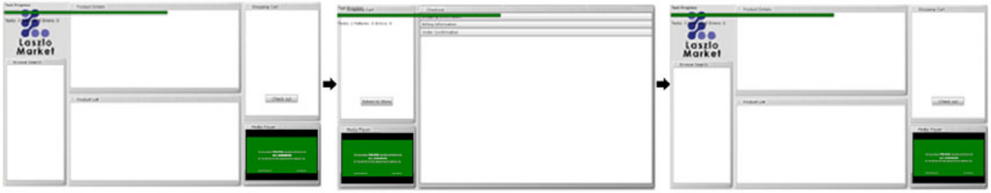


Figure 5.16 The state transitions can be tested both visually and with the green unit testing progress bar. As long as it's green, everything is fine.

5.6.3 Testing from a URL query string

Using a URL query string to control unit testing lets you test a deployed Laszlo application without removing it from a server or making any source code modifications. Controlling unit testing from a URL query string requires the LzBrowser service. We simply need to update the oninit handler to check for the lzunit string in a URL query string to initiate unit testing:

```
<canvas>
   <handler name="oninit">
      if (LzBrowser.getInitArg("lzunit") == "true")
         gController.setAttribute("appstate", 4);
      else
         gController.setAttribute("appstate", 1);
   </handler>
   ...
</canvas>
```

Here's an example of a URL query string containing an `lzunit` parameter:

```
http://localhost:8080/lps/book/main.lzx?lzr=html&lzunit=true)
```

Now it's easy to move from development to unit-testing mode without needing any reconfiguration. It's important that unit testing be easy, so this critical step isn't ignored.

5.7 Putting it all together

The general architecture and skeleton for the Laszlo Market application can be listed in a little over 100 lines of code—and this even includes an embedded state controller and unit-testing facilities. Our skeleton includes the window layout for each of the major screens, main and checkout, and the animation to move from one to the other.

Since various code examples are scattered throughout this chapter, we've brought them all together in listing 5.11 to allow you to see the code in its entirety.

Listing 5.11 The complete Laszlo Market prototype

```
<canvas width="100%" height="100%" bgcolor="0xCCCCCC">
   <include href="lzunit"/>
   <include href="resources.lzx"/>

   <script>
      asynchronoustests = false
   </script>

   <handler name="oninit">
      if (LzBrowser.getInitArg("lzunit") == "true")
         gController.setAttribute("appstate", "UnitTest");
      else
         gController.setAttribute("appstate", "Login to Main");
   </handler>

   <node name="gController">
      <attribute name="appstate"  value="" type="string"/>
      <attribute name="currstate" value="" type="string"/>

      <handler name="onappstate" args="state">
         switch (state) {
            case "Splash to Login":
               this.setAttribute("currstate", "Login");
               break;
            case "Login to Main":
               this.setAttribute("currstate", "Main");
               break;
            case "Main to Checkout":
               this.setAttribute("currstate", "Checkout");
               break;
            case "Checkout to Main":
               this.setAttribute("currstate", "Main");
               break;
            case "UnitTest":
               testsuite.apply();
               break;
            default:
               // Display an error message
               break; }
      </handler>
   </node>

   <view name="main" width="100%" height="100%">
      <view name="header" width="20%" height="30%"
            resource="logo" x="30" y="15"/>
      <window name="details" title="Product Details"
              x="${main.header.width}"
              width="55%" height="50%" resizable="true"/>
      <window name="shopcart" title="Shopping Cart" x="75%"
              width="25%" height="65%" resizable="true">
         <state apply="${gController.currstate == 'Main'}">
            <button x="${(immediateparent.width-this.width)/2}"
                    text="Checkout" y="80%" width="100">
               <handler name="onclick">
                  this.setAttribute("text", "Return to Store");
                  gController.setAttribute("appstate",
                                           "Main to Checkout");
               </handler>
            </button>
         </state>
         <state apply="${gController.currstate == 'Checkout'}">
            <button x="${(immediateparent.width-this.width)/2}"
                    text="Return to Store" y="80%" width="100">
               <handler name="onclick">
                  this.setAttribute("text", "Checkout");
                  gController.setAttribute("appstate",
                                           "Checkout to Main");
               </handler>
            </button>
         </state>
      </window>
```

```
      <window name="browse" title="Browse Search"
              y="${main.header.height}"
              width="20%" height="70%" resizable="true"/>
      <window name="productlist" title="Product List"
              x="${main.browse.width}"
              y="${main.details.height}"
              width="55%" height="50%" resizable="true"/>
      <window name="mediaplayer" title="Media Player"
              x="${main.browse.width+main.productlist.width}"
              y="${main.shopcart.height}"
              width="25%" height="35%" resizable="true">
         <view stretches="both"
               width="100%" height="100%" resource="video"/>
      </window>
   </view>

   <view name="checkout" x="${main.x+main.width}"
         width="75%" height="100%">
      <tabslider name="checkoutsteps"
                 width="100%" height="100%"
                 spacing="2" slideduration="300">
         <tabelement text="Shipping Information"/>
         <tabelement text="Billing Information"/>
         <tabelement text="Order Confirmation"/>
      </tabslider>
   </view>

   <state apply="${gController.currstate == 'Main'}">
      <animator target="main" attribute="x"
                duration="700" to="0"
                onstop="testsuite.testcase.main_test()"/>
   </state>
   <state apply="${gController.currstate == 'Checkout'}">
      <animator target="main" attribute="x"
                duration="700" to="${-(canvas.width*.75)}"
                onstop="testsuite.testcase.checkout_test()"/>
   </state>

   <state name="testsuite" apply="false">
      <TestSuite name="testsuite">
         <TestCase name="testcase">
            <method name="testCheckout">
               gController.setAttribute("currstate", "Checkout");
            </method>
            <method name="checkout_test">
               assertEquals(-(canvas.width*.75), main.x);
               gController.setAttribute("currstate", "Main");
            </method>
            <method name="main_test">
               assertEquals(0, main.x);
               Debug.write('test complete');
```

```
                </method>
            </TestCase>
        </TestSuite>
    </state>
</canvas>
```

This completes the general architecture of the Laszlo Market. We've covered a lot of ground, but have completed a working skeleton for our application. This skeleton will continue to be enhanced throughout the upcoming chapters. We'll cover topics that range from working with layouts and components, displaying the contents of local XML datasets, and providing an application-branded appearance, to integrating Laszlo Flash and DHTML components. We finally convert our prototype into a deployed application interfacing to an HTTP server. It's a long ride and hopefully you'll enjoy the many stops along the way.

5.8 Summary

The objective of this chapter was to demonstrate Laszlo LZX by building a prototype representative of typical Laszlo applications. Design issues focus on the need to maintain visual continuity across different screens. The design process consists of initially using wireframes to identify business processes and to provide an initial representation for their interfaces. These initial low-fidelity wireframe models are used as playing pieces within a storyboarding process to develop the application's transition states. Afterward, more refined higher-fidelity wireframe models are developed to specify layout requirements. These layout specifications can generally be directly encoded in LZX.

A major part of this chapter dealt with determining a design strategy for integrating multiple screens into a single screen. Different strategies are available, such as using external or internal screens. For the Laszlo Market, we decided to use external screens and animate the transition from one screen configuration to another.

We demonstrated the use of a state transition table for implementation. This table serves as a central MVC controller and helps maintain the separate roles of control and process. After we completed the initial prototype, we highlighted several areas in the source code that benefited from code refactoring. The flexibility of this architectural approach allows unit testing to be easily added and controlled within an application. Unit-testing code can be controlled through a URL query string, allowing the application to be unit tested even after deployed.

At this point, the prototype captures the high-level workflow of the Laszlo Market. The next step in the continued development of the prototype is to concentrate on intra-screen design and development. But before we can begin designing the details of the main and checkout screens, we need to take a detour to better understand the operation of layouts.

Part 2

Prototyping the Laszlo Market

This second part of the book concentrates on the Laszlo Market to illustrate new topics. The first of these is the *layout* object, containing an algorithm for creating a visual pattern using a set of view-based objects. From layouts, we move to *reusable components*, which are demonstrated by building the panes in the Shipping Information page of the Laszlo Market. In this context, we devote a considerable amount of time discussing the important topic of field validation. Third, we examine how the Laszlo event-handling system is based on *event* and *delegate* objects and how the delegate mechanism allows a tag's behavior to be changed during execution. Finally, the Laszlo *input services* are used to provide a protected login featuring a modal registration window, tool-tip help messages, keyboard shortcuts for store-related functions, and an introduction to drag-and-drop operations. At the end of this part, you can see the Market start to take shape as a real online store.

6 Laying out the Laszlo Market

This chapter covers

- Solving common layout problems
- Creating custom layouts
- Creating form layouts

> *See first that the design is wise and just: that ascertained, pursue it resolutely; do not for one repulse forego the purpose that you resolved to effect.*
>
> —William Shakespeare

In the previous chapter, we completed a skeletal shell for the Laszlo Market. In this chapter, we'll use *layouts* to build the individual screens for our application. A layout object contains an algorithm designed to create a pattern for a set of sibling view-based objects. These patterns range from a gridlike layout, similar to typical city blocks, to stretchable layouts with variable-length borders. Screen configurations of varying complexity can be handled using different layout combinations.

A screen layout must be flexible enough to be resized without clipping design elements or distorting the overall alignment. A static layout doesn't afford this flexibility, since design changes result in time-consuming manual layout reconfigurations. The key to providing this flexibility is to represent a screen layout with an algorithm. This produces an initial configuration of view objects conforming to a relationship established by this algorithm. Dynamic reconfiguration maintains this relationship when the dimensions of any constituent object are modified. Because a layout contains an algorithm, it can be used in a mathematical way to produce additive compound layouts; this allows some layouts to be used as *layout modifiers.*

Laszlo provides a set of high-level layout tools for handling common layout configurations and a set of low-level API interfaces for creating new layouts. In this chapter, we'll limit ourselves to examining only static configurations of custom layouts. In chapter 8 we'll discuss adding dynamic configuration capabilities to custom layouts.

The chapter starts by exploring layout tools for common problems, moves on to the fundamentals of custom layouts, and ends by showing an important application of custom layouts for laying out input forms that perform validation.

6.1 Common layout problems

Laszlo provides a comprehensive assortment of layout tags to address a wide range of situations, ranging from rectilinear—horizontal or vertical—placement, fixed margins, and stretchable, rotational, and dynamic patterns. All `layout` tags extend the `LzLayout` class. They attach to a parent view and apply a layout algorithm to the parent view's subviews array to create a layout configuration for its child views. By default, all views position themselves at `x="0"` and `y="0"` in their coordinate space. If the `x` and `y` attributes are left unset, the sibling views will end up on top of each other. When a layout is attached to a view, the layout has access to the view's subviews array. The base `layout` class contains an `update` method that contains the algorithm describing the layout configuration. It is called when the parent view completes initialization and arranges the child views into the layout pattern. The source code for the layout tools is supplied with the Laszlo distribution and is found in the directory $LPS_HOME/lps/components/utils/layouts. The attributes listed in table 6.1, which control the major characteristics of a layout, are supported by all layouts.

Table 6.1 Attributes shared by layout classes

Name	Data Type	Tag or Script	Attribute Type	Description
`axis`	`string`	Both	Setter	The axis in which the layout operates; either x or y
`inset`	`number`	Both	Setter	The number of pixels to inset the first view controlled by the layout
`spacing`	`number`	Both	Setter	The number of pixels to use between each view in the layout

Layouts are generally oriented along either the x- or y-axis. If the `axis` attribute is not specified, orientation defaults to the y-axis. The `inset` attribute specifies an initial spacing value for the first child view of the layout. The `spacing` attribute specifies the amount of space between adjacent child views. The default value for both `spacing` and `inset` is 0. A layout is performed only on a set of sibling views and can't descend the parent-child hierarchy. A layout can either be directly attached as an attribute:

```
<view layout="axis: x; spacing: 5"/>
```

or attached as a child node to the parent:

```
<view>
   <layout/>
   ...
</view>
```

Layouts can be categorized into these groups; the *basic layouts* group includes the `simplelayout` and its related variations. The *stretchable layouts* group addresses situations requiring a stretchable border. The final group consists of layout modifiers that can be used to add dynamic capabilities to the other layout tags.

6.1.1 Basic layouts

The simplelayout, the most popular layout, serves as the base pattern used by the other layouts to modify with variations. These variations include reversing the layout, extending it over multiple rows, rotating the subview, or modifying with a constant value.

Using the simplelayout tag

A simplelayout lays out a set of child views along a vertical or horizontal axis with consistent spacing. The variations allow the order of child views to be reversed, modified by an offset, wrapped over multiple rows, or changed to accommodate a rotated member. The initial view can also be inset by a specified amount. This tag ignores conflicting x or y attributes for any of its subviews. Listing 6.1 illustrates its behavior.

Listing 6.1 Using a simplelayout

```
<canvas>
   <class name="bar" width="20" height="100" bgcolor="0xDDDDDD"/>
   <simplelayout axis="x" spacing="5"/>
   <bar x="40"/>
   <bar x="20" y="10"/>
   <bar x="-40"/>
</canvas>
```

Since the layout orientation is along the x-axis, the x attributes of the contained child views are discarded. This results in a consistent spacing of 5 pixels between the child views. Since the y attribute is not in conflict with the layout orientation, it is applied, as shown in figure 6.1, resulting in the second child view having an offset of 10 pixels along the y-axis.

The analogous result holds for the reverse situation, when there are conflicting y attributes in a layout oriented along the y-axis.

A simplelayout can handle dynamic changes. When any child view controlled by a layout changes its size or visibility, the simplelayout's update method is invoked with its layout algorithm:

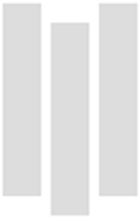


Figure 6.1 Conflicting x attributes are discarded for a simplelayout with a matching axis. Since the y attribute isn't in conflict, it is used in this layout.

```
<canvas>
   <simplelayout axis="x" spacing="2"/>

   <view width="100" height="100" bgcolor="0xBBBBBB"/>
   <view width="100" height="100" bgcolor="0xCCCCCC"
                onclick="this.setWidth(50)"/>
   <view width="100" height="100" bgcolor="0xDDDDDD"/>
</canvas>
```

The results are shown in figure 6.2, where the middle view has its size changed after the initial layout has occurred. When the width is changed this results in the re-execution of the layout algorithm.

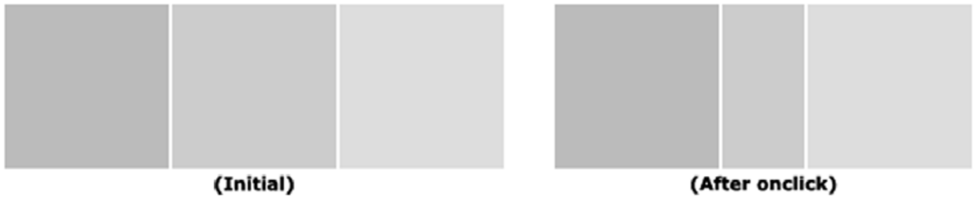


Figure 6.2 Changing the width of a child view participating in a layout results in the layout algorithm being executed.

Analogously, when the visibility of the middle view is turned off, the `simplelayout` is re-executed to close the gap produced by the now nonvisible view:

```
<canvas>
   <simplelayout axis="x" spacing="2"/>

   <view width="100" height="100" bgcolor="0xBBBBBB"/>
   <view width="100" height="100" bgcolor="0xCCCCCC"
                onclick="this.setAttribute('visible', false)"/>
   <view width="100" height="100" bgcolor="0xDDDDDD"/>
</canvas>
```

You can see the results in figure 6.3.

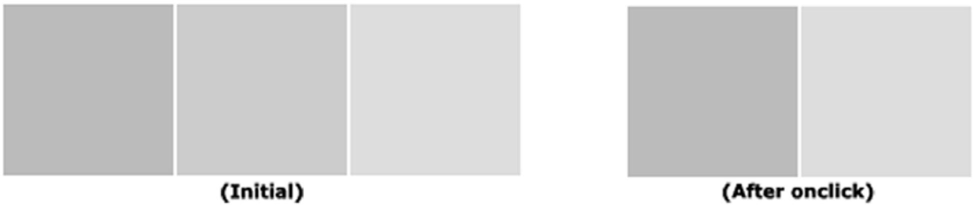


Figure 6.3 Changing the visibility of a child view participating in a layout results in the layout being executed.

Changing the x or y coordinates with a setAttribute, setX, or setY call that matches the layout orientation will not cause the layout to be updated. Instead, it results in the layout spacing becoming irregular. Therefore, these calls shouldn't be used with a matching layout.

Reversing a layout

The reverselayout is similar to the simplelayout, except that it reverses the order of the subviews. The reverselayout tag is useful for situations where information is stored in last-in, first-out order. Although this doesn't have much applicability with declarative objects, (see listing 6.2), the situation can easily occur with replicated views from a dataset.

Listing 6.2 The reverselayout, which displays the views in reverse order

```
<canvas>
   <view height="100" width="100" bgcolor="0xCCCCCC">
      <text>4</text>
      <text>3</text>
      <text>2</text>
      <text>1</text>
      <reverselayout end="false"/>
   </view>
</canvas>
```

The reverselayout's end attribute pushes the subviews to the bottom of the parent view and is the default action for the reverselayout tag. To get the reversed views to display at the beginning of the parent view, the end attribute must be reset to false. The effects of using the end attribute are shown in figure 6.4.

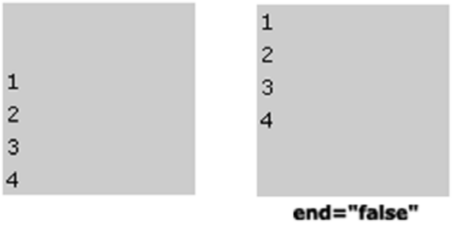


Figure 6.4 The end attribute for the reverselayout tag controls the placement of the layout. Normally, it appears at the bottom of the containing view, but setting end="false" causes the results to be displayed at the top of the containing view.

Up to now, we have dealt with situations where there is sufficient room to contain the child views. The wrappinglayout is designed for situations where line wrapping is required.

Line wrapping

The wrappinglayout is similar to the simplelayout, except that it supports line wrapping. When the number or size of the child views exceeds the available space, this layout wraps the excess subviews onto the next line. Additional attributes, shown in table 6.2, are used to control the placement of the initial row and for subsequent wrapped lines.

Table 6.2 The wrappinglayout line-spacing attributes

Name	Data Type	Tag or Script	Attribute Type	Description
xinset	number	Both	Setter	The number of pixels to offset the first view controlled by the layout on the x-axis
xspacing	number	Both	Setter	The number of pixels to use between the views controlled by the layout on the x-axis
yinset	number	Both	Setter	The number of pixels to offset the first view controlled by the layout on the y-axis
yspacing	number	Both	Setter	The number of pixels to use between the views controlled by the layout on the y-axis

Listing 6.3 illustrates each of the attributes in table 6.2.

Listing 6.3 Wrappinglayout used to wrap views across multiple lines

```
<canvas>
   <view x="10" y="10" width="400" height="150" bgcolor="0xBBBBBB">
      <wrappinglayout axis="x" xinset="10" xspacing="20"
                               yinset="10" yspacing="20"/>
      <view width="100" height="50" bgcolor="0xCCCCCC"/>
      <view width="100" height="50" bgcolor="0xCCCCCC"/>
      <view width="100" height="50" bgcolor="0xCCCCCC"/>
      <view width="100" height="50" bgcolor="0xCCCCCC"/>
      <view width="100" height="50" bgcolor="0xCCCCCC"/>
   </view>
</canvas>
```

Figure 6.5 shows the effect of these spacing and inset attributes when applied to a layout.

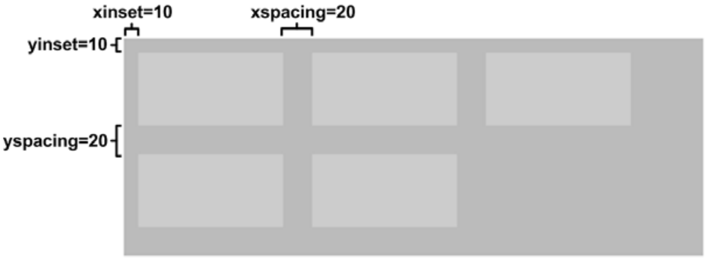


Figure 6.5 The `wrappinglayout` causes extra views that don't fit on the initial axis to be placed along a parallel axis whose distance is specified by the `spacing` attribute. More tightly controlled spacing can be achieved with the `xinset`, `xspacing`, `yinset`, and `yspacing` attributes.

If the `xspacing` and `yspacing` attributes are not set, their values default to that of the `spacing` attribute. Likewise, if the `xinset` and `yinset` attributes are unset, their values default to that of the `inset` attribute. When these attributes are specified, they override the `spacing` and `inset` attributes.

Rotational layouts

A rotational layout adjusts one or more rotated child views so they are evenly arranged along an axis. The rotational layout thus serves as a bridge between a rotated view and other rectilinear views. Listing 6.4 illustrates this using the `simpleboundslayout` tag. Suppose we had an initial display of three colored squares, lined up as shown on the left in figure 6.6:

```
<canvas>
   <view bgcolor="0xDDDDDD">
      <view width="60" height="60" bgcolor="0xCCCCCC"/>
      <view width="60" height="60" bgcolor="0xBBBBBB"/>
      <view width="60" height="60" bgcolor="0xCCCCCC"/>
      <simplelayout axis="x"/>
   </view >
</canvas>
```

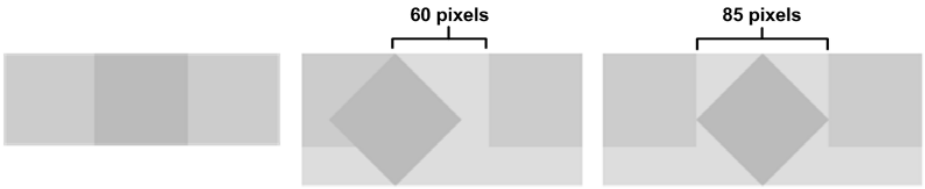


Figure 6.6 In this sequence, we move from a `simplelayout` to an attempt to simply rotate a view, to finally using `simpleboundslayout` to fit the rotated view correctly.

We then rotate the middle view by 45 degrees by adding a `rotation="45"` attribute to it:

```
<view width="60" height="60" bgcolor="0xBBBBBB" rotation="45"/>
```

Listing 6.4 Aligning a rotated view with `simpleboundslayout`

```
<canvas>
   <include href="utils/layouts/simpleboundslayout.lzx"/>
   <view bgcolor="0xDDDDDD">
      <view width="60" height="60" bgcolor="0xCCCCCC"/>
      <view width="60" height="60" bgcolor="0xBBBBBB" rotation="45"/>
      <view width="60" height="60" bgcolor="0xCCCCCC"/>
      <simpleboundslayout axis="x"/>
   </view>
</canvas>
```

Since a view rotates around its upper-left corner, this doesn't produce the desired effect of centering the rotated view; the view is misaligned and overlaps the first view. Manually positioning this rotated view would be a messy and error-prone exercise. Not only would the `xoffset` and `yoffset` attributes need to be correctly set, but the `x` and `y` attributes for the surrounding views would also have to be altered to provide additional space for the rotated center view, as we see on the right side of figure 6.6. The `simpleboundslayout` tag handles all the calculations for us.

Of course, the `simpleboundslayout` doesn't just work on only a single view, but can be extended to rotate any number of child views to produce the pattern seen in figure 6.7.

These views could also be rotated along the y-axis to produce a different type of border. Now that we have seen the basic layouts, let's look at a *layout modifier.*

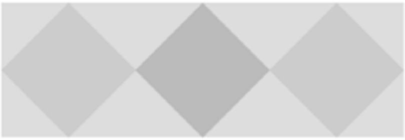


Figure 6.7 This pattern is produced by rotating all the child views within a parent view and using `simpleboundslayout`.

Modifying a layout

A common layout requirement is a fixed margin. The `constantlayout` tag can be used as a layout modifier to supply this fixed margin. Since `constantlayout` only provides an offset, it requires another layout to be useful.

In listing 6.5, the `constantlayout` tag modifies the layout created by the `simplelayout` tag to create a left margin of 20 pixels.

Listing 6.5 Using the constantlayout tag to create a left margin

```
<canvas>
   <class name="bar" height="20" bgcolor="0xCCCCCC"/>
   <view width="400" height="150" bgcolor="0xBBBBBB">
      <simplelayout axis="y" spacing="20"/>
      <constantlayout axis="x" value="20"/>
      <bar width="80"/>
      <bar width="100"/>
      <bar width="90"/>
      <bar width="90"/>
   </view>
</canvas>
```

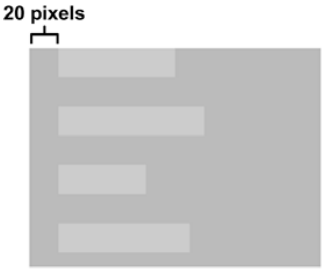


Figure 6.8 The constantlayout works in conjunction with another layout to perform layout modification. In this case, it modifies the layout by shifting it along the x-axis by 20 pixels.

Figure 6.8 shows the subviews offset by the 20-pixel margin specified by the constantlayout tag. The constantlayout tag specifies an orthogonal, or cross, axis in relation to the other layout allowing it to produce a consistent effect. It can't have a matching axis orientation with the other layout, because it would be ignored because of the conflict.

All the basic layouts, with the exception of the wrappinglayout, display a static layout. Although views can change their size, the layout doesn't stretch or compress these views. If the parent view has a fixed space, then, depending on their size and number, the subviews could fall short of filling the available space or extend beyond it. When the layout must be sized to exactly fill a parent view's space, we need to use *stretchable layouts.*

6.1.2 Stretchable layouts

Stretchable layouts have many uses for creating borders, resizable buttons, or anything that requires a variable-length mid-section. A stretchable layout at a minimum consists of a parent view with at least three child views: two end views and a middle view. Generally, the two end views maintain a consistent appearance and size, and serve as terminators or bookends. When the parent view's size changes, the middle views expand or contract, horizontally or vertically, to fill the available area.

The stableborderlayout tag

The stableborderlayout, the simplest stretchable layout, works with three subviews: the middle view stretches to fill the space between the two fixed-length end pieces. In the following example, the parent view has a width of 200 pixels and contains three bars. In listing 6.6, the first and last bar have a fixed width of 20 pixels, while the middle bar purposely has no width specified since it stretches to fill the available space.

Listing 6.6 The stableborderlayout, which stretches a middle section to create a border

```
<canvas>
   <class name="bar" height="100"/>
   <view name="container" x="10" y="10" width="200">
       <stableborderlayout axis="x"/>
       <bar width="20" bgcolor="#BBBBBB"/>
       <bar bgcolor="#CCCCCC"/>
       <bar width="20" bgcolor="#BBBBBB"/>
       <handler name="onclick">
          this.setAttribute("width", this.width+20);
       </handler>
    </view>
</canvas>
```

Omits width to make stretchable

An onclick event handler is added to the parent view to enlarge it by 20 pixels with each click. This causes the middle bar to expand to fill the available space. The result in figure 6.9 shows how the second subview resizes to fill the middle section.

But what if you have more than three child views? For that, the resizelayout tag works like a stableborderlayout tag that contains multiple stretchable views. This is useful for laying out a row containing a number of variable-sized images separated by a common spacing size.

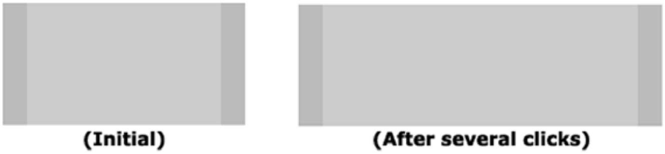


Figure 6.9 The stableborderlayout causes the middle section to expand to fill the space between two fixed-length end pieces. This layout handles dynamic changes by continuing to expand its middle section.

The resizelayout tag

The `resizelayout` tag is similar to `stableborderlayout`, but works with any number of resizable middle views. To specify a resizable child view, its `options` attribute is set to `releasetolayout`. When multiple views have `releasetolayout` set, the available space is split evenly among those views. Listing 6.7 shows three fixed-size child views and two resizable child views.

Listing 6.7 The `resizelayout` tag, which stretches multiple views

```
<canvas>
   <class name="bar" height="100"/>
   <view name="container" x="10" y="10" width="200">
      <resizelayout axis="x"/>
      <bar bgcolor="#CCCCCC" width="20"/>
      <bar bgcolor="#BBBBBB" options="releasetolayout"/>
      <bar bgcolor="#CCCCCC" width="20"/>
      <bar bgcolor="#BBBBBB" options="releasetolayout"/>
      <bar bgcolor="#CCCCCC" width="20"/>
      <handler name="onclick">
         this.setAttribute("width", this.width+20);
      </handler>
   </view>
</canvas>
```

Once again, the `onclick` event handler increases the width of the parent view. This forces the two darker-colored views to resize to accommodate the increased width. You can see the result in figure 6.10.

In the next section, we'll see how these layouts can be modified to act in a dynamic manner.

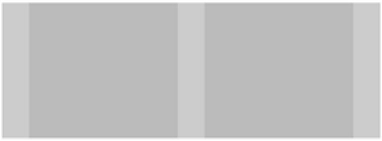


Figure 6.10 The `resizelayout` tag works with several subviews to maintain a stable layout. The resizable views are identified by the `releasetolayout` attribute.

6.1.3 *Dynamic layout modifiers*

The `resizestate` tag is a layout modifier for the stretchable layouts or the `wrappinglayout` to provide them with a dynamic resizing capability. With dynamic resizing, the `width` and `height` attributes of a parent view change in response to the movement of the mouse. This produces an animated effect as the layout continually updates in response to the size changes of the parent view.

In listing 6.8, we take an earlier example and modify its operation with the `resizestate` tag. We also add `onmousedown` and `onmouseup` event handlers to activate and deactivate the animated properties associated with the `resizestate` tag. Since the resizestate tag is derived from the state tag, apply and remove methods can be used to turn its actions on and off.

Listing 6.8 Modifying a layout with the resizedstate tag

```
<canvas>
   <view x="10" y="10" width="400" height="150" bgcolor="0xBBBBBB"
         onmousedown="rs.apply()" onmouseup="rs.remove()">
      <resizestate name="rs"/>
      <wrappinglayout axis="x" spacing="15"
                      xinset="10" xspacing="20"
                      yinset="10" yspacing="20"/>
      <view width="100" height="50" bgcolor="0xCCCCCC"/>
      <view width="100" height="50" bgcolor="0xCCCCCC"/>
      <view width="100" height="50" bgcolor="0xCCCCCC"/>
      <view width="100" height="50" bgcolor="0xCCCCCC"/>
      <view width="100" height="50" bgcolor="0xCCCCCC"/>
   </view>
</canvas>
```

The code in listing 6.8 creates an initial wrappinglayout with five sibling subviews contained within a darker-colored parent view. Dragging the corner of the darker background square with the mouse causes the parent view to change shape. The child views respond by dynamically rearranging themselves to fill this new shape, as shown in figure 6.11.

Although we have only shown this dynamic layout modifier with the supplied layout tags, it can also be used with your own custom layouts to produce innovative effects.

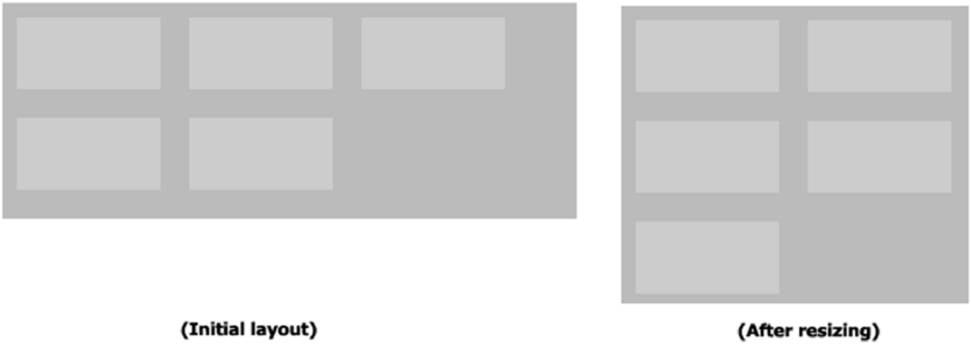


Figure 6.11 A wrappinglayout provides a dynamic animated layout capability when used with resizestate.

6.1.4 Opting out of layouts

There are always exceptions to rules and patterns. In some situations a view is to be excluded from participating in a layout. To do so, the view's `options` attribute is set to `ignorelayout`. When a view is ignored by a layout, it can be layered on top of another layout. Listing 6.9 shows how this could be used to create a centered text label on top of a stretchable view.

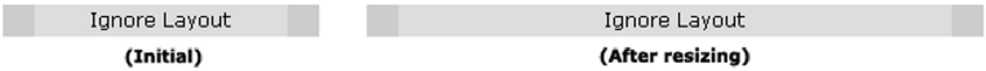
Listing 6.9 Using `ignorelayout` to create a centered text label

```
<canvas>
   <class name="bar" width="100" height="20"
          bgcolor="0xCCCCCC"/>

   <view width="200" y="10" x="10"
         onmousedown="rs.apply()" onmouseup="rs.remove()">
      <resizestate name="rs"/>
      <stableborderlayout axis="x"/>
      <bar width="20"/>
      <bar bgcolor="0xDDDDDD"/>
      <text fontsize="12" text="Ignore Layout" align="center"
            options="ignorelayout"/>
      <bar width="20"/>
   </view>
</canvas>
```

Centers text label

Setting the `text` tag's `ignorelayout` option allows the `stableborderlayout` layout to be used, since there are now only three participating views. Adding the `resizestate` tag allows the parent view to be dynamically resized. Because the `text` tag has its `align` attribute set to `center` and is ignored by the layout, it straddles the stretched parent view and always appears in the center as the stretched view is resized (see figure 6.12).

Now that you've learned the fundamentals of layouts and have seen a cross section of layouts addressing a variety of common layout situations, it's time to create your own layouts.



(Initial) **(After resizing)**

Figure 6.12 Setting a text view's `options` attribute to `ignorelayout` causes the "Ignore Layout" text to be ignored by a layout. This allows the text to straddle the parent view and always remain centered.

6.2 Creating custom layouts

Although you can combine Laszlo's supplied layout tags to address many situations, the full power of layouts is unleashed with *custom layouts*. In this section, we'll show how to do this by extending the LzLayout class, using the aircraft formation model.

6.2.1 Extending the LzLayout class

Layouts extend the LzLayout class, which, in turn, extends the node class. Since layouts have no visual characteristics; their only purpose is to manipulate sibling views to maintain a relationship among them. The layout class includes an update method, which contains the algorithm to perform the layout action. Although this is an internal Laszlo method, it is available to be overridden. This update method is initially executed upon application startup and later will be re-executed whenever its subviews have their layout axis or visibility attribute values changed. Table 6.3 lists a layout's methods, which manipulate the subviews.

Table 6.3 Layout methods

Name	Description
`addSubview(view)`	Called when a new subview is added to the layout
`ignore(s)`	Called when a subview is to be ignored by the layout
`lock()`	Locks the layout from processing updates
`releaseLayout()`	Removes the layout from the view and unregisters the delegates that the layout uses
`removeSubview(sd)`	Called when a subview is removed from the layout
`unlock()`	Unlocks the layout after update completes
`update()`	Contains the algorithm for a layout

Since we've already seen how to produce an aircraft formation using offsets and constraints, we'll replicate the results of that example to introduce custom layouts. Later, we'll expand the capabilities of our layout to include dynamic reconfiguration.

6.2.2 Building an aircraft formation layout

To build our formation layout, we'll start with a parent view, main, containing five sibling jet instances. The parent's initial position is at coordinates (400, 400). With layouts, there is no longer a lead aircraft controlling the movements of its following aircraft. Instead, the Laszlo system controls the layout formation and treats each jet equally. This is a more flexible approach, since the individual jets don't need to contain any attribute values to perform layout. Listing 6.10 contains our custom layout.

Listing 6.10 The aircraft formation model with a custom layout

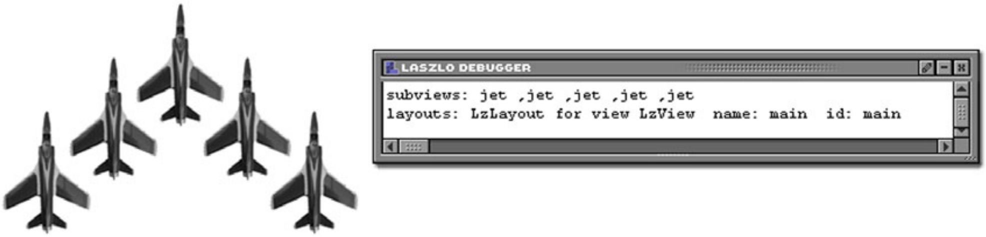
```
<canvas debug="true">
   <resource name="jet" src="F18_Hornet.png"/>
   <class name="jet" resource="jet"/>
   <view name="main" x="400" y="400">
      <layout name="jet_layout">
         <attribute name="xspacing" value="50" type="number"/>
         <attribute name="yspacing" value="40" type="number"/>
         <method name="update">
            <![CDATA[
            for (var i = 0, row = 0; i < subviews.length; i++) {
               if (i % 2)
                  subviews[i].setAttribute('x', -(row * this.xspacing));
               else
                  subviews[i].setAttribute('x', row * this.xspacing);
               subviews[i].setAttribute('y', row * this.yspacing);
               if (i % 2 == 0) row++;
            }
            ]]>
         </method>
      </layout>
      <jet/>
      <jet/>
      <jet/>
      <jet/>
      <jet/>
   </view>
   <handler name="oninit">
       Debug.write("subviews: " + main.subviews);
       Debug.write("layouts: " + main.layouts);
   </handler>
</canvas>
```

Completes row

Specifies custom layout

Specifies five jets

The most important part of a layout is its `update` method, since it contains the layout algorithm. The spacing for the jets is contained in two attributes, `xspacing` and `yspacing`, which are set to 50 and 40 pixels, respectively. The jet formation has an inverted "vee" shape, so logic is needed to determine when the next row is reached. Because there are two jets per row, the first is offset by a negative value and the second by a positive value from the center axis. The result is shown in figure 6.13.



```
subviews: jet ,jet ,jet ,jet ,jet
layouts: LzLayout for view LzView  name: main  id: main
```

Figure 6.13 The five jet instances are contained within the subviews array of the parent view. The attached layout for this parent view is shown in the debug window.

With this custom layout, we can enlarge the formation by simply adding jet instances. Using a layout results in the simplest calling procedure, because there is no need to provide names, constraints, or offsets to any of the jet instances.

Notice that, although a `layout` method is defined, our code contains no calls to it. The parent `main` view automatically calls the layout's `update` method during the parent's initialization. In the next section, we'll see a situation that takes advantage of the simple notation for view-based objects that participate in a layout.

6.3 Laying out forms

One of the most common tasks in web application design is creating a form to gather information from a user. To be as efficient as possible, we want to only specify a set of labels and text fields and have a layout tag create a form with justified labels and text input fields. We want our input fields to be arranged in label-field pairs like this:

```
<canvas>
   <include href="incubator/formlayout.lzx"/>
   <text>name</text>
   <edittext>Bill Higgins</edittext>
   <formlayout/>
</canvas>
```

Although Laszlo supplies a `formlayout` tag under its incubator program, it's useful to know how to create a customized version to handle formatting tweaks for specialized components.

6.3.1 Labeled input fields

We'll define our `formlayout` class to extend the `layout` class and use its `update` method to contain our form algorithm. Our algorithm needs to distinguish between labels and fields to perform the correct field placement. To simplify things, we'll set a fixed label length of 100, default to right justification, and have a fixed spacing of 5 pixels between labels and input fields. Listing 6.11 shows our `formlayout` class.

Listing 6.11 Defining a form layout (formlayout.lzx)

```
<library>
   <class name="formlayout" extends="layout">
      <attribute name="labelwidth" value="100"/>
      <attribute name="spacing" value="10"/>
      <method name="update">
         <![CDATA[
         var curry = 0;
         for (var i = 0; i < subviews.length; i++) {
            var sv = subviews[i];
            if (i%2 != 1) {
               sv.setX(100 - (sv.getTextWidth()+5));
               sv.setY(curry+2); }
            else {
               sv.setX(this.labelwidth);
               sv.setY(curry);
               curry+=sv.height+spacing;
            }
         }
         ]]>
      </method>
   </class>
</library>
```

❶ Controls label length

❷ Controls vertical spacing

❸ Determines y placement

❹ Tests for label or input field

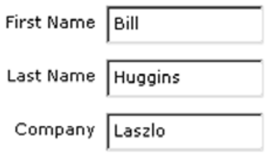
❺ Right-justifies text label with spacer

❻ Updates y placement for next subview

The attribute `labelwidth` specifies the maximum length of a label ❶. Another attribute, `spacing`, controls the vertical spacing between input fields ❷. The local variable `curry` in the `update` method keeps track of the y placement on the screen ❸. The test at ❹ checks if we are working with a label or input field. At ❺ we right-justify the text label and add a spacer. At ❻ we update the y placement for the next subview. The y placement is set to the subview's height plus the value of the spacer.

We're now ready to test our `formlayout` class with a short application using a sampling of label-field pairings:

```
<canvas>
   <include href="formlayout.lzx"/>

   <text>First Name</text><edittext>Bill</edittext>
   <text>Last Name</text><edittext>Huggins</edittext>
   <text>Company</text><edittext>Laszlo</edittext>
   <formlayout/>
</canvas>
```



First Name Bill

Last Name Huggins

Company Laszlo

Figure 6.14
The custom layout formlayout generates a list of left-justified labeled input fields holding a default value.

This produces the output shown in figure 6.14.

This test seems to work very well. So let's now test our new class against a wider assortment of input objects. Since the Laszlo Market's Checkout window uses the `edittext`, `combobox`, and `checkbox` objects, let's incorporate them into the `formlayout`.

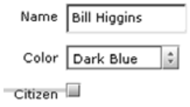
6.3.2 Getting to know formlayout

As a first attempt, let's just try an enhanced list of input objects and see what results:

```
<canvas>
   <include href="formlayout.lzx"/>

   <text>Name</text><edittext>Bill Higgins</edittext>
   <text>Color</text>
   <combobox>
      <textlistitem text="Dark Blue"  value="0x000055"/>
      <textlistitem text="Turquoise"  value="0x66dddd"/>
      <textlistitem text="Light Blue" value="0xaaddff"/>
   </combobox>
   <text>Backups</text><checkbox/>
   <formlayout/>
</canvas>
```

This produces the display shown in figure 6.15.

Although the first two labeled input fields are correctly aligned and right-justified, there's a text alignment problem with the checkbox. Fixing this requires that we identify the object type of the input field, so that adjustments only occur for that object type.



Name Bill Higgins

Color Dark Blue

Citizen □

Figure 6.15 Trying the `formlayout` class with a variety of input fields without further development, output is right-justified correctly but the label for the checkbox isn't aligned correctly. The gray line indicates where the checkbox's label should be aligned.

6.3.3 Identifying class type with instanceof

The `instanceof` function provides JavaScript methods with an easy way to determine the type of a declarative tag. This allows JavaScript code to distinguish between tag types when applying tag-specific formatting. The call is of the form

```
o instanceof c
```

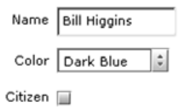
and returns true if o is an instance of the class c. An object matches its own class and any of its parent classes, and it works for both JavaScript and LZX classes. This function is also useful to determine if two objects share a common superclass.

Since a text object is an instance of a JavaScript class, its class instance is LzText. So the text tag would match the LzText, LzView, and LzNode classes. Since the checkbox is an LZX class, we need to check for a checkbox match. We can now modify the update method to check for the checkbox input field and increment the value of its y attribute by 5 pixels to align it with the text label:

```
<method name="update">
   <![CDATA[
   var curry=0;

   for (var i = 0; i < subviews.length; i++) {
      var sv = subviews[i];
      if (i%2 != 1) {
         sv.setX(100 - (sv.getTextWidth()+5));
         sv.setY(curry+2); }
      else {
         sv.setX(this.labelwidth);
         if (sv instanceof checkbox) {
            sv.setY(curry+5); }
         else sv.setY(curry);
         curry+=sv.height+spacing;
      }
   }
   ]]>
</method>
```

Aligns checkbox with label

This produces the output shown in figure 6.16.



Name Bill Higgins
Color Dark Blue
Citizen □

Figure 6.16 The checkbox is moved down to align with the Citizen label.

We now have a general-purpose formlayout tag suitable for any form consisting of a sequence of labeled input fields. Since a layout is used, there is no need to manually set coordinate attributes for the individual fields. On a long, complex form, this can save a lot of time, since a simplelayout tag can be used to lay out multiple formlayout tags along the x-axis.

The major capability missing from our formlayout tag is field and form validation. This is one of the major topics in the next chapter, which deals with Laszlo components.

6.4 Summary

Our goal in this chapter was not only to describe the operation of layouts, but also to provide a rationale for using them. Laszlo supplies both a finished set of layout tools for common situations and a low-level API for creating custom layouts.

A layout is applied to a parent view containing a set of sibling child views. The subviews array in every `LzView` forms the basis for layouts. Every layout contains an `update` method, which iterates through the parent view's subviews array to apply its algorithm to each subview, laying out the child views in a specified pattern.

Laszlo provides a comprehensive assortment of layout tags to address a wide range of common layout situations. These range from the `simplelayout` tag and its variations to various stretchable tags used for creating variable-length borders. Since layouts are implemented as algorithms, layout modifiers can be applied to layouts to achieve compound results.

To demonstrate the basics, we revised the aircraft formation model with layouts. In this way, the lead jet does not determine the base location for the formation; instead, all the jets are treated equally. One advantage is the fact that coordinate attributes are not needed for each jet. As a result, a jet is not tied to a specific screen location, allowing the parent to be moved anywhere on the screen.

Finally, we created a simple `formlayout` custom layout to manage a sequence of labeled input forms. This layout supports simple business forms and is useful in the Checkout window of the Laszlo Market. In the next chapter, we'll extend the `formlayout` with field validation.

7 Introducing Laszlo components

This chapter covers

- Learning component basics
- Building screens with components
- Controlling field placement
- Validating input fields

I would like to see components become a dignified branch of software engineering.

—M. D. McIlroy,
Software Engineering Conference,[1]
Germany, 1968

Reusable components have been a goal of software engineering and application development for decades. A Laszlo component is a customizable user interface widget, written in LZX, that delivers a level of rich behavior previously only seen in desktop applications. Laszlo's component library supplies a comprehensive set of user input controls: radio buttons, scrollbars, and check boxes to build user interfaces with a professional appearance. The appearance of components is important enough that a platform's value is generally judged on the strength of its component library. Laszlo's component library is comparable in breadth and quality to most commercial offerings.

In addition, Laszlo's components are highly customizable, so an application-specific appearance can be easily created. While standard component features such as padding, text margins, and shadow length can be modified through attribute settings, since components are written in LZX and supplied with the Laszlo distribution, we also have the ability to completely alter the behavior or appearance of a component. In chapter 14, we'll explore creating custom components.

There is a large collection of components, so rather than attempting to cover them all, we'll examine the common high-level behavior that pertains to all components. This common behavior governs how components interface to users through the keyboard and mouse. Additionally, it also specifies how components send events when a user selection has been made. Because we believe the best way to learn how to use components is within an application context, we've designed checkout screen using a wide assortment of the most popular components. An additional advantage of using an application context is that field and form validation can be performed on these components. After all, what use are components, no matter how attractive, if they don't protect your application from accepting invalid data values?

7.1 Base component classes

The basic properties for Laszlo components are contained in two classes: `basecomponent`, which provides common properties for all components, and `basevaluecomponent`, which adds the ability to store data. While there is almost

[1] This conference represents the first published use of the term *software engineering*. As such, this quote likely represents the first use of the term *components* in the context of software engineering.

no need to work directly with these base abstract classes, it is necessary to understand the base functionality they supply to higher-level component classes. These base properties, shown in table 7.1 and later in table 7.3, define how input and output is handled by components.

Table 7.1 The `basecomponent` attributes

Name	Data Type	Tag or Script	Attribute Type	Default	Description
`doesenter`	`boolean`	Both	Setter	`false`	Enables the component, if it has focus, to be called with `doEnterDown`—see table 7.2
`enabled`	`boolean`	Both	Setter	`true`	Enables component responses to user events
`hasdefault`	`boolean`	Both	Read-only	`false`	Determines whether or not the component has default focus
`isdefault`	`boolean`	Both	Setter	`null`	Gives the component default focus if it is nearest to the focused view and that view does not have its `doesenter` attribute set to true
`text`	`string`	Both	Setter	`null`	The label displayed on the component

The `text` attribute contains a character string to label a component. But this label can only be used within a component. Many components won't have available space to display it. For instance, text input fields require an external label since the input field is used to contain text. But a window can use this `text` attribute to create a label within its border.

The other attributes control how users interact with components through keyboard and mouse input. These attributes supplement the base attributes inherited from the `LzView` class. For example, when several components are displayed, these attributes determine which component receives input. They also define when and how these components respond to keyboard commands. In the upcoming sections, we'll demonstrate the use of these attributes.

7.1.1 Controlling focus

A component has focus when it is designated as the component to receive keyboard input. As figure 7.1 shows, Laszlo represents focus as a set of corner brackets that slowly fade away. The Tab key is used to transfer focus among a set of eligible components.

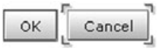


Figure 7.1 Focus is indicated through a set of corner brackets that slowly fade away.

Components can opt out of receiving focus by resetting their focusable attribute to false. Some browsers, such as Firefox, also use focus internally to control its operation through keyboard shortcuts. They require the mouse button to be clicked within the browser screen to switch focus from the browser to the application. Once a component has attained focus, pre- and postprocessing operating features become available to it. Let's first look at how a component with focus interacts with keyboard input.

Capturing keyboard input

After a component gains focus, it can perform the pre- and postprocessing operations listed in table 7.2 by setting its doesenter attribute. The doEnterDown and doEnterUp methods are executed whenever the Enter key is pressed and released. In addition, some components, such as the button, also support the spacebar, allowing the doSpaceDown and doSpaceUp methods to be invoked. Components that receive text input, such as input fields, can't support spacebar-initiated processing, because they need to receive space characters.

Table 7.2 The basecomponent methods

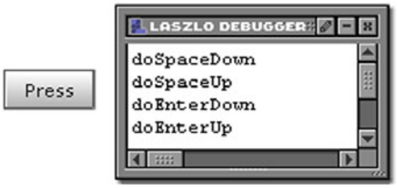
Name	Description
doEnterDown()	Called if the component has focus, doesenter is true, and the Enter key is pressed
doEnterUp()	Called if the component has focus, doesenter is true, and the Enter key is released
doSpaceDown()	Called if the component has focus and the spacebar is pressed
doSpaceUp()	Called if the component has focus and the spacebar is released

Listing 7.1 uses the button component to provide a demonstration of both Enter- and spacebar-initiated processing.

Listing 7.1 Enter- and spacebar-initiated processing

```
<canvas debug="true">
   <button text="Press" doesenter="true">
      <method name="doEnterDown">
         Debug.write("doEnterDown");
      </method>
      <method name="doEnterUp">
         Debug.write("doEnterUp");
      </method>
      <method name="doSpaceDown">
         Debug.write("doSpaceDown");
```

```
        </method>
        <method name="doSpaceUp">
           Debug.write("doSpaceUp");
        </method>
        <handler name="onclick">
           Debug.write("onclick");
        </handler>
     </button>
  </canvas>
```



```
doSpaceDown
doSpaceUp
doEnterDown
doEnterUp
```

Figure 7.2 When the `doesenter` attribute is set, pressing and releasing the Enter key or spacebar initiates the appropriate default processing for that key. Pressing the button shown does not cause any of these methods to be called.

Figure 7.2 shows how pressing and releasing the spacebar and then the Enter key causes a series of calls to the `doSpaceDown`, `doSpaceUp`, `doEnterDown`, and `doEnterUp` methods.

Clicking the Press button only sends an `onclick` event to the `onclick` handler and doesn't result in a call to any of these other methods.

So now that we've established how components use focus to attain keyboard input, we need to set up exceptions to this general rule. For example, most windows are governed by an OK button that must always receive keyboard input. This allows a set of selections to be accepted by a window simply by hitting the Enter key or spacebar. So we need to set up this button to receive keyboard input by default.

Setting keyboard defaults

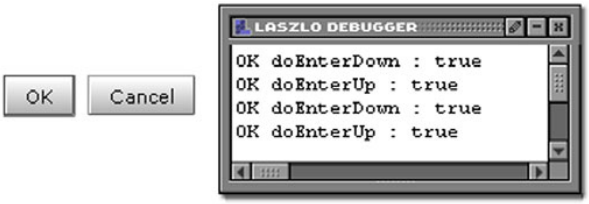
The `isdefault` attribute is settable and overrides focus to establish a particular component member as the master or default control. There is also a read-only `hasdefault` attribute that tests true for a component designated as the default. Finally, if any of the components has its `doesenter` attribute set, then the `isdefault` setting can't override it when it has focus. In listing 7.2, we set the OK button as the default so that it responds to the Enter key and spacebar as well as being clicked.

Listing 7.2 Establishing the OK button as the default

```
  <canvas debug="true">
     <button text="OK" isdefault="true">
        <method name="doEnterDown">
           Debug.write("OK doEnterDown : " + this.hasdefault);
        </method>
        <method name="doEnterUp">
           Debug.write("OK doEnterUp : " + this.hasdefault);
```

```
      </method>
      <handler name="onclick">
         Debug.write("OK Clicked");
      </handler>
   </button>
   <button text="Cancel" x="60" doesenter="false"/>
</canvas>
```

In this example, we want to ensure that pressing the Enter key always results in execution of the OK button's `doEnterDown` and `doEnterUp` methods, independent of which button currently has focus. This is done by setting the Cancel button's `doesenter` attribute to false. This provides the familiar default processing functionality found in most dialog windows. Figure 7.3 shows the debug output.



```
OK doEnterDown : true
OK doEnterUp : true
OK doEnterDown : true
OK doEnterUp : true
```

Figure 7.3
The `isdefault` attribute is set for the OK button, so that its `doEnterDown` and `doEnterUp` are always triggered when the Enter key is pressed and released.

Let's next see how some components can store input values. The `basevaluecomponent` class is inherited by component classes that store data.

7.1.2 Working with data components

The `basevaluecomponent` class has only a single attribute, `value`, shown in table 7.3, and a single method, `getValue`, shown in table 7.4. The `getValue` method is used by all components to retrieve a value. Component classes can override this `getValue` method to add customized data handling.

Table 7.3 The `basevaluecomponent` attribute

Name	Data Type	Tag or Script	Attribute Type	Default	Description
`value`	Any	Both	Setter	null	The value represented by the component

But there is no common method to store a value and each component class must define its own `setValue` method. For example, the check box component uses a `setValue` method to store its value while the combobox component uses a `setText` method to store a text string. However, the `value` attribute is used by all components to store their data.

Table 7.4 The `basevaluecomponent` method

Name	Description
`getValue()`	Returns the value represented by the component

Listing 7.3 shows how these methods can be used to retrieve an input string, convert it to uppercase, and then redisplay it.

Listing 7.3 Converting an input string to uppercase

```
<canvas>
    <edittext width="80" doesenter="true">
        <method name="doEnterDown">
            var value = this.getText();
            this.setText(value.toUpperCase(value));
        </method>
    </edittext>
</canvas>
```

Now that you're familiar with the base component features, we can begin to use these components to develop the various screens for our Checkout window.

7.2 Building a multipage window

The Checkout window contains enough information to require multiple pages. Additionally, each page needs to be subdivided into two panes. Each pane consists of input fields that need input validators to immediately notify users of invalid input. These validators need to perform both field and form validation. Form validation requires that information be consistent across the multiple pages.

Once again, we start with hand-scribbled wireframes to begin the design of the Checkout window shown in figure 7.4, where we've decided to implement each page as a tabelement component.

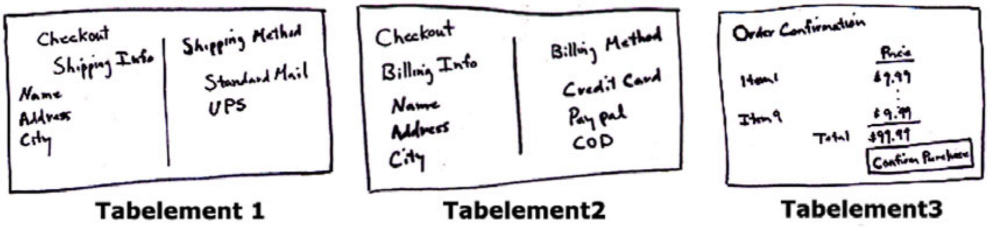


Figure 7.4 This low-fidelity wireframe provides a general overview of the tabelements in the tabslider implementing the Checkout window.

The Checkout window's tabslider holds three tabelements: Shipping Information, Billing Information, and Order Confirmation. The shipping and billing tabelements are divided into address and method panes. The address pane is almost identical for the shipping and billing tabelements. Now, we'll design and develop the panes for these tabelements.

The low-fidelity wireframe is further developed to create the high-fidelity wireframe shown in figure 7.5. Each of the panes is indicated by a light gray square containing the layout of its labeled input fields. There is a 10-pixel border surrounding the two panes and the window boundary, and a 20-pixel separator between the panes. Within the panes is a 5-pixel margin along the top and left sides.

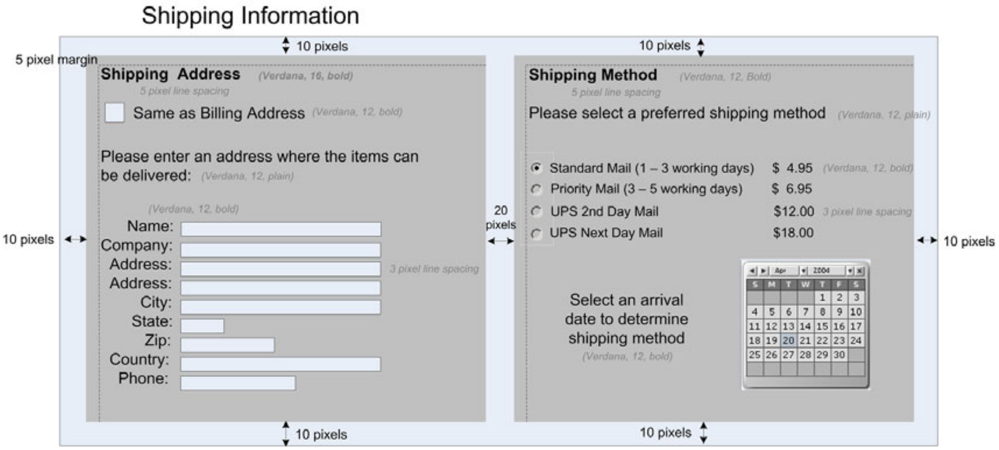


Figure 7.5 The high-fidelity wireframe of the Shipping Information page consists of two light gray squares, each containing a highly detailed specification for the layout of the elements. Each gray square contains a 5-pixel margin on the top and left.

We'll start with the general layout of the panes before working with their interior contents.

7.2.1 Coding the Shipping Information page

Separate shipping information, billing information, and order confirmation classes are created to represent each of the tabslider's tabelements. We want to use all the available space within each tabelement, so their width and height settings are set to `100%`. Since font settings propagate from parent to child, this is a good place to set the following default font specifications: font, `Verdana`, font size, `12`, and font style, `bold`. All subsequent elements contained within these classes will inherit these default settings, and don't need to be specified individually:

```
<class name="shippingInfo" width="100%" height="100%"
       fontsize="12" fontstyle="bold" font="Verdana"/>
<class name="billingInfo" width="100%" height="100%"
       fontsize="12" fontstyle="bold" font="Verdana"/>
<class name="orderConfirm" width="100%" height="100%"
       fontsize="12" fontstyle="bold" font="Verdana"/>
```

These attribute values serve as default values that can be overridden by individual instances. We'll adopt the practice of specifying default class attribute settings within our defined classes to help prevent the calling sections from becoming cluttered with numerous attribute settings.

Although global IDs should be used sparingly, creating a set of global IDs, `shiptab`, `billtab`, and `ordertab`, provides a convenient way to access the contents of each tabelement. Listing 7.4 shows the declaration of our tabslider with its tabelements.

Listing 7.4 Declaring a tabslider with its tabelements for the checkout screen

```
 <tabslider height="100%" width="100%" spacing="2" slideduration="300">
   <tabelement text="Shipping Information">
      <shippingInfo id="shiptab"/>
   </tabelement>
   <tabelement text="Billing Information">
      <billinginfo id="billtab"/>
   </tabelement>
   <tabelement id="ordertab" text="Order Confirmation"/>
</tabslider>
```

The next step is creating the `shippingInfo` and `billingInfo` classes to populate the Shipping and Billing Information tabelements.

Configuring the shipping and billing layouts

Since each of these tabelements consists of address and method panes, we'll define a separate class to represent each of them:

```
<class name="shippingAddress" bgcolor="0xCCCCCC"/>
<class name="shippingMethod"  bgcolor="0xCCCCCC"/>
<class name="billingAddress"  bgcolor="0xCCCCCC"/>
<class name="billingMethod"   bgcolor="0xCCCCCC"/>
```

Each tabelement contains a default 5-pixel margin, which we'll increase to 10 pixels by setting the `x` and `y` attributes for each instance to `5`. To accommodate this 10-pixel margin, we set the `height` attribute to the parent's height minus 10. Now, if the `shippingInfo` instance gets resized, the 10-pixel margin won't change. Because this relationship occurs within a class definition, the parent specification is changed to `classroot`. The `shippingAddress` instance now has a 10-pixel margin on its left, top, and bottom sides. We'll next begin working on its width.

To accommodate the 20-pixel spacer and the 5-pixel left and right margins, we subtract 15 pixels from the width of each gray pane. Both gray panes have their width set to the parent's width divided by 2 minus 15 pixels. Finally, the right gray pane has its `x` attribute offset to the parent's width divided by 2 plus 10 pixels, since this is half the spacer size. Here are the results of our efforts so far:

```
<class name="shippingInfo" fontsize="12" fontstyle="bold" font="Verdana">
   <shippingAddress x="5" y="5"
                     width="${classroot.width/2-15}"
                     height="${classroot.height-10}"
                     bgcolor="0xCCCCCCC"/>
   <shippingMethod x="${classroot.width/2+10}" y="5"
                    width="${classroot.width/2-15}"
                    height="${classroot.height-10}"
                    bgcolor="0xCCCCCC"/>
</class>
```

Displays left gray box

Displays right gray box

Since these gray squares are to be reused in the Billing Information screen, we'll create a `graybox` class to represent them. Next we'll look at creating the 10-pixel margin within each gray square, which serves as an introduction to the issue of placement.

7.2.2 Controlling placement issues

Each gray box contains a form consisting of a header, text messages, and labeled input fields. According to the layout specifications in figure 7.6, each gray box has a 10-pixel margin along its top and left sides surrounding this form. Each pane can be viewed as consisting of two containers: an outer view with a gray background and an inner view offset by 10 pixels to contain the form. A first attempt at creating the `graybox` class might look like this:

```
<class name="graybox" bgcolor="0xCCCCCCC">
   <view name="content" x="10" y="10"
         width="${parent.width-20}"
         height="${parent.height-20}"/>
</class>
```

The `x` and `y` attributes are set to create a fixed margin of 10 pixels, and its width and height are shortened to accommodate this margin. Finally, a text object message is added within a `graybox` instance, and it also needs to have a margin set:

```
<canvas debug="true">
   <include href="library.lzx"/>
   <graybox name="shipping"
            width="${parent.width/2-20}"
            height="${parent.height-20}">
      <text text="message">
         <handler name="oninit">
            Debug.write("parent : ", this.parent);
            Debug.write("immediateparent : ", this.immediateparent);
         </handler>
      </text>
   </graybox>
</canvas>
```

Unfortunately, this doesn't produce the expected results. Looking at figure 7.6, we see that the message is displayed with no margin; the right image shows the desired results.

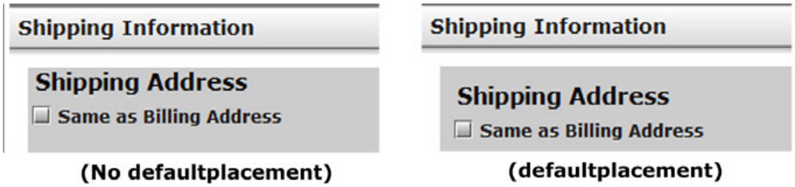


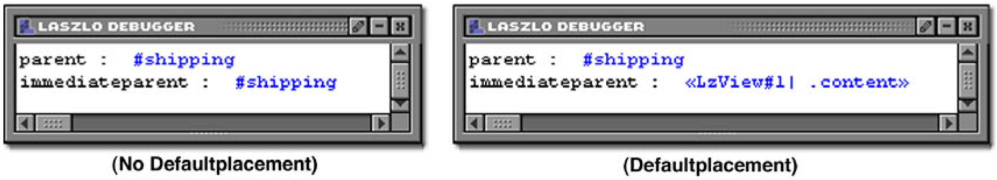
Figure 7.6 The display on the left shows the result of defining a class without specifying a `defaultplacement` attribute. Any view-based objects contained by this class instance are displayed at the wrong level of the node hierarchy.

Before diving into this issue, let's step back and revisit some of the key view attributes and methods concerning placement. Every child node has two parents: a `parent` attribute specifying its read-only parent within an application's node hierarchy, and its settable `immediateparent` attribute, which can display a node anywhere within the node hierarchy.

The message isn't placed correctly because the internal structure of a class is not visible to outside views. By default, whenever an object—our text message—is placed inside a class as a child node, it appears as a top-level instance within the class. But this isn't where we need the text message placed. It needs to be placed inside the inner view named `content`, and not within the `graybox` instance named `shipping`.

The debugger output in figure 7.7 provides a deeper look at the problem. For the text object to be correctly aligned, it must be a child of the view named `content`. But the values for its `parent` and `immediateparent` attributes are both `shipping`.

The `defaultplacement` attribute was created for just this type of situation; it causes the `immediateparent` attribute to be set to the value of the `defaultplacement`. Now all subsequent views placed within the `graybox` parent will be correctly aligned within `content`.



```
parent :  #shipping
immediateparent :  #shipping
```

(No Defaultplacement)

```
parent :  #shipping
immediateparent :  «LzView#1| .content»
```

(Defaultplacement)

Figure 7.7 This debugger output shows that the text message is the child of the class instance shipping and not of the view content. Both `parent` and `immediateparent` have the value `shipping`.

```
<class name="graybox" defaultplacement="content">
   <attribute name="bgcolor" value="0xCCCCCC" type="color"/>
   <view name="content" x="10" y="10"
         width="${parent.width-20}"
         height="${parent.height-20}" bgcolor="0xBBBBBB"/>
</class>
```

The `defaultplacement` attribute affects placement for all class instances, while the `placement` attribute changes the placement only for a particular instance. So setting the placement only for the shipping `graybox` instance would look like this:

```
<graybox name="shipping" placement="content"/>
```

Although the `placement` attribute provides another way to specify placement, it can't be used to override the `defaultplacement` setting. The `defaultplacement` setting always takes precedence over individual `placement` settings.

Now that we've resolved the general layout issues, we are ready to proceed with the two panes for the Shipping Information tabelement. We'll start with the Shipping Address pane.

7.2.3 Creating the Shipping Address pane

We can handle the configuration of the Shipping Address pane with two nested layouts. The top-level organization of the screen is a series of views, indicated by the dashed boxes in figure 7.8, stacked atop one another. This is easily represented with a `simplelayout` tag specifying 5-pixel row spacing.

The last view contains a form, which requires a nested `formlayout`; see listing 7.5. This combination provides a neat solution for handling the row-spacing requirements for this pane.

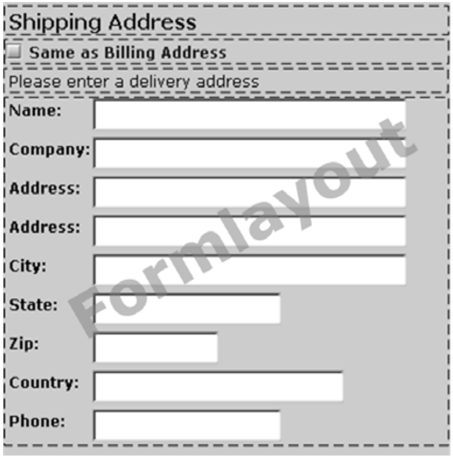


Figure 7.8
A simplelayout is used to perform the higher-level layout of the major view-based objects. The simplelayout provides a consistent spacing of 5 pixels between each element. All the input fields within the last view are laid out with a formlayout.

Listing 7.5 Using a combination of `simplelayout` and `formlayout` for the Shipping Address pane

```
<class name="shippingAddress">
   <graybox width="${classroot.width/2-15}"
            height="${classroot.height-5}">
      <simplelayout axis="y" spacing="5"/>
      <text fontsize="16">Shipping Address</text>
      <checkbox text="Same as Billing Address"/>
      <text multiline="true" width="100%">
            Please enter an address where the items can
            be shipped to
      </text>
      <view width="100%"/>
         <formlayout align="right" spacing="3"/>
         <text text="Name:"/><edittext width="250"/>
         <text text="Company:"/><edittext width="250"/>
         <text text="Address:"/><edittext width="250"/>
         <text text="Address:"/><edittext width="250"/>
         <text text="City:"/><edittext width="250"/>
         <text text="State:"/><edittext width="150"/>
         <text text="Zip:"/><edittext width="100"/>
         <text text="Country:"/><edittext width="200"/>
         <text text="Phone:"/><edittext width="150"/>
      </view>
   </graybox>
</class>
```

Provides 5-pixel spacing between rows

Overrides default font setting

Sets multiple lines of fixed width

Provides right alignment and 3-pixel spacing

Since everything is handled with layouts, this allows us to easily add or remove elements without impacting the overall configuration.

The last missing piece required by this pane is field validation. We need validation at both the field and the form levels. Field validation ensures that no fields contain invalid data, while form validation ensures that the fields are collectively consistent. For example, the state and zip code fields need to be checked to ensure that the zip code belongs within the state. Invalid data needs to be immediately detected and reported to the user in a nonintrusive way. In the next section, you'll learn how validators satisfy each of these goals.

7.3 Validating input fields

Although Flash doesn't natively support regular expressions to check input values, validator tags provide a convenient solution for validating user input. A field input value is checked against a set of rules to determine its validity and a green(OK) or red (Error) icon is displayed along with a relative error message. The rules are written in JavaScript, but the range and requirements of input data types is small enough to be easily handled.

All field validators are derived from the `basevalidator` class. Each validator has a `doValidation` method, containing the validation rules for each input field, which is called for each input character. Initially, each validated field contains a red error icon that changes to green when the input value satisfies the validation criteria. This allows character classes (all numbers or letters) to be checked as well as formatted strings (an email address consisting of name@company.com). Each validator contains a set of attributes with specific error messages available to be overridden to provide customized error messages.

Validator tags must be contained within a `validatingForm` tag. The `validatingForm` tag contains an `errorcount` attribute, which is decremented whenever an input field satisfies its validation criteria. When the `errorcount` value reaches zero, an action, such as enabling a submit button, can be performed. The validator libraries are included like this:

```
<include href="incubator/validators"/>
```

and four validator tags—`stringvalidator`, `datevalidator`, `numbervalidator`, and `emailvalidator`—are included in the distribution. All source code for the current suite of validators is bundled with the Laszlo distribution and can be found in this directory:

```
$LPS_HOME/lps/components/incubators/validators
```

Validator tags are currently part of the Laszlo "incubator" program, but have become popular enough to attain widespread use. They were originally contributed by Togawa Manabu of the Laszlo Japan users group. Since new validator types are easy to write, the current list of supported data types should soon grow into a complete library covering all the major input data types. Later we'll cover how to create a new validator tag.

7.3.1 Using validators

Validator tags must wrap an input field to perform validation on it. The required attribute makes it mandatory that input be entered into a field. All error messages are available as attributes, so it's easy to change the default messages. The requiredErrorstring attribute is set here to change the default requiredErrorstring error message:

```
<stringvalidator required="true" requiredErrorstring="Required">
   <edittext width="150"/>
</stringvalidator>
```

Since we are dealing exclusively with text input fields in the Shipping Address pane, we'll create a set of classes to support a succinct notation:

```
<class name="checkString" extends="stringvalidator" width="150"
       required="true" requiredErrorstring="Required">
   <edittext width="${classroot.width}"/>
</class>
<class name="checkNumber" extends="numbervalidator" width="100"
       required="true" requiredErrorstring="Required">
   <edittext width="${classroot.width}"/>
</class>
```

As listing 7.6 shows, the validatingForm performs validation on each of the fields in the form. The validatingForm keeps track of the number of nonvalidated input fields. Every time a field becomes validated, its errorcount attribute is decremented and its event handler is called to check for nonvalidated input fields. When all input fields have been validated, the pane's confirm attribute is set to true to indicate that all its input fields are now valid. If a user subsequently updates a field with an invalid input value, the confirm attribute changes back to false.

Listing 7.6 Using validators to check input data in the Checkout window

```
<validatingForm width="100%">
   <formlayout align="right" spacing="3"/>
   <text text="Name:"/><checkString width="150"/>
   <text text="Company:"/><checkString name="company"/>
   <text text="Address:"/><checkString name="addr1"/>
   <text text="Address:"/><checkString name="addr2"
         required="false"/>
   <text text="City:"/><checkString name="city"/>
   <text text="State:"/><checkString name="state"/>
   <text text="Zip:"/><checkNumber name="zip"/>
   <text text="Country:"/><checkString name="country"/>
```

```
    <text text="Phone:"/><checkNumber name="phone"/>
    <text text="Email:"/><checkNumber name="email"/>
    <handler name="onerrorcount" args="errors">
       if (errors == 0)
          classroot.setAttribute("confirm", true);
       else
          classroot.setAttribute("confirm", false);
    </handler>
</validatingForm>
```

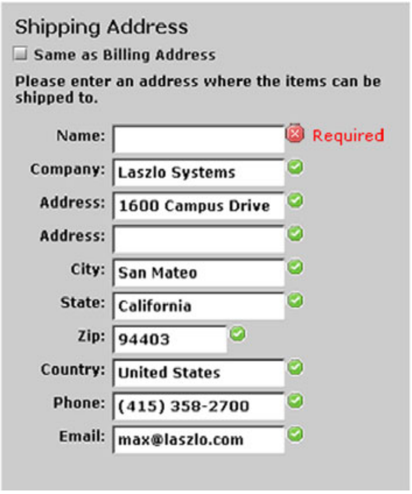
Triggers when field is validated

Signals form has validated

Resets form validation indicator

Figure 7.9 shows that all fields, with the exception of the second address field, are required. Once a required field has been filled, a green checkmark appears next to it, allowing a user to easily verify that all input fields have been correctly completed.

Let's now look at how easy it is create new validator tags to provide more customized data type validation.



Shipping Address

□ Same as Billing Address

Please enter an address where the items can be shipped to.

Name: ☒ Required

Company: Laszlo Systems ✓

Address: 1600 Campus Drive ✓

Address: ✓

City: San Mateo ✓

State: California ✓

Zip: 94403 ✓

Country: United States ✓

Phone: (415) 358-2700 ✓

Email: max@laszlo.com ✓

Figure 7.9 Validator tags provide a visually attractive way to perform field-level validation.

7.3.2 Creating a new validator

In listing 7.6 the `checkNumber` class uses the `numbervalidator` tag to perform validation on zip codes. However, it is a poor match for this job since it only ensures that the input value is numeric and doesn't exceed 99999. Zip code validation should handle both five- and nine-character forms, with the nine-character code containing a dash. The `zipcodevalidator` class, shown in listing 7.7, is called for each input character.

Listing 7.7 Validating five- and nine-character zip codes

```
<class name="zipcodevalidator" extends="basevalidator">
   <attribute name="trim" type="boolean" value="true"/>
   <attribute name="notzipcodeErrorstring" type="string"
              value="Invalid zipcode value"/>

   <method name="doValidation" args="val">
      <![CDATA[
      var valtext = getValueText(val);

      if (required && valtext.length < 1){
```

Trims white space

Is called for every character

Returns entire character string

```
         this.setErrorstring(this.requiredErrorstring);
         return false; }
         var dash = valtext.indexOf("-");
         if (dash > 0) {
            var zips = valtext.split("-");
            if (zips[0].length != 5 || isInt(zips[0])) {
               this.setErrorstring(this.notzipcodeErrorstring);
               return false; }
            if (zips[1].length != 4 || isInt(zips[1])) {
               this.setErrorstring(this.notzipcodeErrorstring);
               return false; } }
         else {
            if (valtext.length != 5 || isInt(valtext)) {
               this.setErrorstring(this.notzipcodeErrorstring);
               return false; } }
      this.setErrorstring("");
      return true;
      ]]>
   </method>

   <method name="isInt" args="value">
      <![CDATA[
      if (!isNaN(value)) return false;
      if (value.toString().indexOf(".") < 0)
         return true;
      else
         return false;
      ]]>
   </method>
</class>
```

Distinguishes between 5- and 9-character codes

Unfortunately, local validators can't be created and this validator class must be moved to $LPS_HOME/lps/components/incubator/validators directory, where it will be available to all Laszlo applications. Additionally, this directory's library.lzx file must be updated to include it.

We can easily include this email validator by adding it to our previously defined set of wrapping tag classes:

```
<class name="checkEmail" extends="emailvalidator" width="100"
       required="true" requiredErrorstring="Required">
   <edittext width="${classroot.width}"/>
</class>
```

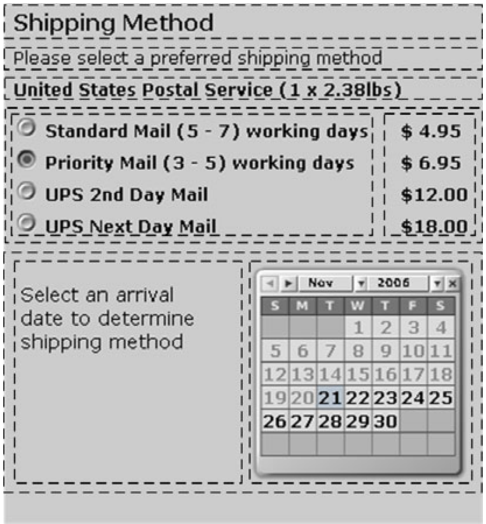
and updating our email input field to use it:

```
<text text="Email:"/><checkEmail name="email"/>
```

This completes the Shipping Address pane. We're ready to move on to the Shipping Method pane.

7.3.3 Creating the Shipping Method pane

Although the Shipping Method wireframe, shown in figure 7.10, contains a number of complicated components, its layout is relatively straightforward. This pane can be configured using a couple of nested `simplelayout` tags.



Shipping Method

Please select a preferred shipping method

United States Postal Service (1 x 2.38lbs)

○ Standard Mail (5 - 7) working days $ 4.95

● Priority Mail (3 - 5) working days $ 6.95

○ UPS 2nd Day Mail $12.00

○ UPS Next Day Mail $18.00

Select an arrival date to determine shipping method

Figure 7.10
The Shipping Method pane can be subdivided into a series of nested `simplelayout` tags. It consists of five view-based objects aligned along the x-axis, where the fourth and fifth objects can be further broken down into a series of views aligned along the y-axis.

Listing 7.8 shows how the nested series of layouts shown in figure 7.10 is implemented through a set of `simplelayout` tags. A `radiogroup` is a parent container to its radio buttons and can use a `layout` attribute for its radio buttons.

Listing 7.8 The Shipping Method pane

```
<graybox x="${parent.width/2+10}" y="5"
         height="${parent.height-5}" width="${parent.width/2-15}">
   <simplelayout axis="y" spacing="5"/>
   <text fontsize="16">Shipping Method</text>
   <text multiline="true" fontstyle="plain">
         Please select a preferred shipping method
   </text>
   <text> United States Postal Service (1 x 2.38lbs)</text>
   <view width="100%">
      <simplelayout axis="x" spacing="30"/>
      <radiogroup id="shipmethod" layout="axis:y;spacing:3">
         <radiobutton value="1" selected="true"
                      text="Mail (5-7 working days)"/>
         <radiobutton value="2" text="Priority Mail (3-5 working days)"/>
         <radiobutton value="3" text="UPS 2nd Day Mail"/>
         <radiobutton value="4" text="UPS Next Day Mail"/>
      </radiogroup>
```

Spaces shipping prices by 30 pixels

Sets radio button spacing at 3 pixels

```
        <view>
            <simplelayout axis="y" spacing="3"/>
            <text text="$ 4.95"/>
            <text text="$ 6.95"/>
            <text text="$12.00"/>
            <text text="$18.00"/>
        </view>
     </view>
     <view width="100%">
        <simplelayout axis="x" spacing="10"/>
        <text name="instructions" valign="middle"
              width="${parent.width/2}"
              multiline="true" fontstyle="plain" fontsize="14"
              text="Select an arrival date to determine shipping method"/>
        <datepicker startAsIcon="false" id="shipdate"
                    selecteddate="new Date()"/>
     </view>
  </graybox>
```

Sets price row spacing at matching 3 pixels

Spaces datepicker by 10 pixels

To complete the Shipping Method pane, an easy-to-use interface is needed that allows a user to pick an arrival date, which then selects the radio button for the appropriate shipping method. This helps prevent users from accidentally spending too much on shipping. Listing 7.9 shows how this is implemented.

Listing 7.9 Using the datepicker to select an arrival date

```
  <datepicker y="30" startAsIcon="false"
              selecteddate="new Date()">
     <method event="onselecteddate" args="d">
        if (d == null) return;
        var now = new Date();
        var daydiff = d.getDate() - now.getDate() - 1;
        shipmethod.setByDate(daydiff);
     </method>
  </datepicker>
```

❶ **Specifies current date**

❷ **Handles date entry**

❸ **Returns invalid date**

❹ **Calculates lead time**

❺ **Displays target date**

When the datapicker is instantiated ❶, the current date `new Date()` is chosen as its initial value. When a new date is selected through the graphical date calendar ❷, the `onselecteddate` event handler is triggered; if an invalid date is selected, it simply returns ❸. The number of days between the selected and current dates is calculated ❹. The graphical date calendar doesn't accept a preceding date, so there is no concern about negative values. The `setByDate` method for the Shipping Method's radio group is called ❺, with the difference in days as its argument. Listing 7.10 uses this value to set the appropriate shipping range.

Listing 7.10 Setting the mail priority based on date

```
<radiogroup y="30" id="arrivalDate">
   <method name="setByDate" args="v">
      <![CDATA[
      if (v == 0)
         this.setAttribute('value', 4);
      else if (v == 1)
         this.setAttribute('value', 3);
      else if (v < 5)
         this.setAttribute('value', 2);
      else
         this.setAttribute('value', 1);
      ]]>
   </method>
   <radiobutton value="1" selected="true"
                text="Mail (5-7 working days)"/>
   <radiobutton value="2" text="Priority Mail (3-5 working days)"/>
   <radiobutton value="3" text="UPS 2nd Day Mail"/>
   <radiobutton value="4" text="UPS Next Day Mail"/>
</radiogroup>
```

It's now time to implement the Billing Information tabelement.

7.3.4 Implementing the Billing Information page

The wireframe for the Billing Information tabelement also consists of address and method panes, as shown in figure 7.11.

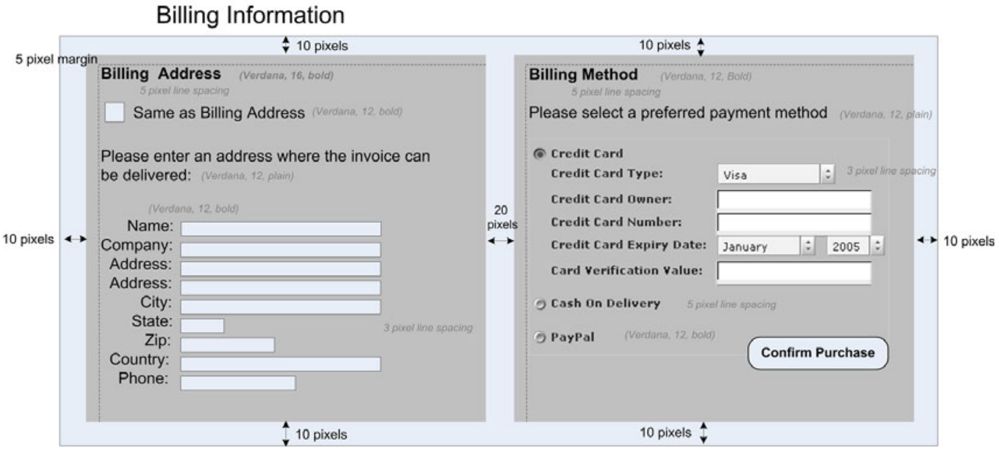


Figure 7.11 The structure of the wireframe for the Billing Information tabelement is similar to the Shipping Information page.

Since the Billing Address and Shipping Address panes are almost identical, except for the Same as Shipping Address check box, it isn't necessary to show its code.

Instead we'll start with the method pane and its combobox input fields.

Creating static comboboxes

A combobox displaying a pop-up list of items for selection can be implemented with either `textlistitem` tags for text labels or `listitem` tags to support images. We'll isolate all the static date information within classes. Later, in chapter 10, we'll show you how local datasets offer a superior solution for containing this type of information. An abridged listing is shown here:

```
<class name="cal_combobox" extends="combobox">
   <textlistitem text="January"   value="1"/>
   <textlistitem text="February"  value="2"/>
   …
   <textlistitem text="November"  value="11"/>
   <textlistitem text="December"  value="12"/>
</class>

<class name="date_combobox" extends="combobox">
   <textlistitem text="2006"   value="2006"/>
   …
   <textlistitem text="2009"   value="2009"/>
   <textlistitem text="2010"   value="2010"/>
</class>
```

The credit card combobox needs to display a credit card icon image along with the name of the credit card company. Since the `listitem` class only contains an image, we'll extend it by adding a `text` attribute:

```
<class name="iconitem" extends="listitem"
       width="100%" height="24" >
   <attribute name="text" value="" type="string"/>
   <text x="20" text="${classroot.text}"/>
</class>
```

We'll add these `resource` tags to our resources.lzx file to supply logical names to the image locations:

```
<resource name="mastercard_logo" src="resources/mastercard_logo.png"/>
<resource name="visa_logo"       src="resources/visa_logo.png"/>
<resource name="amex_logo"       src="resources/amex_logo.png"/>
```

Finally, in listing 7.11, we use `iconitems` to combine these resources with the text labels for each combobox item.

Listing 7.11 Credit card combobox featuring labeled icons

```
<combobox name="creditcard" width="180" editable="false"
          defaulttext=" ">
   <iconitem resource="visa_logo" text="Visa" value="1"/>
```

Ensures combobox is initially blank

```
   <iconitem resource="mastercard_logo" text="MasterCard" value="2"/>
   <iconitem resource="amex_logo" text="American Express" value="3"/>
   <method event="onselect" args="s">
      Debug.write("selection text=" + s.text +
                              " value=" + s.value);
   </method>
</combobox>
```

Contains argument of type listitem

Displays selection's text and value

The combobox's defaulttext attribute must be set to an empty string to create an initial blank display. The editable attribute is reset to false to ensure that this is a read-only display. When a selection is made, the onselect event is sent with an iconitem object as its argument. Figure 7.12 shows the object's text and value attributes.

We're now ready to begin converting our wireframe into LZX using these newly created combobox classes.

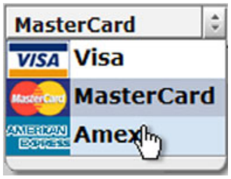


Figure 7.12 A graphical display is used for each of the credit cards listed in the combobox.

7.3.5 Coding the Billing Method wireframe

The spacing among the major elements—titles and radio buttons—has 5-pixel row spacing; the minor elements in a group have 3-pixel row spacing. All spacing is controlled using the spacing attribute from multiple layouts. Listing 7.12 shows the spacing attribute, highlighted in bold, for the nested layouts.

Listing 7.12 The billingMethod class definition

```
<class name="billingMethod" width="100%" height="100%">
   <attribute name="confirm" value="false" type="boolean"/>
   <graybox x="${parent.width/2+5}"
            height="${parent.height-5}"
            width="${parent.width/2-15}">
      <simplelayout axis="y" spacing="5"/>
      <text fontsize="16">Billing Method</text>
      <text multiline="true" width="100%">
          Please select the preferred payment method to use on
          this order
      </text>
      <radiogroup layout="axis:y;spacing:5">
         <radiobutton value="1" selected="true" text="Credit Card">
            <validatingForm width="100%" x="5%">
               <formlayout align="right" spacing="3"/>
               <text text="Type:"/>
               <combobox width="140"
                         editable="false" defaulttext=" ">
```

Controls display of button

Adds 3-pixel spacing for radio buttons

Specifies nested formlayout

```
                  <iconitem resource="visa_logo"
                            text="Visa" value="1"/>
                  <iconitem resource="mastercard_logo"
                            text="MasterCard" value="2"/>
                  <iconitem resource="amex_logo"
                            text="Amex" value="3"/>
               </combobox>
               <text text="Owner:"/><checkString name="owner"/>
               <text text="Number:"/><checkNumber name="number"/>
               <text text="Expiration Month:"/>
               <cal_combobox width="100" editable="false"/>
               <text text="Expiration Year:"/>
               <date_combobox width="80" editable="false"/>
               <text text="Verification:"/>
               <checkNumber name="verification"/>
               <method event="onerrorcount" args="val">
                  <![CDATA[
                  if (val == 0)
                     classroot.setAttribute("confirm", true);
                  else
                     classroot.setAttribute("confirm", false);
                  ]]>
               </method>
            </validatingForm>
         </radiobutton>
         <radiobutton value="2" text="Cash On Delivery"/>
         <radiobutton value="3" text="PayPal"/>
      </radiogroup>
   </graybox>
</class>
```

Marks validation as complete

Marks validation as incomplete

The remaining task is to coordinate the Shipping and Billing Information tabelements. To relieve users of entering identical information, the "Same as … " check box allows one set of address information to be used across both tabelements. But the boxes must be coordinated so that selecting one check box disables the other. This prevents both boxes from being checked with no information entered.

7.3.6 Coordinating multiple pages

When we declared the tabelements within the tabslider, we labeled each with a global ID—`shiptab` for Shipping and `billtab` for Billing—to allow information in either tabelement to be conveniently accessed by the other:

```
<tabelement text="Shipping Information">
   <shippingInfo id="shiptab" … />
</tabelement>
```

```
<tabelement text="Billing Information">
   <billingInfo id="billtab" … />
</tabelement>
```

To implement the “Same as… ” functionality, we’ll add a sameas attribute to each class definition. Each of these attributes can easily be accessed by the other class:

```
<class name="shippingInfo" fontsize="12" fontstyle="bold"
                                         font="Verdana">
   <attribute name="sameas" value="false" type="boolean"/>
   <shippingAddress name="shipAdd"/>
   <shippingMethod name="shipMeth"/>
</class>
<class name="billingInfo" fontsize="12" fontstyle="bold"
                                        font="Verdana">
   <attribute name="sameas" value="false" type="boolean"/>
   <billingAddress name="billAdd"/>
   <billingMethod  name="billMeth"/>
</class>
```

The logic needed in each class is the mirror image of the other. When the Billing Address’s Same as Shipping Address check box is checked, all the Billing Address fields default to the values contained in the Shipping Address fields. Since these fields have already been verified, there is no need to reconfirm them, so a confirm attribute tells the validation tags to ignore these fields. Listing 7.13 shows the code for this.

Listing 7.13 Coordinating the “Same as… ” check boxes in the Billing and Shipping pages

```
<class name="billingAddress" width="100%" height="100%">
   <attribute name="confirm" value="false" type="boolean"/>
   ...
   <checkbox text="Same as Shipping Address"
             enabled="${!shiptab.sameas}">
      <handler name="onclick">
         billtab.setAttribute("sameas", !billtab.sameas);
         classroot.setAttribute("confirm", !classroot.confirm);
      </handler>
   </checkbox>
   ...
</class>

<class name="shippingAddress" width="100%" height="100%">
   <attribute name="confirm" value="false" type="boolean"/>
   ...
   <checkbox text="Same as Billing Address"
             enabled="${!billtab.sameas}">
      <handler name="onclick">
         shiptab.setAttribute("sameas", !shiptab.sameas);
```

Constrains to Shipping setting

Flips current Billing value

Flips current form confirmation

Accesses Billing setting

Accesses Shipping setting

```
            classroot.setAttribute("confirm", !classroot.confirm);
        </handler>
    </checkbox>
    ...
</class>
```

Sets form confirmation

To complete the “Same as … ” feature, the text input fields must be disabled. But the validator tags wrap these text input fields and don’t allow their enabled state to change. One possible solution is to use the state tag to create two sets of tags—a validated set of text input fields, and a simple set of text input fields that can be disabled—and switch between them. But this leads to redundant declarative statements.

A better way is to override the validator’s doValidate method inherited by the checkstring class from its stringvalidator superclass. We’ll add an enabled attribute that is accessible from the edittext object to disable its state. Now when the validator tag’s enabled attribute is reset to false, validation isn’t performed because the input field is disabled. When it is enabled, the stringvalidator superclass’s doValidation method is called to perform its normal validation. Listing 7.14 shows how the doValidate method is overridden.

Listing 7.14 Support routine for validation

```
<class name="checkString" extends="stringvalidator" width="150"
       required="true" requiredErrorstring="Required">
    <attribute name="enabled" value="true" type="boolean"/>
    <method name="doValidation"  args="val">
        if (this.enabled == false) return;
        super.doValidation(val);
    </method>
    <edittext width="${classroot.width}" enabled="${classroot.enabled}"/>
</class>
```

Now the input fields within each tabelement can be easily updated to support a disabled state by adding this constraint:

```
<class name="billingAddress" width="100%" height="100%">
    <validatingForm width="100%">
        <formlayout align="right" spacing="5"/>
            <text text="Name:"/><checkString name="fullname"
                  enabled="${!billtab.sameas}"/>
             ...
        </formlayout>
    </validatingForm>
</class>
and
```

```
<class name="shippingAddress" width="100%" height="100%">
   <validatingForm width="100%">
      <formlayout align="right" spacing="5"/>
         <text text="Name:"/><checkString name="fullname"
               enabled="${!shiptab.sameas}"/>
          ...
      </formlayout>
   </validatingForm>
</class>
```

The result is that all the input fields for either the Shipping or Billing address page are disabled when the other tabelement's "Same as ... " box is checked. Conversely, they are enabled when the check is removed.

Our remaining task is form validation. It leverages all the previous steps to ensure logical consistency across the multiple tabelements.

7.3.7 Form validation

The purpose of form validation is to check for logical consistency across an entire form. To prevent invalid information from entering the system, the Complete Purchase button is not enabled until the validation requirements are satisfied. In our case, form validation consists of ensuring that all the input forms contain validated information. When this condition is met, the `confirm` attribute for each form is set to true.

Using XML escape characters

Although a <![CDATA[section can be used in a JavaScript method to protect ASCII characters that have special meaning to XML, it can't be used in declarative tags. Instead, it's necessary to enclose such characters in an escape sequence. This requires the & character to be expressed as the &amp; escape sequence:

```
<class name="orderConfirm" width="100%" height="100%"
       fontsize="12" fontstyle="bold" font="Verdana">

   <button align="center" valign="middle" text="Complete Purchase"
         enabled="${billtab.billAdd.confirm &amp;&amp;
                    shiptab.shipAdd.confirm &amp;&amp;
                    billtab.billMeth.confirm}"/>
</class>
```

Contains a left gray box

Consequently, if a validated form is updated with an invalid entry, the Complete Purchase button is automatically disabled. This relationship is maintained through a constraint expression between the confirm fields for each of the tabelement pages. You can see this sequence of state changes for the Complete Purchase in figure 7.13.

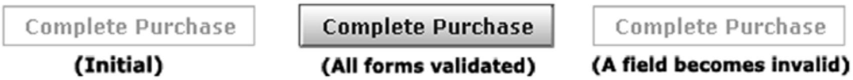


Figure 7.13 The Complete Purchase button is initially disabled. It is enabled when validation is completed. If a validated form is updated with an invalid entry, the button is automatically disabled.

This completes this phase of development for the tabelements of the Checkout window.

7.4 Summary

Laszlo supplies a wide assortment of components offering quick-deployment solutions to many common user interface requirements. Using components allows a developer to rapidly construct graphical user interfaces. These components are based on a common set of base classes providing common behavior. Since there are so many different components available, rather than provide individual examples we have used a wide cross-section of these components within the Laszlo Market. The intent is to not only demonstrate their use, but to show how components can be used collectively to construct larger user interfaces.

In the Laszlo Market, components are used to construct the Checkout window. This window contains too much information to fit in a single screen, so it is divided into a set of pages. These pages are implemented as tabelements within a tabslider. Each page corresponds to a Shipping Information, Billing Information, or Order Confirmation tabelement. We start with a set of hand-scribbled wireframes that progress to high-fidelity wireframes, which are directly converted to LZX code.

These screens are constructed using layouts to provide them with a flexible configuration. This allows them to be resized while still maintaining a consistent spacing among the input fields. We also use the `formlayout` with validation forms and validators to perform field validation. This immediately alerts a user to invalid input values. Field validation is further leveraged to support form validation, which provides validation across the multiple tabelements comprising the form. You should now have the necessary skills to create any set of business forms within Laszlo.

8 Dynamic behavior of events and delegates

This chapter covers

- Understanding events and delegates
- Adding dynamic behavior
- Setting complex actions with attribute setters

> *When I can't handle events, I let them handle themselves.*
>
> —Henry Ford, automobile manufacturer

Although we've already shown you how Laszlo uses built-in event handling and constraints for communication among objects, our explanation has left some unresolved issues. In this chapter we'll cover these remaining issues by exploring how this communication system is based on a publisher-subscriber foundation built with *event* and *delegate* objects. Then we'll apply this knowledge to add dynamic behavior to applications. This technique supports more complex behavior by allowing objects to react to inputs in different ways depending on application context. It also allows an application to transition between various states. Earlier we used dynamic behavior in the Laszlo Market to transition between its main and checkout states. In this case, the Change button reacted in different ways, depending on the application's current state. Dynamic behavior was provided by a declarative `state` tag that executed a different declarative event handler determined by a `curr_state` constraint. But this declarative approach is limited to a finite number of states defined at compile time. In this chapter, we'll cover a more general approach that supports runtime modification to both declarative and JavaScript objects using events and delegates.

While the declarative approach provides dynamic behavior by switching between whole event handlers, we'll separate event handlers into handler headers and their underlying methods. Our approach to dynamic behavior will be achieved by replicating the information contained in these handler headers through event and delegate objects. This provides a more general approach to allow any number of methods to be dynamically configured to handle an event from a sender.

In this chapter, we'll begin by demonstrating how Laszlo implements event handling and constraints with events and delegates. Then we'll use events and delegates to dynamically change a declarative tag's behavior by modifying how they respond to events. Next, we'll explore how this same technique can be applied to layouts. By the end of this chapter, you should be comfortable dynamically changing the behavior of tags.

8.1 Exploring event-handler and constraint operation

Laszlo's event-based communication system implements the publisher-subscriber design pattern. It describes a one-to-many dependency among objects, where one object acts as a *publisher* with which other objects register as dependent *subscribers.* Whenever any of the publisher's attributes change, an updated value is immediately communicated to the subscribers. This arrangement produces a loose coupling among objects, allowing a publisher to send notifications without having to know its subscribers. This flexibility enables subscribers to be easily added and removed without impacting the publisher.

In Laszlo, events are publishers and delegates are subscribers. Assuming there are listeners, when the `setAttribute` method is used to update an object's attribute, an event sends a message to every registered delegate. Every attribute that has listeners also has an associated event object that contains a list of listening delegate objects. A message is sent by calling a registered delegate's associated method with an on+`attribute` argument. This is the basis for how Laszlo implements built-in event handling and constraints.

In the next section, we'll examine how event handlers and constraints are implemented with `event` and `delegate` objects.

8.1.1 How event handling and constraints work

You've already seen how attributes can trigger event handlers and constraints when their value is updated through the `setAttribute` method. However, when no event handlers or constraints have been declared for an attribute, then no on+`attribute` events are sent. In other words, Laszlo only creates an event for an attribute that has listeners. So when no objects are registered to receive an `onx` event, the call

```
setAttribute("x", 100);
```

defaults to this:

```
this.x = 100;
```

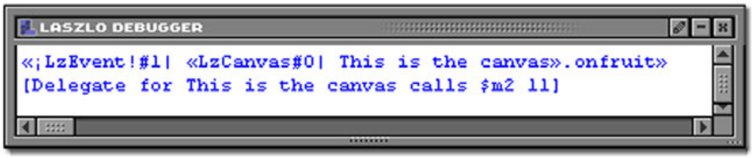
So how does Laszlo know whether a listener exists? The answer is simple. When the Laszlo compiler encounters a `handler` tag or constraint notation, it instantiates and registers a set of event and delegate objects. This communication can occur within the object itself or between objects. Listing 8.1 shows how the presence of a `handler` tag results in the instantiation of these event and delegate objects.

Listing 8.1 Demonstrating how a handler results in the creation of an event-delegate pair

```
<canvas debug="true">
   <method name="init">
      this.setAttribute("fruit", "pear");
   </method>
   <attribute name="fruit" value="apple" type="text"/>
   <attribute name="new_fruit" type="text"/>
   <handler name="onfruit" args="fruit">
```

```
            this.new_fruit = fruit;
            Debug.write(canvas.onfruit);
            Debug.write(canvas.onfruit.delegateList);
        </handler>
    </canvas>
```

An event handler is nothing more than a decorated method tag. Figure 8.1 shows how this decoration signals the Laszlo compiler to instantiate an `onfruit` event object with a delegate list containing a single delegate object. Since the method underlying this event handler is only invoked internally, it has a machine-generated name of `$m2 11`.



```
«¡LzEvent!#1| «LzCanvas#0| This is the canvas».onfruit»
[Delegate for This is the canvas calls $m2 11]
```

Figure 8.1 This debug output shows the event and delegate objects automatically instantiated to support an event handler or constraint. The canvas object is indicated by the "This is the canvas" text and the delegate object is indicated by the "$m2 11" text.

Because `event` objects play such an important role in providing communication within Laszlo, they are generally automatically instantiated and there is rarely a need to directly instantiate an `event`.

8.1.2 Working with events

To act as a publisher, an event contains an event name, the name of the originating sender, and an array of the subscribers—that is, the delegate list. While an event is in the process of being sent, its `locked` attribute is set to true.

Since Laszlo instantiates an event object to serve an integral role within the infrastructure, as table 8.1 shows, its attributes are read-only to prevent any modification.

Table 8.1 Event attributes

Name	Data Type	Tag or Script	Attribute Type	Description
`locked`	`boolean`	Script	Read-only	Value is true when an event is being sent
`name`	`string`	Script	Read-only	Event name

Table 8.1 Event attributes *(continued)*

Name	Data Type	Tag or Script	Attribute Type	Description
`sender`	`string`	Script	Read-only	Originating object
`delegates`	`array`	Script	Read-only	An array of delegates to receive this event
`ready`	`boolean`	Script	Read-only	Value is true when event is ready to be sent

In our example, a `canvas` object acts as both the sender and receiver. So the `event` object lists the `canvas` as its sender and its array of `delegates` contains a single recipient, which is also the `canvas` object. When an attribute is updated, this causes an on+`attribute` event to send an attribute message to each of the delegates in its delegate list. The left diagram in figure 8.2 shows the objects as written and the right diagram shows the resulting objects generated during compilation.

If this example is revised to use a constraint instead of an event handler, as in listing 8.2, it still produces the identical set of event and delegate objects displayed in figure 8.1.

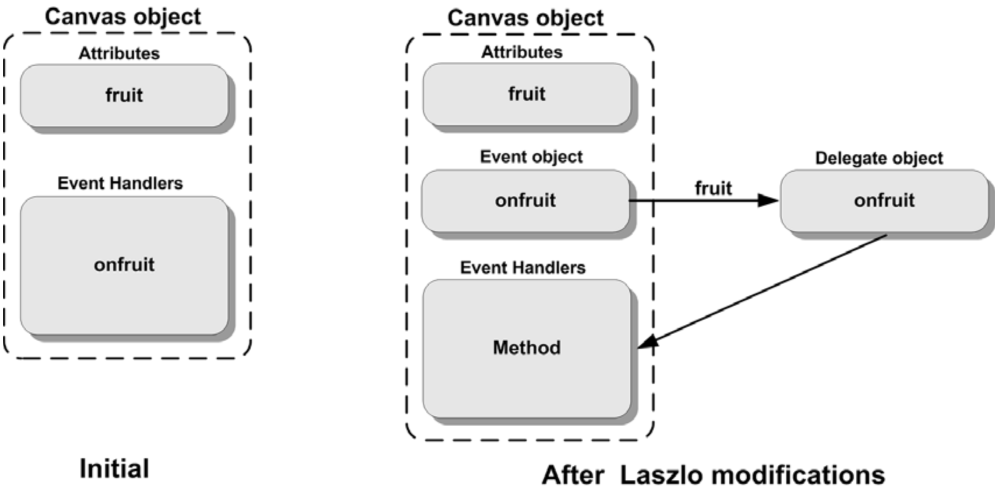


Figure 8.2 The initial state of the example in listing 8.1 directly reflects the LZX code. When the code is compiled, the event handler serves as an indicator to supplement the initial objects with the supplemental objects on the right.

Listing 8.2 Creating an event-delegate pair with a constraint

```
<canvas debug="true">
   <attribute name="fruit" value="apple" type="text"/>
   <attribute name="new_fruit" value="${canvas.fruit}"/>
   <method name="init">
      this.setAttribute("fruit", "pear");
      Debug.write(canvas.onfruit);
      Debug.write(canvas.onfruit.delegateList);
   </method>
</canvas>
```

Generates event-delegate pair

The previous examples only contained a single delegate in the event object's delegate list. Let's now turn things around by adding another delegate to this list and look at the declarative code corresponding to this delegate configuration. This requires another subscriber to receive `onfruit` events from the canvas object. Listing 8.3 shows this additional subscriber, a view object named `main`, that references the `canvas` publisher. By referencing the `canvas` object, this handler signals to the compiler that another `delegate` object should be instantiated and registered for the canvas's `onfruit` event.

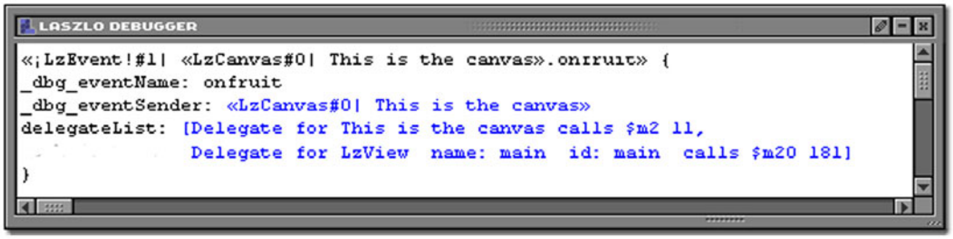
Listing 8.3 Creating more than one delegate for an event

```
<canvas debug="true">
   <attribute name="fruit" value="apple" type="text"/>
   <method name="init">
      this.setAttribute("fruit", "pear");
   </method>

   <handler name="onfruit">
      Debug.write(canvas.onfruit);
      Debug.write(canvas.onfruit.delegateList);
   </handler>

   <view name="main">
      <handler name="onfruit" reference="canvas">
         Debug.write("main onfruit handler");
      </handler>
   </view>
</canvas>
```

Adding delegate to delegate list

Figure 8.3 shows this additional delegate, called `main`, in the delegate list.

Delegates are stored in the `delegateList` array as they are encountered, and they are called in that order. But there is no way to lock a position in this array,



```
«¡LzEvent!#1| «LzCanvas#0| This is the canvas».onfruit» {
_dbg_eventName: onfruit
_dbg_eventSender: «LzCanvas#0| This is the canvas»
delegateList: [Delegate for This is the canvas calls $m2 11,
               Delegate for LzView  name: main  id: main  calls $m20 181]
}
```

Figure 8.3 Additional delegates are added to an event's delegate list, as shown in listing 8.3. The delegates are identified by their different context names. One is used to handle events sent to the canvas object and the other for the main object.

since the order of execution depends on application state. For example, if a declarative structure contains a `state` tag with event handlers referencing the sending object, the `state` tag's `applied` attribute dictates the presence or absence of a handler. As a result, you can't depend on the execution order of event handlers. Let's take a quick look at a technique for controlling their execution order.

Controlling the execution order of handlers

Because handlers can't be overridden and their calling order is indeterminate, handlers support a method argument to contain their body. If a superclass has a handler, and a subclass also declares a handler, then both handlers execute in an uncontrolled order. But since methods can be overridden, the order of handler invocation can be controlled. Suppose we define an `onfruit` handler using a `handleFruit` method to contain its body, like this:

```
<handler name="onfruit" method="handleFruit"
                                     args="fruit" />
<method name="handleFruit" args="fruit">
   // superclass code here
</method>
```

When an instance or a subclass needs to handle this event in a different manner, it can override its superclass's `handleFruit` method like this:

```
<method name="handleFruit" args="fruit">
 // pre-processing
 super.handleFruit(this, "fruit")
 // post-processing
</method>
```

Now the execution of the superclass's handling method is controlled to allow processing to occur prior or after this call. For this same reason, when creating a class it's better to override the `init` method rather than handle the `oninit` event.

Let's now take a closer look at the delegate object.

8.1.3 Working with delegates

To fulfill its duties as a subscriber, a *delegate* presents an interface allowing a specific method to be called within a specified context. It operates like an anonymous function in Java, since it acts as a function pointer within a class instance. An anonymous function has no name, to prevent it from being referenced by other callers. The intent is for a delegate to control all access to this function. An anonymous function is defined inline, in JavaScript, like this:

```
ref = function(arg1, arg2, … ) {
   // do stuff here
}
```

The only value this function can return is a reference to itself. Now, whenever the `ref` handle is evaluated, the referenced anonymous function is executed.

For a delegate to communicate with an object's event, it must register with the event as a subscriber. Table 8.2 shows that a delegate consists of a set of `c` and `f` attributes, also known as the *context* and the *function pointer*. The context identifies the receiving object, and the function pointer identifies the event handler's underlying method.

Table 8.2 Delegate attributes

Name	Data Type	Tag or Script	Attribute Type	Description
`c`	`object`	Script	Read-only	The context in which to call the method
`f`	`string`	Script	Read-only	The name of the method to call

Consequently, a delegate requires two steps for setup: instantiation and registration. There are two different delegate constructors; one performs instantiation and registration as two separate calls, while the other condenses them into a single call.

We'll start with the two-step constructor first. A delegate construction takes the following form:

```
var del = new LzDelegate(context, method);
del.register(sender, event);
```

where

- `del` is the name of the new `LzDelegate` object.
- `context` is the object that executes this method.

- method is the name of a method in the context object.
- sender is the sender of this event.
- event is the name of this event.

The context and method arguments set the values of the delegate attributes. These values can only be set once within a constructor, since afterward they are read-only. The single-step constructor combines these two steps into a single statement:

```
var del = new LzDelegate(context, method, sender, event);
```

and uses the identical set of arguments.

Let's reimplement the original example from listing 8.1 and manually instantiate these event and delegate objects. This is only done as an exercise to help demonstrate the concepts, as there's no benefit from using manual instantiation and making the developer responsible for releasing the object's resources by calling its destroy method.

In listing 8.4, we'll remove the event-handler indicator and replace it with an ordinary method to prevent the Laszlo compiler from automatic instantiation. Instead, we'll explicitly instantiate and register these objects within our init method. When the delegate is instantiated, its arguments specify the canvas as its executing context and point to a fruit_method method. This serves as our internal method that corresponds to the $m2 11 method previously created by Laszlo. No instantiation is necessary for the onfruit event object as it is automatically instantiated when the delegate's register method is executed.

Listing 8.4 Manually instantiating events and delegates

```
<canvas debug="true">
   <attribute name="fruit" value="apple" type="text"/>

   <method name="init">
      var del = new LzDelegate(canvas, "fruit_method");
      del.register(this, "onfruit");
      this.setAttribute("fruit", "pear");
   </method>

   <method name="fruit_method">
     Debug.write(canvas.onfruit);
     Debug.write(canvas.onfruit.delegateList[0]);
   </method>
</canvas>
```

❶ **Sends onfruit event**

At ❶, the onfruit event is sent when the setAttribute method updates the fruit attribute. This results in the fruit_method being invoked, which produces the same output you saw in figure 8.1.

There is no requirement for a delegate to be registered with an event. In many situations, it's convenient to have a pool of instantiated delegates available to be registered to different events. A succinct notation for utilizing delegates in this way is to declare them as attributes:

```
<attribute name="del"
   value="$once{new LzDelegate(this, 'fruit_method')}"/>
```

This code creates a delegate object that executes the fruit_method method in the context of the current object. This delegate object can easily be registered, and later unregistered, with an event. Since developers are responsible for deallocating manually instantiated objects, it makes sense to maintain a pool of reusable delegates. This results in a smaller number of delegate objects whose lifespan lasts throughout the application. But to use a pool, we'll need to know how to unregister a delegate from an event.

Unregistering delegates

Unregistering a delegate removes it from an event's delegate list, resulting in one less subscriber for a published event. A delegate can call unregisterFrom to unregister itself from a particular event:

```
del.unregisterFrom(event);
```

A delegate can also call unregisterAll to unregister itself from all events:

```
del.unregisterAll();
```

Unregistering a delegate doesn't deallocate its resources; a delegate object is still available to register with another object. The only way to deallocate an object's resources is by calling its destroy method. One of the benefits of using event handlers and constraints is that Laszlo manages the allocation and deallocation of its automatically generated event and delegate objects. This relieves developers from these tasks, as they are only responsible for deallocating manually instantiated objects. But using techniques such as pooled delegate objects further relieves this responsibility.

Next we'll look at different approaches for adding dynamic behavior to an application.

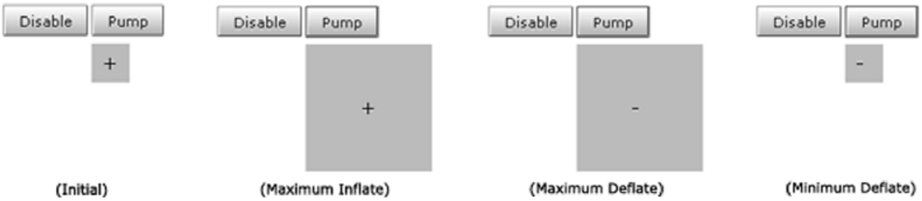
8.2 Adding dynamic behavior

Now that you've seen how event handlers and constraints are implemented with `event` and `delegate` objects, we'll show you how to use these objects to dynamically change the way an object responds to events. We'll use an example application featuring a button that inflates a box on each click. When a size limit is reached, the button's response to `onclick` events changes so that each subsequent click results in the box deflating.

Changing an object's behavior requires modifying the events and delegates that interconnect objects, events, and methods. When modifying these events and delegates, you'll find it equally valid to work from a *delegate-centric* perspective, using delegate objects to register with events, as from an *event-centric* perspective, using event objects to add delegate objects—in other words, the result is the same. However, we'll generally default to working with delegates rather than events.

8.2.1 Taking a delegate-centric perspective

Now that we can register and unregister delegates, we're ready to explore how event behavior can be dynamically modified during execution. We've created an example application that uses two buttons, Enable/Disable and Pump, to control the inflation and deflation of the box shown in figure 8.4. When the Pump button is clicked, the box inflates up to a maximum size. At that point, the pumping action reverses so that subsequent Pump clicks deflate the box down to a minimum size. The Enable/Disable button, not surprisingly, enables and disables the Pump button. Although not shown in the figure, its label alternates between Enable and Disable when clicked.



(Initial) (Maximum Inflate) (Maximum Deflate) (Minimum Deflate)

Figure 8.4 Clicking the Pump button causes the box to inflate until a maximum limit is reached. At that point, the delegates switch from an inflate to a deflate method, resulting in subsequent pumps deflating the box until a lower limit is reached. Once again, the delegates are reversed to inflate the box. The box label changes to reflect its state. An added feature is the Enable/Disable button for the Pump button, which remains enabled in the sequence shown.

This example is designed to demonstrate the usage of a cross section of the delegate methods listed in table 8.3.

Table 8.3 Delegate methods

Name	Description
`LzDelegate(context,method, eventSender, eventName)`	Delegate constructor
`disable()`	Disables a delegate until `enable` is called
`enable()`	Enables a delegate that has been disabled
`execute(data)`	Executes the named method in the given context with the given data
`register(eventSender, eventName)`	Registers the delegate for the named event in the given context
`unregisterAll()`	Unregisters the delegate for all the events for which it's registered
`unregisterFrom(event)`	Unregisters the delegate for the stated event

Listing 8.5 demonstrates how the behavior of the Pump button is changed by swapping a delegate, whose associated method inflates the box, with another delegate, whose method deflates it. We'll also temporarily disable or enable the delegate's behavior, which is useful since it doesn't require the delegate to unregister from the event.

Listing 8.5 Demonstrating delegate methods

```
<canvas debug="true">
   <button text="Disable">
      <handler name="onclick">
         <![CDATA[
         if (v.inf[0] && v.inf.enabled) {
            v.inf.disable();
            this.setAttribute("text", "Enable");
            return; }
         else if (v.inf[0] && !v.inf.enabled) {
            v.inf.enable();
            this.setAttribute("text", "Disable");
            return; }
         else if (v.def[0] && v.def.enabled) {
            v.def.disable();
            this.setAttribute("text", "Enable");
            return; }
```

Disable, if enabled and inflating

Enable, if disabled and inflating

Disable, if enabled and deflating

```
         else {
            v.def.enable();
            this.setAttribute("text", "Disable");
            return; }
         ]]>
      </handler>
   </button>
   <button name="b" text="Pump" x="80"
           onclick="v.setAttribute('pump', 0)"/>
   <view name="v" x="80" y="40" width="30" height="30"
         bgcolor="0xBBBBBB">
      <attribute name="inf"
                 value="$once{new LzDelegate(this,'inflate')}"/>
      <attribute name="def"
                 value="$once{new LzDelegate(this,'deflate')}"/>
      <attribute name="pump" type="number"/>
      <text name="msg" align="center" valign="middle"
            text="+" fontsize="16"/>
      <handler name="oninit">
         this.inf.register(v, "onpump");
      </handler>
      <method name="inflate">
         if (this.width == 100) {
            this.msg.setText("-");
            this.inf.unregisterFrom(v.onpump);
            this.def.register(v, "onpump");
            return; }
         this.msg.setText("+");
         this.setAttribute("width", this.width + 10);
         this.setAttribute("height", this.height + 10);
      </method>
      <method name="deflate">
         if (this.width == 30) {
            Debug.inspect(v.onpump);
            this.def.unregisterFrom(v.onpump);
            this.inf.register(v, "onpump");
            return; }
         this.msg.setText("-");
         this.setAttribute("width", this.width - 10);
         this.setAttribute("height", this.height - 10);
      </method>
   </view>
</canvas>
```

Enable, if disabled and deflating

Allocates inflate delegate

Allocates deflate delegate

Initially registers onpump events with inflate method

Unregisters onpump events from inflate method

Registers onpump events with deflate method

Unregisters onpump events from deflate methods

Registers onpump events with inflate method

Now that you know how to implement this from a delegate perspective, let's reimplement it from an event-centric perspective. Instead of registering and unregistering `delegate` objects to an event, we'll accomplish the same thing by adding and removing `delegate` objects from the event's delegate list.

8.2.2 Taking an event-centric perspective

An event constructor is also supplied, for those rare situations when an event needs to be manually instantiated:

```
var event = new LzEvent(eventSender, eventName);
```

where

- `eventSender` is the object sending the event.
- `eventName` is the name of the event to be sent.

An `event` contains the methods shown in table 8.4 to manage the delegates in its delegate list. Adding a `delegate` object to an `event` object, using the `addDelegate` method, results in this delegate being registered and contained in the event's `delegateList` array. But delegates, unlike events, aren't automatically instantiated by Laszlo, so adding a nonexistent delegate to an event generates a runtime error.

To demonstrate the general equivalence of the event-centric and delegate-centric perspectives, we'll reimplement the inflating-and-deflating-box example in listing 8.6 by modifying its `inflate` and `deflate` methods to work with `event` instead of `delegate` methods. Otherwise, the rest of the code is identical.

Table 8.4 Event methods

Name	Description
`addDelegate(delegate)`	Adds a delegate to the delegate list
`clearDelegates()`	Removes all delegates from the delegate list
`LzEvent(eventSender, eventName, delegate)`	Calls all delegates in turn
`getDelegateCount()`	Returns the number of delegates registered for the event
`removeDelegate(delegate)`	Removes a delegate from the delegate list
`sendEvent(data)`	Sends the event, passing its argument as data to the called delegate

Listing 8.6 The inflating-box example using events rather than delegates

```
<method name="inflate">
   if (this.width == 100) {
      this.msg.setText("-");
      this.onpump.removeDelegate(v.inf);     <-- Removes inflate delegate from event
      this.onpump.addDelegate(v.def);
      return; }
   this.msg.setText("+");
   this.setAttribute("width", this.width + 10);
   this.setAttribute("height", this.height + 10);
</method>

<method name="deflate">
   if (this.width == 30) {
      this.onpump.removeDelegate(v.def);
      this.onpump.addDelegate(v.inf);
      return; }
   this.msg.setText("-");
   this.setAttribute("width", this.width - 10);
   this.setAttribute("height", this.height - 10);
</method>
```

Adds deflate delegate to event

Removes deflate delegate from event

Adds inflate delegate to event

With this change, the inflate and deflate delegates are alternatively added and removed from an event. The result is equivalent to registering and unregistering the delegates from the event.

The dynamic communication capabilities of delegates are applied to different topics in upcoming chapters. In the next section, we'll see how delegates are incorporated into layouts to make them extensible. Although a layout is normally updated only when the `spacing` or `visible` attributes of its child subviews are changed, using delegates extends a layout to respond to changes in any attribute.

8.3 Using delegates with layouts

In this section, we'll examine how delegates expand the capabilities of layouts. We'll demonstrate this feature by revisiting the aircraft formation example to dynamically resize a configuration both horizontally and vertically whenever the largest jet's size increases. In chapter 6, we created a special layout to maintain our jets in a "vee" formation. While that layout could accommodate visibility changes—when a plane disappears, the layout closes the gap—it can't handle size changes, since all the jets were identically sized. We now want to be able to click on a plane and have its size expand by 2 pixels in both directions, while still maintaining the consistent spacing required by its formation. Figure 8.5 shows a relationship between identical and different-sized planes.

To accommodate this resizing, we need to override the way the `update` method is invoked. Normally, a layout executes its `update` method whenever the

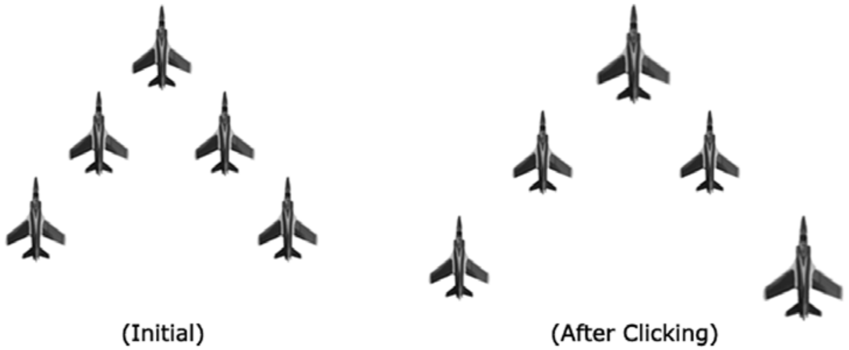


Figure 8.5 Clicking on any of the jets increases its size while maintaining a consistent spacing among the jets in both the x and y directions. Notice that the lead and bottom-right aircraft are larger than the others.

height, width, or visible attribute of any subview changes. This supports horizontal or vertical spacing, but not both. Our jet formation layout needs to override this behavior, because changes in jet size require the spacing to be updated in both directions.

The layout tag uses its updateDelegate method to establish a connection between an event and its update method. Since this must occur for each jet, we need to override the addSubview method to ensure that this method is applied to each jet subview when added to its parent node. A Laszlo system method is being overridden, so we also need to call the superclass's addSubview method:

```
<method name="addSubview" args="newsub">
   this.updateDelegate.register(newsub, "onwidth");
   this.updateDelegate.register(newsub, "onheight");
   super.addSubview(newsub);
</method>
```

The updateDelegate is a delegate object controlling access to the layout's update method. We'll register it to accept onwidth and onheight events to ensure that changes in these attributes cause the layout to be recalculated. Now, when any jet is clicked, it enlarges and generates a set of onwidth and onheight events, resulting in the layout's update method being executed and ensuring a consistent jet formation:

```
<class name="jet" resource="jet" stretches="both">
   <handler name="onclick">
      this.setWidth(this.width+2);
      this.setHeight(this.height+2);
   </handler>
</class>
```

An update only needs to occur when the largest jet increases its size, since the smaller jets can increase their size without disrupting the formation. When an update occurs, the jets redistribute themselves to accommodate the largest jet while still maintaining a consistent spacing. Spacing is set by the width and height values of the largest jet; these values are set by the `setMaxwidth` and `setMaxheight` methods. Listing 8.7 puts this all together.

Listing 8.7 Handling dynamic layouts using delegates

```
<canvas>
   <resource name="jet" src="F18_Hornet.png"/>
   <class name="jet" resource="jet" stretches="both">
      <handler name="onclick">
         this.setWidth(this.width+2);
         this.setHeight(this.height+2);
      </handler>
   </class>

   <view x="300" y="100">
      <layout>
         <attribute name="xspacing" value="50" type="number"/>
         <attribute name="yspacing" value="40" type="number"/>
         <method name="setMaxwidth" args="subviews">
            <![CDATA[
            for (var i = 0; i < subviews.length; i++) {
               if (this.xspacing < subviews[i].width)
                  this.xspacing = subviews[i].width; }        <- Finds widest plane
            ]]>
         </method>
         <method name="setMaxheight" args="subviews">
            <![CDATA[
             for (var i = 0; i < subviews.length; i++) {
                if (this.yspacing < subviews[i].height)
                   this.yspacing = subviews[i].height; }      <- Finds tallest plane
            ]]>
         </method>
         <method name="addSubview" args="newsub">
            this.updateDelegate.register(newsub, "onwidth");   <- Allows width changes to force layout update
            this.updateDelegate.register(newsub, "onheight");  <- Allows height changes to force layout update
            super.addSubview(newsub);
         </method>
         <method name="update">
            <![CDATA[
            if (this.locked) return;
```

```
                  setMaxwidth(subviews);
                  setMaxheight(subviews);
                  for (var i = 0, j = 0; i < subviews.length; i++) {
                                    if (i % 2)
                        this.subviews[i].
                             setAttribute('x', -(j * this.xspacing));
                     else
                        this.subviews[i].
                             setAttribute('x', j * this.xspacing);
                     this.subviews[i].
                          setAttribute('y', j * this.yspacing);
                     if (i % 2 == 0) j++; }
                  ]]>
               </method>
            </layout>
            <jet/>
            <jet/>
            <jet/>
            <jet/>
            <jet/>
         </view>
      </canvas>
```

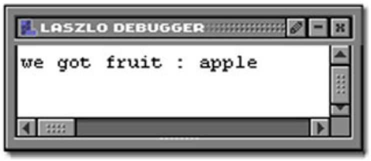
The updateDelegate method allows a layout to be dynamically reconfigured based on inputs from almost any attribute event. Such flexibility, just one example of how Laszlo tags use delegates to add dynamic capabilities, opens the way for innovation. In the next chapter, which deals with Laszlo services, you'll see other examples where delegates are used to provide even more flexibility.

8.4 Dynamically adding attributes

JavaScript allows attributes to be declared dynamically, which means they don't need to be declared at compile time. Nevertheless, it's good programming practice to declare attributes at compile time because it allows a type specifier to be used. However, if needed, an attribute can be dynamically created using only the setAttribute method, as demonstrated here:

```
<canvas debug="true">
   <handler name="oninit">
      this.setAttribute("fruit", "apple");
   </handler>
   <handler name="onfruit">
      Debug.write("we got fruit : " + fruit);
   </handler>
</canvas>
```

Figure 8.6 shows not only that an attribute was dynamically created but that it also has an event object associated with it to support event handling and constraints. This is the capability that separates attributes from ordinary JavaScript variables.



```
we got fruit : apple
```

Figure 8.6 Attributes are automatically added to an object simply by being set.

This brings up a related problem: since you don't want to clobber an existing attribute, how do you ensure that an attribute doesn't currently exist? The safest way to determine this is by checking the attribute's type field for undefined:

```
if (typeof this.getAttribute("fruit") == "undefined")
```

Simply checking for a null value won't work, because JavaScript distinguishes between null and undefined attributes and variables.

Normally, updating setAttribute results in an on+attribute event being sent. But some situations require more complex actions than can be expressed by sending a single event. For that we need *attribute setters*.

8.5 Handling complex behavior with attribute setters

Sometimes an algorithm can be more clearly expressed by having an attribute change result in a complex effect, which might require sending multiple events or having some auxiliary processing performed. For these situations, the attribute tag's setter option is used to specify a *custom setter* method to perform this processing. Because Laszlo doesn't know what processing will occur, it doesn't automatically send an on+attribute event to all registered listeners. Instead, the developer is responsible for creating and sending any events with the event object's sendEvent method. Additionally, the developer is also responsible for performing any cleanup operations to release instantiated objects.

Although a custom setter can have any name, it's considered good practice to name it with a set prefix. However, its event names still must have the form on+attribute.

Listing 8.8 is an example where a speed attribute contains a speed value, whose value must conform to several restrictions. Its value is restricted to a range of allowable speeds; any speed greater than 65 is adjusted down to 65, and any negative speed is raised to 0. Additionally, if its duration is known, the distance can also be calculated. Here's how a custom setter for speed can be specified:

```
<attribute name="speed" value="0" type="number" setter="setSpeed"/>
```

We'll now create a method called `setSpeed` as a custom attribute setter to adjust any new value and send out a set of `onspeed` and `ondistance` events.

Listing 8.8 A custom setter method that filters input sent to the `setAttribute` method

```
<canvas debug="true">
   <node>
      <attribute name="speed" type="number"
                 setter="setSpeed(speed)"/>
      <attribute name="duration" value="60" type="number"/>
      <attribute name="distance" type="number"/>

      <method event="oninit">
         this.setAttribute("speed", 100);
         this.setAttribute("speed", 55);
         this.setAttribute("speed", -10);
      </method>

      <method name="setSpeed" args="s">
         <![CDATA[
         if (s > 65) this.speed = 65;
         else if (s < 0)  this.speed = 0;
         else this.speed = s;
         if (this.duration)
            this.distance = this.duration * this.speed;
         if (this['ondistance'].ready)
            this.ondistance.sendEvent(this.distance);
         if (this['onspeed'].ready)
            this.onspeed.sendEvent(this.speed);
         ]]>
      </method>

      <handler name="onspeed" args="speed">
         Debug.write("speed: " + speed + " MPH");
      </handler>

      <handler name="ondistance" args="distance">
         Debug.write("distance: " + distance + " miles");
      </handler>
   </node>
</canvas>
```

Executes this routine after setting attribute

Sends an event if ondistance listener exists

Sends an event if onspeed listener exists

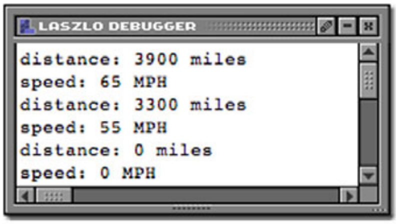
To ensure that events aren't sent to nonexistent event handlers, it's necessary to check for their existence. Unfortunately, when an undefined property is referenced, it results in a warning message sent to the log and appearing in the debug window. To perform this check without generating a warning message, the property must be referenced with JavaScript array notation rather than dot notation:

```
if (this['onspeed']) this.onspeed.sendEvent(this.speed);
```

Now an event is sent to the onspeed event handler only if it exists.

Each event handler is triggered and receives its updated value as an argument. The results are shown in figure 8.7.

Dynamic behavior techniques add another level of interactivity that can be applied to many areas in an application. Its wide applicability makes understanding how to apply dynamic behavior a fundamental technique for developers.



```
distance: 3900 miles
speed: 65 MPH
distance: 3300 miles
speed: 55 MPH
distance: 0 miles
speed: 0 MPH
```

Figure 8.7 Attribute setters allow custom processing when an attribute is set. This example imposes limits on a speed attribute so that it can't exceed 65 or be negative.

8.6 Summary

The combination of event and delegate objects provides an implementation of the publisher-subscriber design pattern to support a loosely coupled communication system in Laszlo. This system consists of publishers represented by events, and subscribers represented by delegates, where publishers send messages to their registered subscribers whenever any changes occur to their attributes. This loosely coupled one-to-many relationship has the benefit of allowing a publisher to be bound to its subscriber. This enables subscribers to be added or removed freely.

The benefit of this approach is that it allows developers to concentrate on object interactions without being burdened with the implementation details of a publisher-subscriber communication system. This relieves developers of having to instantiate and release objects required by this communication infrastructure and instead lets them concentrate on building applications. This system is also flexible enough to accommodate special behavior that doesn't fit within the standard procedure of instantiating and associating an event with each attribute that has registered listeners.

Once developers understand the operation of events and delegates, they can use these tools to add dynamic behavior to their applications. This allows for increased application complexity since an object's response to events can be made dependent on application context. Input controls can be overloaded to respond differently to events to support different operating contexts. Adding dynamic behavior to an application is a fundamental technique that will be leveraged in different ways in the upcoming chapters.

9

Using Laszlo services

This chapter covers

- Using services
- Building a stopwatch
- Building modal window interfaces
- Building drag-and-drop networks

> *A child of five would understand this. Send someone to fetch a child of five.*
>
> —Groucho Marx, humorist

Even the most visually stunning graphical displays aren't terribly useful if there is no way to interact with them. Users have grown accustomed to the usability of their favorite desktop applications and expect web-based applications to be similar. In particular, they expect support for a full range of input services such as modal windows, the mouse scroll wheel, keyboard shortcuts, tool tips, access to menus from the right mouse button, and drag-and-drop operations.

Most of this *input functionality* is supported in Laszlo by a new object type called a *service*. Up to now, we have only been working with `LzNode`-based objects. This is the base class that allows objects to be used declaratively and to participate in various communication systems. But services aren't based on this class; instead, each service is implemented as a *global singleton*. You might be familiar with the term *singleton*, since it's probably the most widely used design pattern in Java. The pattern is simple; a class is limited to a single instance. Since it has a single instance, a singleton class needs to be globally accessible, and it must have publicly available static methods.

9.1 Overview of services

Singletons are useful when a single object is needed to coordinate actions across a system. This makes them appropriate for representing the input devices, such as the keyboard, mouse, or cursor. In addition, a number of software interfaces are represented as services.

Although Laszlo uses services to handle many system-related functions, in this chapter we'll focus on those services that interface to users through components and other view-based objects. We'll also supplement this set with some basic abstract services—`LzIdle`, for generating a series of fixed interval events, and `LzTimer`, for establishing timed intervals—since they can be used in combination with these services. Table 9.1 contains a list of the services that we'll cover in this chapter.

Table 9.1 Input services covered in this chapter

Service Name	Description
`LzBrowser`	Interacts with the browser
`LzCursor`	Modifies the mouse cursor image
`LzFocus`	Handles keyboard focus

Table 9.1 Input services covered in this chapter *(continued)*

Service Name	Description
`LzGlobalMouse`	Provides application-wide control of and information about the mouse
`LzIdle`	Generates events at a fixed frequency
`LzKeys`	Handles keyboard input events
`LzModeManager`	Controls modal properties of windows
`LzTimer`	Invokes an action after a specific time interval
`LzTrack`	Tracks mouse events over a group of views

Each service contains a set of methods for interfacing with one of the standard input devices shown in figure 9.1. For example, there are services to control the behavior of a physical device—`LzCursor` for the mouse cursor and `LzKeys` for keyboard input—or the behavior of a software component—`LzFocus` and `LzModeManager`.

Since services are not derived from the `LzNode` class, their operating characteristics are slightly different than other objects. We'll first cover the various ways that services can interact with objects. Then we'll build some simple examples to highlight each of these techniques. Finally, we'll see how services can be used to add functionality to the Laszlo Market application.

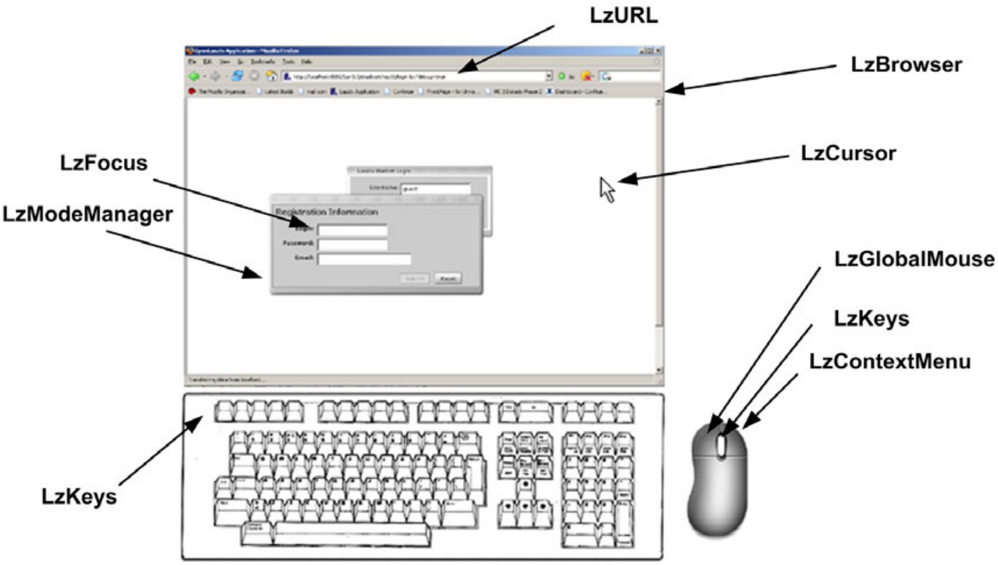


Figure 9.1 The Laszlo services interface to the physical devices and display components of an application.

9.2 Different ways to use a service

Interfacing to services is simpler and at the same time more difficult than interfacing to other objects. Because they are singletons, they don't need to be instantiated; a global singleton is created for each service during an application's initialization. Because all their methods are static, they are easily accessible through JavaScript by prefacing the method name with the name of the service. So all services can be contacted by calling one of their methods.

But using a service can be problematic—it can't communicate with other objects by sending or receiving events. Although services can't operate as subscribers, some services need to send events. For this situation, Laszlo has supplied each service with auxiliary functionality, allowing them to send events in one of two ways:

- Through a registration method
- Through a declarative reference

We'll now demonstrate each of these techniques for interacting with a service.

9.2.1 Calling a service method

The most straightforward way to use a service is to call one of its methods. Because a service doesn't need to be instantiated and is globally accessible, calling a method is easy. In listing 9.1, `LzFocus.setFocus` is used to initially set focus to the login input field. For browsers such as IE that set focus within the web application, using `setFocus` allows input to be immediately entered into the login field. Unfortunately, some browsers, such as Firefox, hold onto focus. This forces users to manually place the cursor within the login field. Nevertheless, it's still always a good idea to set focus for those users who can use it.

Listing 9.1 Calling an `LzFocus` service method

```
<canvas>
   <handler name="oninit">
      LzFocus.setFocus(login);
   </handler>
   <simplelayout axis="y"/>
   <edittext name="login" width="100"/>
   <edittext name="password" width="100"/>
</canvas>
```

When a service is used in this way, there's no object context, so the entire application is affected. Next you'll see how a service and an object can directly communicate with a service.

9.2.2 Receiving service events through registration methods

Services that need to send events containing a combination of possible arguments are supplied with a registration method. A registration method is supplied with a delegate and an array argument. The array argument contains a combination of arguments that can be sent. Additionally, this method can be called repeatedly to build up a sequence of arguments.

Listing 9.2 shows how the `LzKeys` service uses a `callOnKeyCombo` registration method to deliver from among a range of possible keyboard values to an object. Whenever the Ctrl+A, Ctrl+B, or Ctrl+C combination of keys is pressed, `callOnKeyCombo` sends an event, along with the matching key value as an argument, to its delegate. This results in a call to the `main` view's `keySeq` method for this series of keystroke combinations. `LzKeys` also contains a `removeKeyComboCall` method to unregister this subscriber.

Listing 9.2 Receiving a service event through a registration method

```
<canvas debug="true">
   <view name="main">
      <method name="keySeq" args="key">
         Debug.write("Escape key pressed : " + key);
      </method>
   </view>
   <handler name="oninit">
      var del = new LzDelegate(main, "keySeq");
      LzKeys.callOnKeyCombo(del, [ "control", "a" ]);
      LzKeys.callOnKeyCombo(del, [ "control", "b" ]);
      LzKeys.callOnKeyCombo(del, [ "control", "c" ]);
   </handler>
</canvas>
```

Now we'll look at a declarative technique supported by other services.

9.2.3 Receiving service events through declarative references

Another way some services send events is through the reference field of an event handler. Table 9.2 lists these services and their events.

Table 9.2 Service events

Service	Event	Description
`LzFocus`	`onfocus`	Sent when focus changes
`LzIdle`	`onidle`	Sent on each idle event
`LzKeys`	`onkeydown`	Sent when a key is pressed
	`onkeyup`	Sent when a key is released
`LzModeManager`	`onmode`	Sent when the mode changes
`LzTrack`	`onmousetrackout`	Sent when a mouse is dragged out
	`onmousetrackover`	Sent when a mouse is dragged over
	`onmousetrackup`	Sent when the mouse button is released

Because the service event is specified declaratively, the delegate instantiation and registration are automatic. In this example, the canvas contains handlers for the `onkeyup` and `onkeydown` events sent by the `LzKeys` service:

```
<canvas debug="true">
   <handler name="onkeydown" reference="LzKeys" args="d">
      Debug.write("key " + d + " down");
   </handler>
   <handler name="onkeyup" reference="LzKeys" args="d">
      Debug.write("key " + d + " up");
   </handler>
</canvas>
```

Now that you've seen how objects can interact with services, let's look at how these base services can be used to enrich an application with powerful features, such as timers, modal window controllers, and drag-and-drop operators.

9.3 Building a stopwatch

This example shows how dynamic behavior can be added to a simple time-keeping mechanism, the `LzTimer` service, to create a stopwatch. The `LzTimer` service publishes an event after a specified interval of time has elapsed to allow subscribers to handle this event. Although it can't be guaranteed that the delegate will be called at the precise specified time in the future, the call will not occur before that time. `LzTimer` can easily be used to create a clock by adding a `resetTimer` call at the end of each `updateTimer` cycle. This produces a repeating cycle, whose interval is the period of each clock tick. Adding each unit of time to a total produces a clock.

In listing 9.3, we add stopwatch functionality by allowing the clock to be started, stopped, and restarted without resetting its total elapsed time.

Listing 9.3 An `LzTimer` stopwatch

```
<canvas debug="true">
   <simplelayout inset="10" axis="x" spacing="2"/>
   <button y="10" text="Timer" fontstyle="bold">
      <handler name="onclick">
         this.onclick.clearDelegates();
         this.del = new LzDelegate(this, "stopTimer");
         this.del.register(this, "onclick");
         time.updateTimer();
      </handler>
      <method name="stopTimer">
         time.del.disable();
         this.onclick.clearDelegates();
         this.del = new LzDelegate(this, "restartTimer");
         this.del.register(this, "onclick");
      </method>
      <method name="restartTimer">
         time.del.enable();
         time.updateTimer(this);
         this.onclick.clearDelegates();
         this.del = new LzDelegate(this, "stopTimer");
         this.del.register(this, "onclick");
      </method>
   </button>
   <text name="time" y="10" resize="true"
         fontsize="14" fontstyle="bold">
      <attribute name="elapsedTime" type="number" value="0"/>
      <method name="updateTimer">
         this.setAttribute('text', this.elapsedTime/10);
         this.elapsedTime++;
         if (typeof this.del == "undefined"){
            this.del = new LzDelegate(this, "updateTimer");
            LzTimer.addTimer(this.del, 100);}
         else LzTimer.resetTimer(this.del, 100);
      </method>
   </text>
</canvas>
```

❶ Prevents subsequent invocations

❷ Stops timer on next onclick

❸ Starts timer

❹ Restarts timer

❺ Starts another timing cycle

❻ Creates delegate for LzTimer

❼ Calls timer again

❽ Resets timer on subsequent calls

Executing this application produces the results shown in figure 9.2.

The first time the `updateTimer` method is called, it sets the tick interval by calling `addtimer` with a value of 100 msecs. To prevent subsequent `onclick` events from accessing the `updateTimer` method, we'll remove all delegates from the `onclick` event ❶. Next a `stopTimer` event handler is registered and bound to

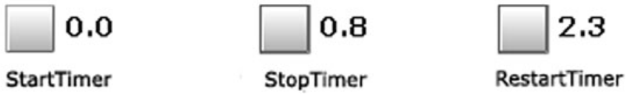


Figure 9.2 The event handler associated with `onclick` events is dynamically changed. The first time the button is clicked, it starts the timer. The second click stops the timer. This switching of event handlers allows the timer to be repeatedly started and stopped without losing its count.

`onclick` events ❷. The next time this button is clicked, its `onclick` event is handled by the `stopTimer` method to stop the timer. Finally, we'll start the clock ticking at ❸ by calling `updateTimer`.

The repeating cycle of the clock is established by creating a delegate for the `updateTimer` method ❻, and calling `addTimer` with an argument of 100 msecs ❼. This `addTimer` method only needs to be called once. The timer expires in at least 100 msecs, and then calls the `updateTimer` method associated with its delegate argument to repeat the cycle. But on its next trip ❽ through `updateTimer`, the timer delegate exists, so the timer is reset to the next increment of 100 msecs. This completes the clock.

Now we'll add the stopwatch functionality. When the button is clicked, its `onclick` event calls the `stopTimer` method ❹ to stop the timer. Once again, we'll remove all delegates from the `onclick` event so we can change its functionality on the fly, by registering the `restartTimer` method with the `onclick` event. Now the next time the button is clicked, the `restartTimer` method is called to restart the timer ❺ to continue counting time clicks.

`LzTimer` is a powerful tool to maintain state within an application. Instead of writing every state change to a server, an `LzTimer`-based method could be used to automatically update the client state to a server at an established time period. This allows application state to be recovered if the application prematurely terminates.

9.4 Demonstrating services with a login window example

To demonstrate the other services, we'll add simple login functionality to the Laszlo Market. This isn't intended to provide realistically secure access to our application. For example, we haven't yet connected to an authenticating back-end server. Rather, the goal is only to illustrate a cross section of services. But it will provide a gateway to the application for a new user to register through a nested sequence of windows.

Since only a generic nondescript login window is needed, in the interest of brevity we'll skip the wireframe diagrams. Recall that *focus* determines which window receives keyboard input. When a window receives focus, it gets moved to the front along the z-axis to become the front-most window. Table 9.3 lists the methods available to the `LzFocus` service.

Table 9.3 `LzFocus` methods

Name	Description
`clearFocus()`	Sends an `onblur` event to the currently focused view, if there is one, and removes the focus from it.
`getFocus()`	Returns the currently focused view.
`getNext(focusview)`	Returns the next focusable view.
`getPrev(focusview)`	Returns the previous focusable view.
`next()`	Moves the focus to the next focusable view.
`prev()`	Moves the focus to the previous focusable view.
`setFocus(focusview)`	Sets the focus to the given view. If this is not the currently focused view, an `onblur` event is sent to the currently focused view, and an `onfocus` event is sent to the new view.

Although focus is generally controlled by mouse or keyboard input, it can also be set programmatically and controlled with the `LzFocus` methods listed in table 9.3. The `setFocus` method establishes focus on a particular view. Once this is done, focus can be moved with the `next` and `prev` methods. The current location in a screen can be determined with any of the `getFocus`, `getNext`, or `getPrev` methods. Finally, focus can be removed with the `clearFocus` method. Whenever focus changes from one window to another, an `onblur` event is sent to both windows.

When the Login window is displayed, all other views in the application must be inoperable until that window is closed. This behavior requires a *modal window*. The `modaldialog` component supplies modal window behavior and the `login` class extends its properties. Listing 9.4 shows our example Login window.

Listing 9.4 Using `LzFocus` for a Laszlo Market Login window

```
<canvas debug="true">
   <include href="graybox.lzx"/>
   <include href="incubator/formlayout.lzx"/>
   <include href="incubator/validators"/>
   <include href="check_validators.lzx"/>

   <handler name="oninit">
      LzFocus.setFocus(logwin.top.form.uname);
      logwin.open();
   </handler>

   <class name="login" extends="modaldialog" title="Login"
          content_inset_bottom="3" content_inset_top="3"
          content_inset_left="5" content_inset_right="5">
      <graybox name="top" width="${immediateparent.width}"
               height="${immediateparent.height}">
         <simplelayout inset="5" axis="y" spacing="5"/>
         <view name="form">
            <formlayout align="right"/>
            <text>Username:</text>
            <edittext name="uname" width="150" text="guest"/>
            <text>Password:</text>
            <edittext name="upass" password="true"
                      width="150" text="guest"/>
         </view>
         <view>
            <button text="OK" isdefault="true"
                    onclick="logwin.doLogin()"/>
            <button text="Cancel" doesenter="false"
                    onclick="logwin.doCancel()"/>
            <button text_padding_y="8" text="Register" doesenter="false"
                    onclick="regwin.openWindow()"/>
            <simplelayout inset="65" axis="x" spacing="10"/>
         </view>
      </graybox>
      <method name="doLogin">
         Debug.write("Interface to back-end for authentication");
         gController.setAttribute("appstate", "Login to Main");
         this.close();
      </method>
      <method name="doCancel">
         logwin.top.form.uname.clearText();
         logwin.top.form.upass.clearText();
      </method>
   </class>

   <login name="logwin" width="320" height="160"/>
</canvas>
```

❶ Opens Login window

❷ Sets up default guest login

❸ Clears login and password

Since the Login window is implemented as a modaldialog, it must be explicitly opened ❶ with the open method. We'll only implement the bare requirements necessary to log in to an application. Login accounts ❷ default to guest. Additionally, there is a Register button to allow new users to register. In a subsequent section, we'll implement this functionality, but currently it only contains a stub. Finally, the Cancel button clears ❸ the Username and Password fields. Figure 9.3 shows the complete Login window.

The OK button's `isdefault` attribute is set to true, and the other buttons have their `doesenter` attribute reset to false. Now when the user presses Enter, the action associated with the OK button is triggered.

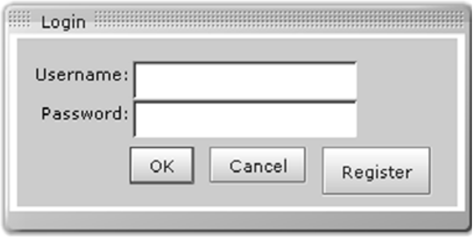


Figure 9.3 The OK button is established as the default button, so any Enter key input results in the action associated with the OK button.

9.4.1 Controlling the mouse cursor

The mouse cursor, by default, appears as an arrow, but its image changes to reflect its context and the application state. Several images are accepted as general conventions; the hourglass indicates that the system is busy, the I-beam indicates that text is selectable or editable, and the outstretched pointing hand indicates a clickable link. Changing the cursor icon is an effective way to provide the user with subtle hints about application state. Table 9.4 shows the `LzCursor` methods for controlling changes to the cursor image.

Table 9.4 `LzCursor` methods

Name	Description
`lock()`	Prevents the current cursor image from changing
`setCursorGlobal(resource)`	Sets the cursor to a resource
`showHandCursor(boolean)`	Shows the hand cursor when the mouse is over a clickable view
`unlock()`	Restores the default cursor image and allows the image to be changed

As an example of the `LzCursor` service, we can change the cursor when the mouse is over the Register button of the Login window. We can easily modify listing 9.4 to accomplish this. The code in the Register `button` tag is expanded to include additional event handlers for the `onmouseover` and `onmouseout` events to control the setting of the mouse cursor:

```
<resource name="handcursor_plus" src="handcursor_plus.png"/>
…
<button x="10%" y="75" text_padding_y="8"
        onmouseover="setCursor()" onmouseout="unsetCursor()"
        onclick="register.openWindow()">Register
   <method name="setCursor">
      LzCursor.setCursorGlobal("handcursor_plus");
   </method>
   <method name="unsetCursor">
      LzCursor.unlock();
   </method>
</button>
```

In each case, a method is called to change the mouse cursor while the mouse pointer is over the button, and to restore the mouse cursor back to its original image when the mouse moves outside the button. Figure 9.4 shows the result; the "+" in the upper-right corner of the cursor indicates that a new user can be added through a Registration window.

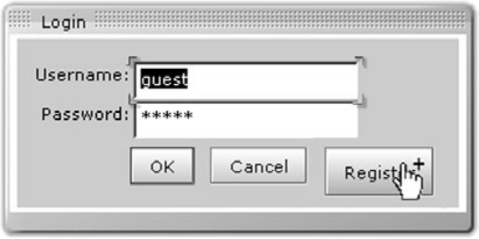


Figure 9.4 LzCursor is used to change the mouse cursor to the pointing hand when over the Register button. The cursor has a "+" in the upper-right corner to indicate that a new user can be added through a Registration window.

One problem in mouse-cursor management is finding an image, especially one that fits into 32 pixels, and producing an immediate association with an activity. The only generally accepted icons are the hourglass, the hand, and the text I-beam. Rather than use an unfamiliar icon, it's often easier and more informative to simply provide a text description with a tool tip.

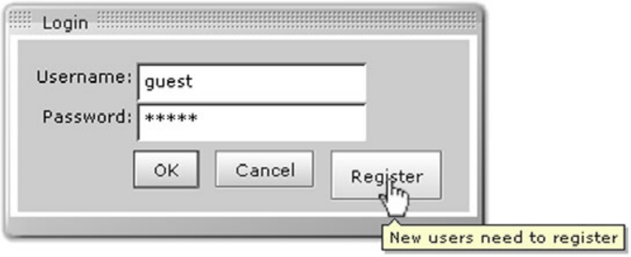
Adding tool tips

Although tool tips aren't a service, we've included a section on them here because they're a great way to unobtrusively provide additional information to users. Although they are generally used to provide informational text labels for graphical icons, they can provide descriptive help messages for any field. Such a help message appears on the screen as long as the mouse is over the target field. Tool tips are specified in Laszlo with the `tooltip` tag.

The alignment of a tool-tip message defaults to left-justified but can be changed to right-justified with the `tipalign` attribute. Tool tips can be found in the incubator directory in the Laszlo distribution. We can add a tool tip to the Register button in our market Login window by simply adding it as a child view to the parent tag. Modify the Register button to look like the following:

```
<include href="incubator/tooltip/tooltip.lzx"/>
   …
   <button text_padding_y="8" text="Register"
           doesenter="false" onclick="regwin.openWindow()">
      <tooltip>New users need to register</tooltip>
   </button>
```

As figure 9.5 shows, a tool tip complements any application by providing a convenient online help facility. Since they are so easy to add, all graphic icons should be furnished with a tool tip with a text explanation.



Username: guest
Password: *****

New users need to register

Figure 9.5
Tool-tip messages should be extensively used in any application, since they provide informative help messages in a convenient and unobtrusive way.

The Login window acts as a gateway to the application by restricting further access until a valid login is presented to close its dialog window. However, new users need to access the Register window to sign up for a login and a password. The Login window's restrictive behavior is too draconian. Instead, a way is needed to provide controlled access to its child windows. This functionality is provided by the `LzModeManager` service.

9.4.2 Sequencing windows with LzModeManager

A *mode manager* maintains a stack of *modal view* objects, constituting the windows present on the screen at any one time, and ensures that only they can receive mouse and keyboard events. The modal view mechanism is used to direct a user through a series of sequential actions by controlling the passage of mouse and keyboard events within a sequence of windows. This mechanism is used whenever there is a critical sequence of steps that must be performed in a particular order.

While the judicious use of modal windows can be comforting by providing the user with a sense of guidance, their overuse can quickly become annoying and make users feel as though they're trapped in a maze.

A view is made modal when it calls `LzModeManager.makeModal`. This places the view and its subviews on the modal stack. Now a parent and its children views can be accessed, with the restriction that a parent can't close until all its children are also closed.

Although LzModeManager has no attributes, it does have several methods for manipulating the properties of the modal stack. These are listed in table 9.5.

Table 9.5 LzModeManager methods

Name	Description
globalLockMouseEvents()	Prevents all mouse events from firing
globalUnlockMouseEvents()	Restores normal mouse event firing
hasMode(view)	Tests whether the given view is in the mode list
makeModal(view)	Pushes the view onto the stack of modal views
passModeEvent(event, view)	Allows certain events to be passed to nonmodal views
release(view)	Removes the view, and all the views below it, from the stack of modal views
releaseAll()	Clears all modal views from the stack

Using modal windows in the Market

The registration process occurs through a modal regwin window that's only accessible through the Login logwin window. Once the Register window opens, the Login window is not responsive until the Register window has closed. Listing 9.5 shows its implementation.

Listing 9.5 The Register modal window in the context of the Login window

```
<canvas>
   …
   <class name="logwin" … >
      …
   </class>
   <class name="register" extends="window"
          fontstyle="bold" fontsize="12"
          visible="false" align="center"
          valign="middle">
      <graybox name="graybox"
               width="${parent.width-10}"
               height="${parent.height-10}">
         <simplelayout axis="y" spacing="5"/>
         <text fontsize="16"
               text="Registration Information"/>
         <validatingForm name="form"
                         width="100%">
            <formlayout align="right"
```

```
                        spacing="5"/>
            <text text="Login:"/>
                  <checkString name="login"/>
            <text text="Password:"/>
                  <checkString name="password"/>
            <text text="Email:"/>
                  <checkEmail name="email"/>
            <method event="onerrorcount" args="val">
               if (val == 0)
                  parent.buttons.submit.setAttribute("enabled", true);
            </method>
         </validatingForm>
         <view name="buttons" x="60%">
            <simplelayout axis="x" spacing="10"/>
            <button name="submit" enabled="false" text="Submit"
                    isdefault="true" onclick="classroot.closeWindow()"/>
            <button name="reset" text="Reset"
                    doesenter="false"
                    onclick="parent.parent.clearFields()"/>
         </view>
         <method name="clearFields">
            form.login.editbox.clearText();
            form.password.editbox.clearText();
            form.email.editbox.clearText();
         </method>
      </graybox>
      <method name="openWindow">
         this.open();
         LzModeManager.makeModal(this);
      </method>
      <method name="closeWindow">
         LzModeManager.release(this);
         this.close();
      </method>
   </class>

  <login name="logwin" width="320" height="160"/>
  <register name="regwin" width="460" height="220"
            visible="false"/>
</canvas>
```

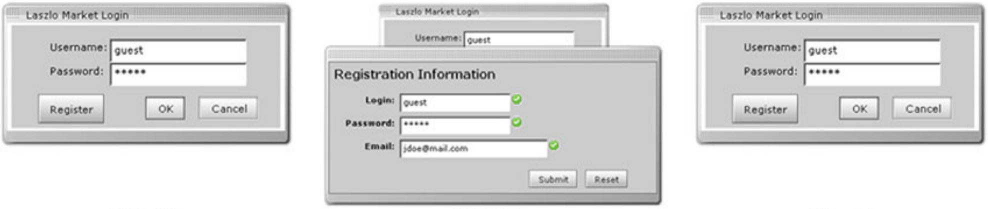
Makes Register window modal

Releases Register window from being modal

Defines Login window

Defines Register window

Figure 9.6 shows the results of executing listing 9.5. In step 1, the Login window appears as a `modaldialog` window, able to receive all mouse and keyboard events. In step 2, the Register window has been opened and made modal. The Login window is now blocked from receiving any mouse or keyboard events. If the window contained any Close or Resize icons, they would also be disabled. The Register window can be released from the modal manager by calling `LzModeManager`'s



Username: guest
Password: *****

Login: guest
Password: *****
Email: jdoe@mail.com

Username: guest
Password: *****

(Step 1) **(Step 2)** **(Step 3)**

Figure 9.6 The login sequence shown in listing 9.5 uses a modal register window so that focus returns to the Login window after the Register window has been successfully completed.

`release` method. But we've configured things so that it is only accessible through the Submit button, which is enabled only when both of its input fields are completed and validated. In step 3, control has returned to the Login window.

Just as a house has a locked door and then a foyer transitioning to the living area, an application should supplement its login facility—the locked door—with an intermediary step—the foyer—before proceeding directly into the main area. We'll add an animated transition to provide a foyer for the Laszlo Market. Now our application will gently materialize, instead of abruptly popping into view. Later we'll see how this transition period also provides a good place to hide startup costs. Listing 9.6 shows the code to create this transition.

Listing 9.6 A smooth transition from login to the application

```
<canvas>
   <handler name="oninit">
      if (LzBrowser.getInitArg("lzunit") == "true")
         gController.setAttribute("appstate", "UnitTest");
      else
         gController.setAttribute("appstate", "Splash to Login");
   </handler>
   …
   <login name="logwin" width="320" height="160"/>
   <register name="regwin" width="460" height="220" visible="false"/>

   <view name="main" width="100%" height="100%" opacity="0">
   …
   <state apply="${gController.appstate == 'Login to Main'}">
     <animator target="main" attribute="opacity"
                 duration="1500" to="1.0"
                 onstop="testsuite.testcase.main_test()"/>
   </state>
   …
</canvas>
```

Sets initial state to Login

Sets main view to initially be transparent

Materializes main view over one second

This transition is created by having the application gradually increase its opacity to 100 percent. Since our application architecture allows additional operators to be easily added, a `Login to Main` operator is added that contains an animator to generate this effect.

9.4.3 Capturing keyboard input with LzKeys

The `LzKeys` service allows keyboard input to be directed to any view. It handles not only text input but also keyboard bindings such as the Shift, Ctrl, and Esc keys. This allows actions to be associated with keyboard sequences. For example, the key sequence Shift+F1 could be configured to execute the method associated with a delegate. Since `LzKeys` is a service, key bindings are global throughout the application, triggering regardless of which view has focus.

The `LzKeys` service has a variety of attributes and methods for determining which keys are currently pressed. Table 9.6 lists the attributes and table 9.7 the methods.

Table 9.6 `LzKeys` attributes

Name	Data Type	Tag or Script	Attribute Type	Description
`downKeysArray`	`array`	Script	Read-only	An array of currently pressed key codes
`downKeysHash`	`object`	Script	Read-only	A hash by which each pressed key is set to true
`keyCodes`	`object`	Script	Read-only	A hash that maps key names to key codes

The `downKeysArray` attribute and the `getDownKeys` method both return an array of key codes for all keys that are currently pressed. The same information is available from the `downKeysHash` attribute, as an associative array where all the pressed keycodes map to true. The `isKeyDown` method takes a key name—such as Shift or Tab—and returns a boolean indicating whether or not the key is down.

Table 9.7 `LzKeys` methods

Name	Description
`callOnKeyCombo(delegate, keycodeArray)`	Instructs the service to call the given delegate whenever the given key combination is pressed
`getDownKeys()`	Returns an array of the keys currently in the down position

Table 9.7 `LzKeys` methods *(continued)*

Name	Description
`isKeyDown(key)`	Checks if a particular key is down
`removeKeyComboCall(delegate, keycodeArray)`	Removes the request to call the delegate on the key combo

In addition to interfacing to the keyboard, `LzKeys` interfaces to the mouse's scroll wheel. In chapter 10, you'll see how the mouse's scroll wheel can be attached to a scrollbar to scroll through its contents.

Using LzKeys in the Laszlo Market

`LzKeys` can be added to the Laszlo Market to support "hot keys." This allows the Shift+F1 and Shift+F2 key sequences to trigger the Main-to-Checkout and Checkout-to-Main screen transitions. While `LzKeys` sends an event, there is no need for an argument, so it is omitted. Listing 9.7 shows how the `LzKeys` service uses registration methods to send events.

Listing 9.7 Using the `LzKeys` service

```
<canvas>
   <handler name="oninit">
      var echo = new LzDelegate(this, 'gotoCheckout');
      LzKeys.callOnKeyCombo(echo, ['f1', 'shift']);
      var del = new LzDelegate(this, 'gotoMain');
      LzKeys.callOnKeyCombo(del, ['f2', 'shift']);
      ...
   </handler>

   <method name="gotoCheckout">
      gController.setAttribute("appstate", "Main to Checkout");
   </method>
   <method name="gotoMain">
      gController.setAttribute("appstate", "Checkout to Main");
   </method>
   ...
</canvas>
```

Sends an event

The `LzKeys` service is globally accessible, so you can set up a hot key to support almost any function in a Laszlo application.

9.5 Building a drag-and-drop network

We'll first look at combining the LzGlobalMouse and LzIdle services to implement a simple drag-and-drop system. Then, we'll look at the advantages provided by building drag-and-drop networks with the LzTrack service. But all drag-and-drop operations are based on mouse events, so let's start by covering how local and global mouse events serve complementary purposes.

9.5.1 Detecting local and global mouse events

Mouse events that report only within the boundaries of a view are known as *local mouse events*. Because it's frequently necessary to receive events for all mouse state changes regardless of the mouse's position, the LzGlobalMouse service provides global mouse events that report mouse state changes from any screen location and under all conditions. It even reports mouse events from nonclickable and LzModeManager-controlled views. Listing 9.8 demonstrates how the LzGlobalMouse service uses a declarative reference to send events.

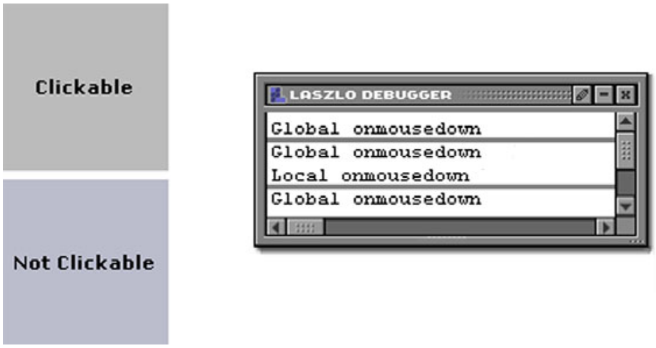
Listing 9.8 Using the LzGlobalMouse service

```
<canvas debug="true">
   <simplelayout axis="y" spacing="5"/>
   <view width="100" height="100" bgcolor="0xBBBBBB">
      <text text="Clickable" align="center"
            valign="middle" fontstyle="bold"/>
      <handler name="onmousedown">
         Debug.write ("Local onmousedown");
      </handler>
   </view>
   <view width="100" height="100" bgcolor="0xBBBCCC" clickable="false">
      <text text="Not Clickable" align="center"
            valign="middle" fontstyle="bold"/>
      <handler name="onmousedown">
         Debug.write ("Won't ever be displayed");
      </handler>
   </view>
   <handler name="onmousedown" reference="LzGlobalMouse">
      Debug.write ("Global onmousedown");
   </handler>
</canvas>
```

Reports local mouse events

Reports LzGlobalMouse events

Figure 9.7 shows the debugger output for two views, one clickable and one not. When the mouse is clicked outside of both views, the onmousedown event is only reported by LzGlobalMouse. When the mouse is clicked within the top clickable



```
Global onmousedown
Global onmousedown
Local onmousedown
Global onmousedown
```

Figure 9.7 The `LzGlobalMouse` service reports all mouse events from any location under any conditions. The first mouse click occurs outside the views, so only `LzGlobalMouse` reports it. The second mouse click occurs within the clickable view, so it is reported both locally and globally. The third mouse click occurs within the nonclickable view, so it is only reported by `LzGlobalMouse`.

view, the `onmousedown` event is reported both locally and by the `LzGlobalMouse` service. But when the bottom nonclickable view is clicked, the event is only received by `LzGlobalMouse`. Figure 9.7 shows the difference between the reported local and global mouse events.

`LzGlobalMouse` enables global mouse tracking in an application, which is necessary for drag-and-drop operations since a view-based object can be transported and deposited to any location on the screen. The next step involves combining `LzGlobalMouse` with the `LzIdle` service to generate continuous tracking.

9.5.2 Generating continuous tracking with LzIdle

The next step needed to build our drag-and-drop system is an object that generates a continuous stream of events. The `LzIdle` service generates a continuous stream of `onidle` events. These events are ideal for tracking mouse movements, because they are issued at the highest frequency available to Laszlo. This ensures the smallest possible latency between mouse movements and their corresponding screen display. Listing 9.9 shows how a drag-and-drop operation can be written using a combination of `LzIdle` and `LzGlobalMouse` services.

Listing 9.9 Implementing drag-and-drop with `LzIdle` and `LzGlobalMouse`

```
<canvas>
   <class name="dragger" width="$once{parent.width}" height="20">
      <attribute name="text" value="" type="string"/>
      <attribute name="count" value="0" type="number"/>
      <text name="txt" text="${parent.text}"
```

```
            resize="true" align="center"/>
      <handler name="onmousedown">
          this.image = new dragger(canvas,
                       {width: this.width, height: this.height,
                       bgcolor: 0xCCCCCC, clickable: true});
          this.image.setAttribute("text", this.text + ++this.count);
          this.startDrag();
      </handler>
      <method name="startDrag">
         if (typeof this.updatePos == "undefined")
         this.updatePos = new LzDelegate(this, "updatePosition");
         this.updatePos.register(LzIdle, "onidle");
         if (typeof this.eDrag == "undefined")
            this.eDrag = new LzDelegate(this, "endDrag");
         this.eDrag.register(LzGlobalMouse, "onmouseup");
      </method>
      <method name="updatePosition">
         this.image.setX(canvas.getMouse("x"));
         this.image.setY(canvas.getMouse("y"));
      </method>
      <method name="endDrag">
         var x = this.image.x - target.x;
         var y = this.image.y - target.y
         if (target.containsPt(x, y)) Debug.write("hit");
         else Debug.write("miss");
         this.updatePos.unregisterAll();
         this.eDrag.unregisterAll();
      </method>
   </class>
   <dragger text="Laszlo" bgcolor="0xBBBBBB" width="60"/>
   <view name="target" x="100" y="50"
         width="100" height="100" bgcolor="0xCCCBBB">
      <text valign="middle" align="center"
            fontstyle="bold" fontsize="16" text="Target"/>
   </view>
</canvas>
```

❶ Creates new dragger instance

Starts dragging

❷ Updates position on clock tick

❸ Ends dragging

Updates floater's x coordinates

❹ Tests location of drop object

❺ Releases delegates

In listing 9.9 a `dragger` object is dragged and dropped onto a target and, depending on its dropped location, a *hit* or *miss* message appears in the debugger window.

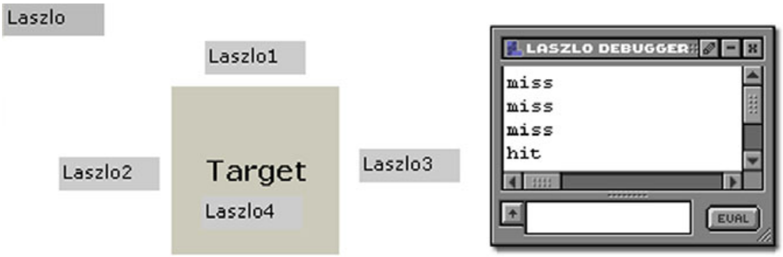
This action is initiated with an `onmousedown` event, which creates ❶ a new instance of the dragged object. This dragged object is a view-based object containing a label, an `onmousedown` event handler, and three methods—`startDrag`, `updatePosition`, and `endDrag`—to perform its drag-and-drop operations. When a new object is instantiated, its background color is lightened and a unique label identifies it.

The startDrag method is called to create the updatePos and eDrag delegates ❷ that add its dynamic drag-and-drop behavior. The updatePos delegate is registered to handle onidle events generated by the LzIdle service, to repeatedly call updatePosition.

Releasing the mouse button ❸ signals that dragging has ended. The eDrag delegate is registered to handle onmouseup events received from LzGlobalMouse. After the drop, all the delegates are unregistered ❺. Since a new delegate is instantiated only once, we don't release their resources.

The initial onmousedown event indicating the drag start is a local mouse event, as it occurs within a clickable view. However, because the dragged image can be dragged anywhere on the screen, LzGlobalMouse provides the onmouseup event indicating the drop.

To determine whether the object was dropped on the target, the final coordinates of the dragged view are compared against the target ❹. Figure 9.8 shows where three missed drops would have landed and a fourth "hit."



```
miss
miss
miss
hit
```

Figure 9.8 Dragging and dropping is used to create identical icons that are dropped onto a target. The debugger shows that there were three misses before a hit.

We have now implemented our own simple drag-and-drop class, which operates like a simplified version of Laszlo's dragstate tag. The dragstate tag can be applied to any view to make it draggable. It's derived from the state tag, so it uses the same set of apply and remove methods to turn dragging on and off. A simple example showing how dragstate can be applied to a square to allow it to be dragged across the screen looks like this:

```
<canvas>
   <view name="box" bgcolor="0xBBBBBB" width="20" height="20"
         onmousedown="dragger.apply()"
         onmouseup="dragger.remove()">
      <dragstate name="dragger"/>
   </view>
</canvas>
```

Although this works fine for simple cases, applications typically require a *network* of drag-and-drop operations that can involve multiple target areas, operations, and source items. Laszlo provides the LzTrack service to manage these types of complex drag-and-drop operations.

9.5.3 Advanced drag-and-drop with LzTrack

Advanced drag-and-drop operations are organized through a set of networking rules supplied by the LzTrack service. For example, suppose we have three views, A, B, and C, and want to impose the following set of rules:

- Items can be dragged from A to B and C.
- Items can be dragged from B to C.

but

- Items can't be dragged from B to A, C to A, or C to B.

Figure 9.9 provides a visual representation of these rules.

LzTrack provides this network through *tracking groups*, groups of views that receive mouse-tracking events. Group members now receive these *tracking events*, onmousetrackover, onmousetrackout, and onmousetrackup, which signal when the mouse—and the dragged object—enter a view, leave a view, or release a button. The methods listed in table 9.8 control the registration and activation of tracking-group membership.

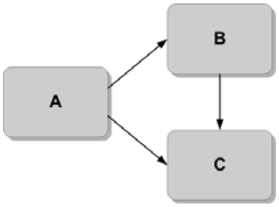


Figure 9.9 This schematic illustrates the rules of a drag-and-drop network. Items can be dragged from view A to both the B and C views. Items from view B can only be dragged to view C. Items in view C can't be dragged at all.

Table 9.8 LzTrack methods

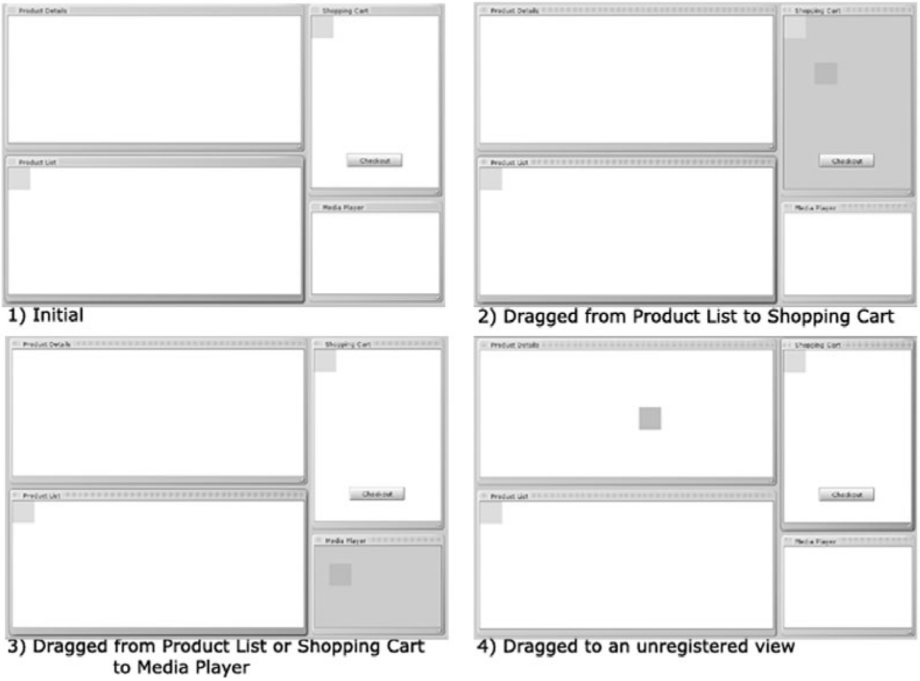
Name	Description
activate(group)	Activate tracking for a particular group. Multiple groups can be tracked simultaneously.
deactivate(group)	Deactivate tracking for the currently active group.
register(v, group)	Register a view to be tracked by a particular tracking group.
unregister(v, group)	Unregister a view from tracking by a particular tracking group.

A view will typically register itself as a member of a tracking group during its `oninit` stage, and call the `register` method with a tracking-group name. Normally, a tracking group lies dormant, but when any member receives an `onmousedown` event, it can call `LzTrack`'s `activate` method with its tracking-group name to activate it. Now all group members will receive the mouse-tracking events.

Later, if a tracking-group member receives an `onmouseup` event, it can disable all tracking events by calling `LzTrack`'s `deactivate` method with the tracking-group name. Finally, members can leave a tracking group by calling the `LzTrack`'s `unregister` method.

Drag-and-drop in the Laszlo Market

We'll use `LzTrack` in the Laszlo Market to let customers select a product item simply by dragging and dropping it from the Product List to the Shopping Cart window. Customers will also be able to view the video trailers by dragging and dropping them from the Product List window to the Media Player window. These tracking relationships are illustrated in figure 9.10.



1) Initial

2) Dragged from Product List to Shopping Cart

3) Dragged from Product List or Shopping Cart to Media Player

4) Dragged to an unregistered view

Figure 9.10 Screen 1 shows two drag source boxes in the Product List and Shopping Cart windows. In screen 2, the Product List box is dragged to the Shopping Cart. In screen 3, the Product List or Shopping Cart box is dragged to the Media Player. In screen 4, a draggable box is dragged to the unregistered Product Details window.

To represent a draggable item, two colored boxes are used, one in the Product List window and the other in the Shopping Cart. To indicate when a dragged item is over its target window, the background window color will darken. Whenever the mouse button is released or the mouse moves outside the target, the window background reverts to its original color. Since the Product Details window is not a member of any tracking group, it should not respond to a dragged item.

Since it's the simpler case, we'll start by working with the Media Player window. It receives dropped objects from two sources, so it needs to register as a member for both the `product_target` and `shop_target` tracking groups within its `oninit` handler, as shown in listing 9.10.

Listing 9.10 Registering the Media Player in tracking groups

```
<window name="mediaplayer" title="Media Player" … >
   <handler name="oninit">
      LzTrack.register(this, "product_target");
      LzTrack.register(this, "media_target");
   </handler>
   <handler name="onmousetrackover">
      this.setAttribute("bgcolor", 0xCCCCCC);
   </handler>
   <handler name="onmousetrackout">
      this.setAttribute("bgcolor", 0xFFFFFF);
   </handler>
   <handler name="onmousetrackup">
      this.setAttribute("bgcolor", 0xFFFFFF);
   </handler>
</window>
```

Becomes dark on mouse over

Becomes light again on mouse out

Becomes light again on mouse button up

The window is now eligible to receive mouse-tracking events from these groups and will change its background color depending on the received mouse-tracking event.

Next, we'll implement the Shopping Cart window. It only receives items dragged from the Product List, so it only needs to register within the `product_target` group. It will also change its background color to reflect the current mouse track state. Since the Shopping Cart window is a source for draggable objects, a small viewable object is also added to represent this source. Dragging operations are initiated by a local `onmousedown` event. Listing 9.11 shows the steps necessary to support drag-and-drop functionality in the shopping cart.

Listing 9.11 Adding drag-and-drop functionality to the shopping cart

```
<window name="shopcart" title="Shopping Cart" … >
   <handler name="oninit">
      LzTrack.register(this, "product_target");
   </handler>
   <handler name="onmousetrackover">
      this.setAttribute("bgcolor", 0xCCCCCC);
   </handler>
   <handler name="onmousetrackout">
      this.setAttribute("bgcolor", 0xFFFFFF);
   </handler>
   <handler name="onmousetrackup">
      this.setAttribute("bgcolor", 0xFFFFFF);
   </handler>
   <view width="40" height="40" bgcolor="0xDDDDDD">
      <handler name="onmousedown">
         dragger.startdrag("media_target");
      </handler>
   </view>
   …
</window>
```

Creates drag source

Begins dragging

Rather than instantiating a separate object for each draggable source, we'll create a draggable class for these transient draggable images. There will be a single global draggable instance, dragger, that can be reused. Its dimensions match those of the drag sources and its opacity is turned down a notch to provide it with a transient ghostly image:

```
<canvas>
   <draggable name="dragger" width="40" height="40"
              bgcolor="0xDDDDDD" opacity=".8"/>
   …
</canvas>
```

The draggable class is similar to our earlier examples, as it primarily consists of the startdrag and enddrag methods. The startdrag method is more complex since its drag source could potentially be registered with several LzTrack groups. It receives an argument that can either be a string, for a single LzTrack group, or an array representing multiple groups. Depending on the argument's type, one or more LzTrack groups are activated and the argument is saved in a droptarget array to make it accessible for deactivating. Listing 9.12 shows the startdrag and enddrag methods.

Listing 9.12 The `startdrag` and `enddrag` methods

```
<class name="dragger" visible="false">
   <attribute name="droptarget"/>
   <attribute name="del"/>
   <dragstate name="dragstate"/>

   <method name="startdrag" args="target">
      <![CDATA[
      this.bringToFront();
      this.setVisible(true);
      this.setX(canvas.getMouse('x') - (this.width/2));
      this.setY(canvas.getMouse('y') - (this.height/2));
      dragstate.apply();
      if (typeof target == "object") {
         this.droptarget = new Array();
         for (var i = 0; i < target.length; i++) {
            LzTrack.activate(target[i]);
            this.droptarget[i] = target[i]; } }
      else {
         LzTrack.activate(target);
         this.droptarget = target; }
      this.del = new LzDelegate(this, "enddrag");
      this.del.register(LzGlobalMouse, "onmouseup");
      ]]>
   </method>

   <method name="enddrag">
      <![CDATA[
      this.setVisible(false);
      draggerid.del.unregisterAll();
      dragstate.remove();
      if (typeof this.droptarget == "object") {
         for (var i = 0; i < this.droptarget.length; i++) {
            LzTrack.deactivate(this.droptarget[i]);
            this.droptarget[i] = null; } }
      else {
         LzTrack.deactivate(this.droptarget);
         this.droptarget = null; }
      ]]>
   </method>

   <view height="${classroot.height}" width="${classroot.width}"
         bgcolor="${classroot.bgcolor}"/>
</class>
```

❶ **Starts drag**

❷ **Handles multiple targets**

❸ **Handles single target**

Sets up for drop

Handles drop

In listing 9.12, to create the draggable icon at ❶, we make the icon visible and bring it in front of the other windows. When dragging begins, the draggable icon must be slightly offset from the background image to provide a sense of originating from it. Finally, we use the dragstate's apply method to make it draggable.

The next step ❷ supports the drag-and-drop network by determining whether this draggable icon has one or multiple drop targets. If the target argument is an array, then there are multiple targets. Each target is activated ❸ and saved for later deactivation.

When the icon is dropped onto its target, the draggable icon becomes invisible, its dragstate is removed, and it is unregistered to prevent it receiving further onmouseup events.

The drag source for the Product List window has multiple possible destinations, so it needs to register with both the product_target and media_target LzTrack groups. These targets are stored in an array to indicate to the dragger object its multiple destinations:

```
<window name="productlist" title="Product List" … >
   <handler name="oninit">
      LzTrack.register(this, "product_target");
      LzTrack.register(this, "media_target");
   </handler>
   <view width="40" height="40" bgcolor="0xDDDDDD">
      <handler name="onmousedown">
         var a = new Array();
         a[0] = "product_target";
         a[1] = "media_target";
         dragger.startdrag(a);
      </handler>
   </view>
</window>
```

Using LzTrack groups removes the need to perform any screen coordinate tests to determine a target position. Drag-and-drop sources are now connected to their targeted drop zones by logical track names. This allows target combinations to be easily updated by adding or removing members from the tracking groups.

Although we've used a simple view here, this example serves as the foundation for our drag-and-drop network. In subsequent chapters, we'll enhance this network with additional functionality. But in each case, only a few additional lines of code will need to be added to this foundation.

9.6 Summary

It isn't enough for an application to be visually appealing; its real worth is determined by how well it fits into the workflow of its users. Over the years, desktop applications have discovered a number of user interface conventions that have struck enough emotional resonance with users to have stood the test of time and be popularly accepted. Today's user can't accomplish much without a mouse, favorite keyboard shortcuts, and a scroll wheel to scan through pages. Just as important as these physical devices are the visual paradigms that define the workings of graphical interfaces. Most users would judge an application to be incomprehensible if it didn't support common graphical metaphors such as drag-and-drop, modal windows, and window selection. While many of these interface devices and visual paradigms may have started their life as "bells and whistles," they have managed to succeed and thrive in popularity because they fit so nicely into a user's workflow.

Laszlo provides a rich set of services supporting a wide array of interface devices and graphical metaphors. These services provide the fundamental set of features required to produce an application satisfying the design requirements of today's applications. These features are a starting point; we can augment them with animation, audio, and video to build future interface metaphors for tomorrow's world. In the following chapters, we'll continue to develop the drag-and-drop network by defining the composition of datasets that populate the views and provide the products available for drag-and-drop operations.

Part 3

Laszlo datasets

Part 3 is all about Laszlo's approach to data handling, all of which, with the limited exception of resources, is performed through its *data-binding system* whereby any visual object can establish a direct relationship with data. This system provides communication between the view and model layers of the MVC architecture. Most significantly, the communication is bidirectional, which along with the event-handling system, greatly enhances the versatility of Laszlo's architecture because any visual object can directly interact with any other element—object or data. The resulting integrated system allows a high degree of interactivity among all its elements. Laszlo's model-layer data repositories are called *datasets*. Chapter 10 introduces fixed bindings between a visual object and data in a dataset. Chapter 11 moves on to bindings that change during runtime. Chapter 12 incorporates datasets into the Laszlo Market.

10

Working with XML datasets

This chapter covers

- Exploring XML and datasets
- Binding visual objects to data nodes
- Introducing the replication manager
- Sorting datasets
- Using grids to create listings

> *It is a capital mistake to theorize before one has data.*
> *Insensibly one begins to twist facts to suit theories,*
> *instead of theories to suit facts.*
>
> —Sir Arthur Conan Doyle, author

You've probably been anxiously wondering when we're going to start working with data. Up to now, we've restricted ourselves to the horizontal aspects of event-delegate communications. But data requires a vertical approach that uses the services of the data-binding communication system. Later in this chapter, we'll see how the directionality of Laszlo's communication system cleanly maps onto the layout of horizontal and vertical prototype models.

Laszlo's approach to data is so comprehensive that we need four chapters to describe it completely. Since all data for a Laszlo application is encoded in XML and stored in datasets, we'll begin by exploring the structural elements of XML, followed by the XPath expressions that establish a relationship between LZX and data. In the next chapter, we'll explain how to manipulate the contents of datasets. In chapter 16, we'll introduce buffered datasets for interfacing to HTTP servers. Finally in chapter 17, we'll cover the special handling requirements needed to efficiently work with large datasets.

Laszlo's method of data interaction involves a twist on conventional data access methods. Although its unique qualities make its initial learning curve steeper, it rewards this effort with an elegant system that allows complex data interactions to be easily described. Since all data is persistently stored in datasets, data is never digested. Instead, visual objects establish persistent relationships to individual or multiple data elements within a dataset. Because datasets are stored in memory, their persistence lasts only for the application's duration. In chapter 16, we'll finally see how to transfer data to a web server.

10.1 Introducing XML-based datasets

In most systems, code manages data. Laszlo reverses this—data governs the presentation of coded objects. Although a visual object initiates a binding to data, once the binding is established the object cedes control of its presentation to the data. This may seem odd at first, but consider the appealing aspects of this relationship:

- When there is no data, nothing is displayed.
- When a data node exists, a visual object is displayed.
- When multiple data nodes exist, multiple visual objects are displayed.

This results in data changes being automatically reflected in the visual presentation. When new data nodes appear, visual objects spontaneously appear. When data nodes are deleted, these same objects spontaneously disappear, thus eliminating the display of stale data.

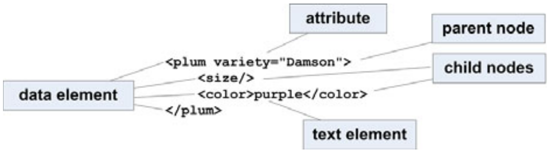
All data access in Laszlo, with the exception of media resources, is performed through its *data-binding* communication system. Just as event-based communications allow multiple listeners for a single sender, data bindings allow multiple visual objects to be bound to a data node. This uses the publisher-subscriber communication system discussed in chapter 8, but it is now applied vertically to data-bound visual objects. The XML data node operates as the publisher and the bound objects are the subscribers. Any changes to a data node's properties are immediately communicated to all its listeners. But since each object can handle changes differently, this supports complex user interface interactions.

The *data-binding* communication system differs from the event-handling system by allowing bidirectional communication between publishers and subscribers. This gives subscribers a way to directly update the XML contents of a dataset that compels the publisher to issue new events to its subscribers. Furthermore, visual objects can now communicate horizontally with one another through object-based attribute settings or vertically through the application state represented by data elements in a dataset.

We'll begin by defining the composition of an XML data structure in a dataset. Next we'll examine the different entities that comprise a query in an XPath expression. Then we'll demonstrate how a visual object can use an XPath expression to establish a data binding to a location in an XML dataset. Depending on the XPath query and the composition of the XML dataset, the XPath expression can return zero, one, or multiple matches. While the first two cases are handled in a standard manner, handling multiple matches introduces the concept of replication. Later we'll apply replication to sorting. At the end of the chapter, we'll use these concepts in the Laszlo Market to create a product listing. This demonstrates how a window can easily be produced to contain a listing featuring alternating backgrounds, selections, scrolling, column titles, and sorting.

10.1.1 Exploring XML elements

Laszlo's approach to data access is grounded in two widely accepted standards: XML and XPath. A Laszlo application can have an unlimited number of datasets, where each dataset contains an XML document. A valid XML document consists of a hierarchical tree structure with a *root node* containing any number of *child nodes*, each of which, following the usual recursive definition of a tree, can contain other



```
<plum variety="Damson">
   <size/>
   <color>purple</color>
</plum>
```

Figure 10.1
In this XML code fragment, the `plum` data element contains two other data elements, `color` and `size`, and has an attribute `variety`. The data element `color` contains the text element `purple`.

child nodes. Figure 10.1 shows an example with the root node `plum` and its two child nodes, `size` and `color`. The node `color` has one child node, the text node containing the `purple` string.

Laszlo uses a nonstandard terminology to describe an XML structure. While the World Wide Web Consortium (W3C) only defines XML in terms of data elements, Laszlo adds a *text element* definition. Data elements can have any number of attributes and child nodes, but a text element consists only of text data and must be a child node of a data element.

A dataset can be initially stocked with data from a resident or networked source. In this chapter, we'll only work with resident datasets. In chapter 16 we'll see how a resident dataset can easily be converted into a remote HTTP-accessed dataset.

Sample dataset for examples

Listing 10.1 shows a resident dataset contained in the myData.lzx dataset. This dataset has a single root node called `fruit` and four child nodes: `peach`, `plum`, `peach`, and `cherry`. Each data element has a single attribute named `variety`. Each child element contains a set of `color` and `size` child elements. This sample dataset is used for all examples in this chapter and chapter 11.

Listing 10.1 Contents of the myData.lzx library file

```
<library>
  <dataset name="myData">
    <fruit type="drupe">
      <peach variety="Freestone">
         <color>white</color>
         <size>medium</size>
      </peach>
      <plum variety="Damson">
         <color>purple</color>
         <size>small</size>
      </plum>
      <peach variety="Clingstone">
         <color>yellow</color>
         <size>large</size>
      </peach>
```

```
        <cherry variety="Bing">
           <color>red</color>
           <size>small</size>
        </cherry>
      </fruit>
    </dataset>
  </library>
```

The contents of this dataset can be included into an application with this tag:

```
<include href="myData.lzx"/>
```

Now that we have a dataset, we need a way to access it. For this, we'll need to use XPath expressions.

10.1.2 Using XPath to select data elements

XPath is a compact query language for selecting elements or element sets from an XML document. Laszlo supports only a subset of XPath, but its descriptive power is sufficient for our purposes. The location path notation is designed to mimic the Uniform Resource Identifier (URI) and common file path syntaxes. All of the XPath *path expressions* that we'll be using have the following form: a *location path* followed by a *wildcard, predicate,* or *function.*

A location path identifies the dataset by specifying its name followed by a colon and forward slash. Appending a root element name to the location path selects the dataset's root element. Further appended element names following a forward slash select subelements. A location path selects a data element and establishes a *context.* This context can be further refined by specifying a predicate. Table 10.1 summarizes the syntax of location paths.

Table 10.1 XPath location path forms

Location Path Form	Description	Example Path Expression	Example Result
`datasetname:/`	Selects a dataset	`myData:/`	`<myData>…</myData>`
`datasetname:/ root`	Selects the root element of the dataset	`myData:/fruit`	`<fruit>…</fruit>`

`datasetname:/ root/…/elementN`	Selects `elementN`	`myData: /fruit/plum`	`<plum variety="Damson">`
			`…`
			`</plum>`

A predicate narrows a set of matching data elements to a subset according to the different forms shown in table 10.2. It can select a single element using one-based indexing, or a group of elements by using a range. JavaScript uses zero-based indexing, so this is the only place where one-based indexing is used in Laszlo. A range extends from a start to an indicated element, from an indicated element to the end, or within a specified range. Finally, the `@` operator can be used to specify a matching attribute name. In each case, the context is updated. This context serves as a base and a function is used to select the various data contents within the specified data elements.

As table 10.3 shows, Laszlo only supports a single wildcard selector, the asterisk (`*`), to select all elements. None of the other popular wildcard characters, such as the question mark (?) or plus (+), are supported. The purpose of the wildcard selector is to select all of a data element's children.

Table 10.2 XPath predicates

Predicate Form	Description	Example Path Expression	Example Result

`[i]`	Selects the i^{th} element	`myData:/fruit/peach[1]`	`<peach variety="Freestone">`
			`...`
			`</peach>`

`[-i]`	Selects up to and including the i^{th} element	`myData:/fruit/peach[-2]`	`<peach variety="Freestone">`
			`...`
			`</peach>`
			`<peach variety="Clingstone">`
			`...`
			`</peach>`

`[i-]`	Selects from the i^{th} element to the last element	`myData:/fruit/peach[2-]`	`<peach variety="Clingstone">`
			`...`
			`</peach>`

`[i-j]`	Selects from the i^{th} element to the jth element	`myData:/fruit/peach[1-2]`	`<peach variety=”Freestone”>`
			`...`
			`</peach>`
			`<peach variety="Clingstone">`
			`...`
			`</peach>`

`[@name='value']`	Selects an element with a specified attribute containing a specified value	`myData:/fruit/peach[@variety=‘Freestone]`	`<peach variety="Freestone">`
			`...`
			`</peach>`

Table 10.3 XPath wildcard selectors

Wildcard	Description	Example Path Expression	Example Result

`*`	Selects all contained element(s)	`myData:/fruit/plum/*`	`<color>purple</color>`
			`<size>small</size>`
			`<price>1.25</price>`

Finally, we get to *terminal selectors* that access the data contained within a data element. A terminal selector can be any of the operators or functions listed in table 10.4. They can return a data element's attribute value, text value, or name.

Table 10.4 XPath terminal selectors

Function Form	Description	Example Path Expression	Example Result
`@attribute`	Displays the name of the specified attribute	`myData:/fruit/plum/@variety`	`Damson`
`text()`	Displays the contents contained within the text element	`myData:/fruit/plum/size/text()`	`small`
`name()`	Displays the name of an element	`myData://fruit/cherry/name()`	`cherry`

Laszlo supplies the `datapath` attribute to contain these XPath expressions within a declarative tag. Because this attribute is defined by the `LzNode` object, every LZX object has access to it. As a result, any declarative tag can use XPath to access datasets.

10.1.3 Binding declarative tags to XML elements

A *data path* embeds an XPath expression in a visual tag to bind it to a dataset's XML elements. The behavior of a data path depends on whether the XPath expression returns a single matching data element, multiple data elements, or no elements. Let's start with the simplest case, a single matching XML element.

An XPath expression that returns no match or a single match governs the visibility of its bound view object. This visibility relationship can be turned off by resetting a view's `dataControlsVisibility` attribute to false; it defaults to true. A match represents a returned data, attribute, name, or text element. A match also returns a context that is propagated to all child nodes. This allows child nodes to execute relative XPath expressions to retrieve the data values in a data element. In this way, the parent's returned value also controls the visibility of its child nodes. So if the contents of the dataset change so that the XPath expression no longer returns a match, then the bound object loses its visibility, as does its children.

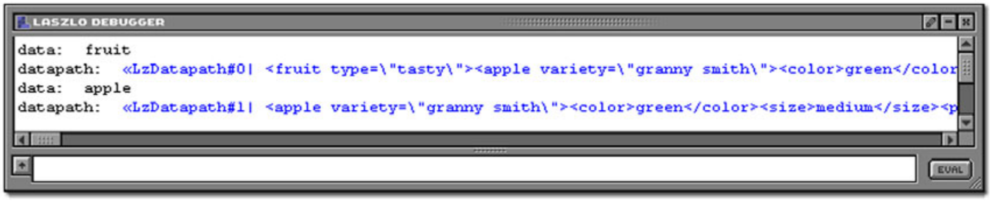
When an XPath expression returns a single matching data element, a pair of data and LzDatapath objects is automatically created and an ondata event is generated. The data object contains the matched value, while the LzDatapath object contains, among other things, a pointer to the current location or data context within the XML dataset. Listing 10.2 illustrates how these data and LzDatapath objects can be displayed in the ondata event handler.

Listing 10.2 Using a data path to bind a tag to an XML element in a dataset

```
<canvas debug="true">
   <include href="myData.lzx"/>

   <node datapath="myData:/fruit/@type">
      <handler name="ondata">
         Debug.write("data: ", this.data);
         Debug.write("datapath: ", this.datapath);
      </handler>
   </node>
   <node datapath="myData:/fruit/peach[1]/@variety">
      <handler name="ondata">
         Debug.write("data: ", this.data);
         Debug.write("datapath: ", this.datapath);
      </handler>
   </node>
</canvas>
```

Figure 10.2 shows the debugger output of the data attribute displaying the type and variety attributes for the fruit and peach data elements, along with a set of pointers to the dataset whose position reflects each match within the XML structure.

Since each of these XML element nodes has child nodes containing data elements, the data context can be used to access the data contained within these child data elements.



```
data:   fruit
datapath:   «LzDatapath#0| <fruit type=\"tasty\"><apple variety=\"granny smith\"><color>green</color
data:   apple
datapath:   «LzDatapath#1| <apple variety=\"granny smith\"><color>green</color><size>medium</size><p
```

Figure 10.2 A declarative tag uses data and datapath attributes to bind to an XML data element. These objects are automatically created to store the information retrieved by the XPath query. The data attribute stores the matching data value returned by the XPath expression. The datapath attribute acts like a pointer to the XML dataset to provide a data context.

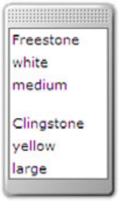
10.1.4 Establishing a data path context

The data context established by a parent node is available to its children. A child node can extend its parent's context with relative XPath addressing to add a terminal selector to access subsequent data elements. This approach provides a succinct way to access and display a sequence of data elements:

```
<canvas>
   <include href="myData.lzx"/>

   <window>
      <view datapath="myData:/fruit">
         <text datapath="peach[1]/@variety"/>
         <text datapath="peach[1]/color/text()"/>
         <text datapath="peach[1]/size/text()"/>
         <simplelayout axis="y" spacing="2"/>
      </view>
      <view datapath="myData:/fruit/peach[2]">
         <text datapath="@variety"/>
         <text datapath="color/text()"/>
         <text datapath="size/text()"/>
         <simplelayout axis="y" spacing="2"/>
      </view>
      <simplelayout axis="y" spacing="15"/>
   </window>
</canvas>
```

Here the view's data path is used to establish a context, which is extended by its children with terminal selectors to access its contained data. The results are shown in figure 10.3.

We've now seen how a data path's XPath expression can be split across parent and child nodes. Let's move on to an even more general approach.



Freestone
white
medium

Clingstone
yellow
large

Figure 10.3 Once a context has been established by a parent, it is available to all its children. A child node can extend its parent's context with operators to access the data contained in a data element.

10.1.5 The $path{} constraint notation

Having a bidirectional communication system isn't terribly useful, unless there is a way for these systems to interact with one another. Once again, constraints are used to provide the glue allowing the event-delegate and data-binding communication systems to interact. An object's attributes can be set with a special *path constraint* that works with data paths. It requires that a data path context first be established; afterward, relative XPath addressing with a terminal selector is used to access individual data elements. Listing 10.3 shows how one view can communicate values to another view using event constraints that transmit values set through data-bound path constraints.

Listing 10.3 Showing how constraints and path constraints can interact

```
 <canvas>
   <dataset name="myData">
       <width>100</width>
       <height>100</height>
   </dataset>
   <view name="main" datapath="myData:/" bgcolor="0xCCCCCC">
      <attribute name="width"   value="$path{'width/text()'}"/>
      <attribute name="height"  value="$path{'height/text()'}"/>
   </view>
   <view name="another" width="${main.width}" height="${main.height}"
                                          bgcolor="0xDDDDDD"/>
   <simplelayout spacing="5"/>
</canvas>
```

The purpose of this example is to demonstrate the general equivalence of these communication systems and not to serve as an example of programming style. But for certain applications that dynamically generate a display, storing presentation values in a dataset might be an appropriate coding methodology.

Another major difference between these two communication systems is that data paths and path constraints operate in a two-way manner. This allows an object to write or update the elements of an XML dataset. In the next section, we'll see how an object can transfer updated data elements back to an XML dataset.

10.1.6 Updating a dataset

To complete this circuit, the `datapath` object's `updateData` method allows an object to transfer its updated value back into the bound data element. Whenever an XML element in a dataset is updated, this results in an `ondata` event, which causes any bound objects to be updated. This completes a communication loop between visual objects, as one object can update a data element, which results in other bound views automatically having their displays updated with this new information. Listing 10.4 demonstrates this process using a pair of `width` and `height` input fields.

Listing 10.4 Updating a dataset

```
<canvas>
   <include href="/incubator/formlayout.lzx"/>
   <dataset name="myData">
       <width>100</width>
       <height>100</height>
   </dataset>
```

Sets initial XML data value

```
    <view datapath="myData:/">
       <text text="Width"/>
       <edittext datapath="width/text()" doesenter="true">
          <method name="doEnterDown">
             this.datapath.updateData(this.getText());
          </method>
       </edittext>
       <text text="Height"/>
       <edittext datapath="height/text()" doesenter="true">
          <method name="doEnterDown">
            this.datapath.updateData(this.getText());

          </method>
       </edittext>
       <formlayout inset="30" align="right"/>
    </view>
    <view name="box" x="150" y="20"
          datapath="myData:/" bgcolor="0xCCCCCC">
       <attribute name="width"
           value="$path{'width/text()'}"/>
       <attribute name="height"  value="$path{'height/text()'}"/>
    </view>
 </canvas>
```

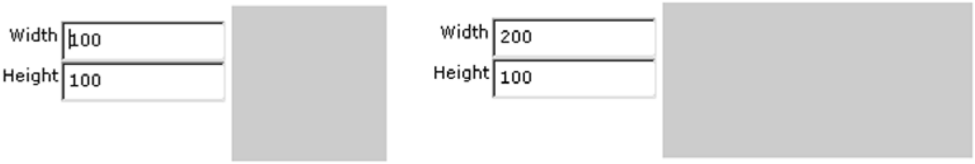
❶ Establishes data path

❷ Updates XML dataset with new value

❸ Updates width attribute with new XML value

When a user enters values into these fields and presses Enter ❶, the `doEnterDown` method is invoked to access the new value ❷ and update the `width` and `height` data elements in the `myData` dataset ❸.

The view named `box` has its `width` and `height` attributes bound to these data elements through path constraints. Since everything is bound together, when the data elements are updated with new values the dimensions of the box are also updated. This allows the values entered into the input fields to control the dimensions of the box. Figure 10.4 shows the result.



Width 100
Height 100

(Initial)

Width 200
Height 100

(After updating width field)

Figure 10.4 The label and input fields are bound to the same XML data element. When the XML data element is updated, its new value is displayed by both tags.

Next we'll tackle text objects, which are used so frequently that Laszlo adds some optimization features.

10.1.7 Handling ontext events

One useful way to work with data paths is with bound text objects for displaying character strings from XML data elements. However, this introduces some timing complexities. When an XPath match is returned, it generates an `ondata` event. Unfortunately, this occurs before the instantiation of the `LzText` object has completed. If an `ondata` event handler performs postprocessing on the `text` attribute, those results will be overwritten by the instantiation of the `LzText` object. To address this, Laszlo provides a special `ontext` event for working with text objects. Listing 10.5 illustrates this problem.

Listing 10.5 Using the `ontext` event

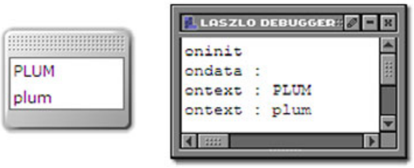
```
<canvas>
   <include href="myData.lzx"/>

   <window datapath="myData:/fruit/plum">
      <text datapath="name()">
         <handler name="ontext">
            this.setText(text.toUpperCase());
         </handler>
      </text>
      <text datapath="name()">
         <handler name="ondata" args="d">
            Debug.write("ondata : " + this.text);
            this.setText(d.toUpperCase());
         </handler>
         <handler name="oninit">
            Debug.write("oninit");
         </handler>
         <handler name="ontext">
            Debug.write("ontext : " + text);
         </handler>
      </text>
      <simplelayout axis="y" spacing="2"/>
   </window>
</canvas>
```

Overwrites text value set by ondata

Writes initial text value

The expected result for the example in listing 10.5 is a matching set of uppercase `PLUM` text strings. But as you can see in figure 10.5, the text string set in the `ondata` event handler is still in lowercase. This indicates that the text object later overwrote this value. Bottom line, ontext events should be used in preference to ondata events with text objects. The timing sequence of the events is shown in figure 10.5.

Laszlo also provides another more general and efficient method, applyData, to perform postprocessing for any object.



```
PLUM
plum
```

```
oninit
ondata :
ontext : PLUM
ontext : plum
```

Figure 10.5 The intended output here is a matching pair of uppercase names. However, the ondata change to uppercase is overwritten by the text instantiation. The timing sequence for the instantiation of the text object appears in the debug window.

10.1.8 Updating with the applyData method

You've seen how event handlers are implemented with event and delegate objects, and the benefits of their associated loose coupling. However, since it is not uncommon to have a large number of bound text objects in a form, with each served by an ontext event handler, the savings from not instantiating an equal number of delegates can be significant. The applyData method, which is directly callable, saves a delegate object from being instantiated for each text object.

Because applyData saves system resources, it is the preferred way to handle events for data-bound visible objects. Furthermore, its call occurs after object instantiation, so there are no timing issues. When an applyData method is supplied, it is called when an XML dataset is instantiated and when an associated data path's matching data element is updated.

The following code reimplements the previous example to illustrate the operation of the applyData method and to demonstrate that it doesn't suffer from timing problems with instantiating objects:

```
<canvas>
   <include href="myData.lzx"/>

   <window datapath="myData:/fruit/plum">
      <text datapath="name()">
         <method name="applyData" args="d">
            this.setText(d.toUpperCase());
         </method>
      </text>
   </window>
</canvas>
```

Figure 10.6 shows that the postprocessing to uppercase occurred correctly.

Next, we'll look at a specialized type of resident dataset.

10.1.9 Using local datasets

We've been using the myData dataset as a top-level element. Although this is the default location for datasets, they can be further localized with a definition in a container. Local data is useful for containing static data. For example, it can be used to list the states in the United States, so that the data is conveniently located close to display components. Local datasets can only be referenced with dot notation relative to a parent node. Also local datasets must be specified explicitly; they can't be in an included LZX dataset or XML document. The previous example can be expressed using a local dataset like this:

```
<canvas>
   <window>
      <dataset name="myData">
         <fruit type="drupe">
            …
            <plum variety="Damson">
               <color>purple</color>
               <size>small</size>
            </plum>
            …
         </fruit>
      </dataset>
      <view datapath="local:parent.myData:/fruit/plum">
         <text text="$path{'name()'}"/>
      </view>
   </window>
</canvas>
```

This produces the same display shown in figure 10.6.

The next obvious question is, what happens when an XPath expression matches a multiple number of data elements? This seemingly innocuous question opens the door to a world populated with exotic objects such as replicators, clones, and nodes that seem to spontaneously appear out of nowhere. But once you understand them, you'll see how they seem to magically simplify your development tasks.

Figure 10.6 The applyData method provides a means to preprocess XML data elements. In this example, the input field is converted to uppercase.

10.2 Matching multiple data elements

Processing dramatically changes when a data path matches multiple data elements. When multiple matches are returned, ordinary objects aren't able to handle this situation and an LzReplicationManager object, often just called a *replication manager*, is automatically instantiated to serve as an intermediary. The replication manager uses the original object as a template to instantiate a number of cloned copies equal to the number of matched XML elements. The process is illustrated in figure 10.7.

Until now, we have worked only with data paths returning a single match, reflected in the left side of figure 10.7, which produces a single output. The `myData` dataset contains multiple XML elements named `peach` to demonstrate the replication manager services depicted on the right side of figure 10.7.

When an object's data path returns multiple matching XML elements, the original object is destroyed and a replication manager with the same `name` or `id` attribute is instantiated in its place. Now, the `name` or `id` references the `LzReplicationManager` object. Regardless of the replication manager's assumed name, clones can always refer to it as `cloneManager`.

This replication manager is responsible for instantiating two groups of objects; the size of each group is determined by the number of matching XML elements. The first group, called the *clones*, contains identical copies, including the `name` or `id` attribute, of the original destroyed object. The second group, called the *nodes*,

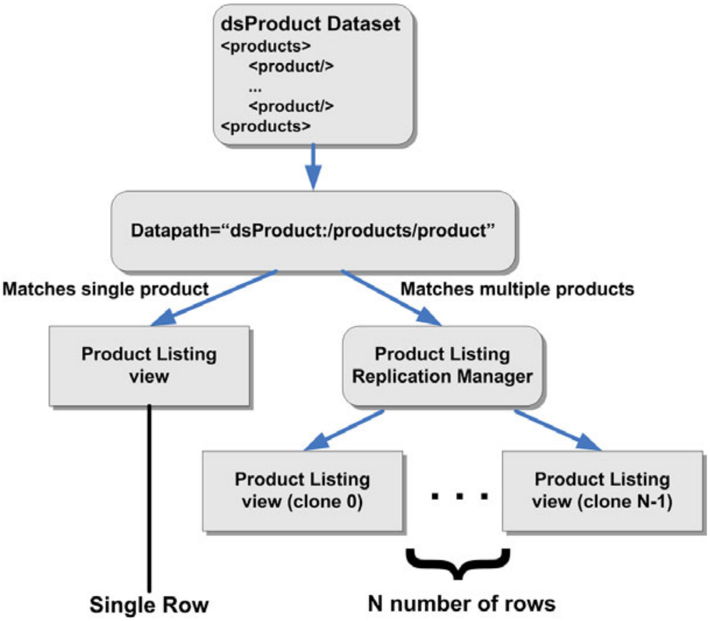

dsProduct Dataset
<products>
<product/>
...
<product/>
<products>
Datapath="dsProduct:/products/product"
Matches single product
Matches multiple products
Product Listing view
Product Listing Replication Manager
Product Listing view (clone 0)
. . .
Product Listing view (clone N-1)
Single Row
N number of rows

Figure 10.7 Depending on the number of matching data elements, a data path either displays one corresponding view or, for more than one match, instantiates a replication manager to control the replicated views, sometimes known as clones.

contains the matching XML elements. Tables 10.5 and 10.6 summarize the attributes and methods of the replication manager.

Table 10.5 Replication manager attributes

Name	Data Type	Attribute	Description
Clones	array of `LzNodes`	Read-only	The `LzNodes` that this replication manager has created
Nodes	string	Read-only	The data nodes that map to the clones

Table 10.6 Replication manager methods

Name	Description
`getCloneForNode(node)`	Returns a clone that is mapped to the given data node
`getCloneNumber(n`th `clone)`	Returns a pointer to the nth clone controlled by the replication manager
`setVisible`	Sets the visibility of all clones controlled by the replication manager

There is no rank or priority within clones; all are created equal. These replicated objects can no longer be directly accessed by their `name` or `id`; instead, all accesses must go through their replication manager. To access a particular replicated object, the replication manager is accessed and a particular clone is indexed in the clones array. Similarly, a particular matching XML element is selected through the replication manager's nodes array.

Listing 10.6 illustrates the relationship between a replication manager and its clones. Since the debug messages would normally be displayed multiple times, for each clone, a flag is added to limit their display.

Listing 10.6 Relationship between replication manager and clones

```
<canvas debug="true">
   <include href="myData.lzx"/>
   <attribute name="already_displayed" type="boolean" value="false"/>

   <text id="main" datapath="myData:/fruit/peach">
       <handler name="ondata">
           if (canvas.already_displayed) return;
           Debug.write("main=" , main);
           Debug.write("cloneManager=" , this.cloneManager);
```

Accesses replication manager by ID

Accesses replication manager by name

```
            Debug.write("cloneManager.clones=",
                         this.cloneManager.clones);
            Debug.write("cloneManager.nodes=" + this.cloneManager.nodes);
            Debug.write("subviews : " , canvas.subviews);
            canvas.already_displayed = true;
        </handler>
    </text>
    <simplelayout axis="y" spacing="2"/>
</canvas>
```

Accesses clones array

Accesses individual clones

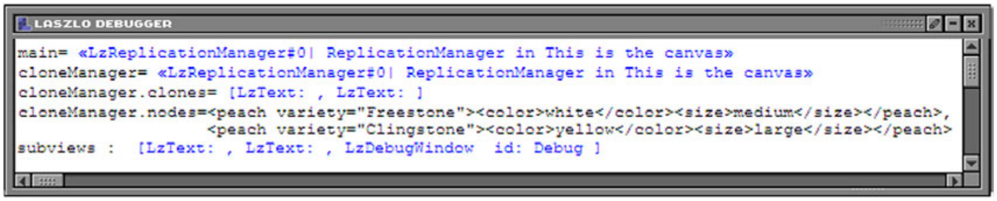
Displays clones

Each object in the clones array is indexed by a *clone number.* A particular clone can by accessed by zero-based array indexing or through the `getCloneNumber` method. The debugger window contents, shown in figure 10.8, are repeated for each replicated object. In our case, there are only two occurrences, one for each `peach`.

Although clones are directly accessible only through their replication manager, they are still listed in the subviews array of their parent node. This allows cloned objects to be configured with a `layout` tag. So, while updating cloned views requires a little more effort, going through the `cloneManager` they display normally, just like other visual objects. As a result, a `simplelayout` tag can handle the layout for any number of cloned objects.

Since the production of clones and nodes is driven by XML elements, which are supplied by outside sources, this whole cloning replication process needs to be carefully managed, since it has the potential to overwhelm the system with resource consumption. The replication manager is equipped with facilities to handle this situation. In chapter 17 you'll see how pooling and a "lazy" replication manager are used to efficiently handle large datasets.

Now that we have datasets consisting of multiple commonly named elements or sibling elements, we want to be able to sort them in either ascending or descending order.



```
main= «LzReplicationManager#0| ReplicationManager in This is the canvas»
cloneManager= «LzReplicationManager#0| ReplicationManager in This is the canvas»
cloneManager.clones= [LzText: , LzText: ]
cloneManager.nodes=<peach variety="Freestone"><color>white</color><size>medium</size></peach>,
                   <peach variety="Clingstone"><color>yellow</color><size>large</size></peach>
subviews :  [LzText: , LzText: , LzDebugWindow  id: Debug ]
```

Figure 10.8 A replication manager is created by a data path when multiple matching data elements are returned.

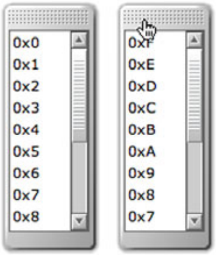
10.3 Sorting datasets

Laszlo provides simple and custom sorting for datasets consisting of sibling elements. Sorting defaults to a dictionary sort, whereby numeric digits precede letters and lower class precedes upper. To get a numeric sort—so that *7* precedes *11*—it's necessary to create a numeric sorting method. Sorting occurs on a single key, unless a custom sort method is created to handle multiple keys. All the standard sorting operators are specified with `datapath` attributes and methods. Since every viewable object has a `datapath`, this allows advanced sorting techniques to be used by all viewable objects.

10.3.1 Simple sorting

Let's start with an example illustrating a simple sort on a dataset. The key for sorting is specified by the `sortpath` attribute. Using this attribute requires that the `datapath` be separated from its parent text object:

```
<canvas>
   <dataset name="numbers">
      <num>0x3</num><num>0x6</num><num>0x8</num><num>0x7</num>
      <num>0xA</num><num>0x1</num><num>0x5</num><num>0x4</num>
      <num>0x9</num><num>0xB</num><num>0x0</num><num>0xD</num>
      <num>0xE</num><num>0x2</num><num>0xC</num><num>0xF</num>
   </dataset>
   <window width="80" height="200">
      <view height="100%" width="100%" clip="true">
         <view width="100%">
            <simplelayout axis="y"/>
            <text>
               <datapath xpath="numbers:/num/text()"
                    sortpath="text()"/>
            </text>
         </view>
         <scrollbar/>
      </view>
   </window>
</canvas>
```



0x0
0x1
0x2
0x3
0x4
0x5
0x6
0x7
0x8

0xF
0xE
0xD
0xC
0xB
0xA
0x9
0x8
0x7

Figure 10.9 Sorting order can be either ascending or descending, controlled with the `sortorder` attribute.

Because replication occurs on the `num` dataset element, the `sortpath` attribute works together with the XPath expression to specify the sort key. By default, sorting occurs in ascending order. Figure 10.9 shows the output.

Sorting order is controlled with the `sortorder` attribute, set with a value of either `descending` or ascending, or with a function name to perform a custom sort. The datapath tag shown here generates the second window displayed in figure 10.9, where the data is shown in descending order:

```
<datapath xpath="myData:/list/*/text()"
          sortpath="text()"
          sortorder="descending"/>
```

Now that you've seen simple sorting in ascending and descending order, we'll explain how to use sortorder with a function name.

10.3.2 Custom sorting

Laszlo uses a common convention, found on many other systems, to support customized sorting. Alphanumeric is supported by default while customized sort routines, also known as *sort comparators*, can be supplied with an algorithm for customized sorting.

To perform a custom sort, a method name, contained in a constraint, is specified to the sortorder attribute. All sorting methods accept two input arguments, a and b. These arguments are evaluated, with the values 0, 1, and –1 returned to reflect respectively the results: a == b, a > b, and a < b.

For example, suppose that a dataset is changed to contain the numeric word names one, two, and three, with the contents to be sorted in that order. A custom method named wordsort could be supplied to provide this sorting algorithm (see listing 10.7).

Listing 10.7 Custom sorting with the sort comparator `wordsort`

```
<canvas>
   <dataset name="numbers">
      <num>two</num><num>three</num><num>one</num>
   </dataset>
   <window x="10" y="10" width="100" height="100">
      <view>
         <simplelayout axis="y"/>
         <text>
            <datapath xpath="numbers:/num/text()" sortpath="text()"
                      sortorder="${this.wordSort}">
               <method name="wordSort" args="a,b">
                  <![CDATA[
                  var stat = 0;
                  if (a == b) return 0;
                  else if (a == "one") return 1;
                  else if (a == "two" && b == "three") return 1;
```

Uses constraint to set sorted values

Uses custom sorting algorithm

```
                    else return -1;
                    ]]>
                 </method>
              </datapath>
           </text>
        </view>
     </window>
  </canvas>
```

Specialized sorting criteria can also be established with the setComparator and setOrder methods, allowing the attribute settings sortpath and sortorder to be specified in a method format. A setComparator example might look like this:

```
<datapath xpath="myData:/num/text()" sortpath="text()">
   <handler name="oninit">
            this.setComparator(this.wordSort);
   </handler>
   …
</datapath>
```

The setOrder method might look like this:

```
<datapath xpath="myData:/num/text()">
   <handler name="oninit">
            this.setOrder("text()", this.wordSort);
   </handler>
   …
</datapath>
```

Each of these methods provides a specifier for the custom wordSort algorithm and produces the output shown in figure 10.10.

Although we have shown sorting only on a single key, multiple-key sorting can be performed by setting the sortpath to return a parent node. Later, the child nodes can be accessed in the sorting method to handle more complex sorting. An example of multiple-key sorting with setNodes is shown in a later section.

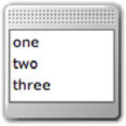

one
two
three

Figure 10.10 The custom sorting algorithm in the wordSort method produces this sorted output.

Now that we've examined Laszlo's interface to XML datasets, we're ready to return to the Laszlo Market to use some of these new tools.

10.4 Prototyping datasets for the Laszlo Market

Up to this point, we have been working from a horizontal prototype perspective, focusing strictly on visual layout issues, supported by only a thin veneer of functionality. The goal was a preliminary sketch of the application without getting bogged down in implementation details. However, the prototype has now advanced to the point where further progress requires a minimum implementation.

The complementary approach of vertical prototyping, illustrated in figure 10.11, provides such a thin sliver of implementation. In our case, this requires defining a dataset whose contents are displayed in the Product List window. The intersection of these two orthogonal prototypes is known as a *scenario prototype*, in this case a "list products" scenario.

With most development environments, supplying test data for prototype or application development involves a substantial vertical prototype infrastructure, either a system of mock data objects or an interface to a database supplied with test objects. But a vertical prototype infrastructure is only an overhead expense to support the continued development of the interface. Consequently, we want the most lightweight mechanism available for this vertical functionality. But we also want the option to reuse infrastructure elements in a final implementation.

Laszlo's datasets are ideal for all these needs. Since datasets are standard objects in Laszlo, the cost of stocking them with data is minimal. And because they can be either local or accessed using HTTP, we can use resident datasets

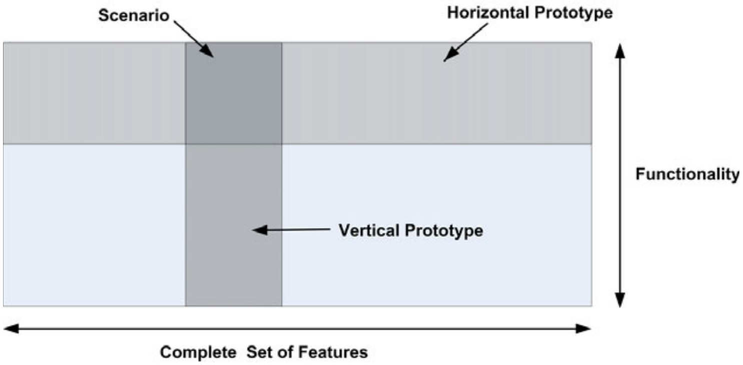


Figure 10.11 Normally, the main focus of prototype development concerns building a horizontal prototype. However, certain features require a supporting vertical prototype for their demonstration. These areas are known as scenario prototypes, because they are full-featured.

during development and then switch to HTTP datasets for deployment.

Incorporating sample data marks another step in our development framework, illustrated in figure 10.12. Although we're working at a prototype stage, until we advance to an API for communicating with a back-end server, we'll now graduate from simple horizontal prototyping to scenario prototyping. Now prototype development encompasses both visual display and the accompanying data composition required to support it.

A disciplined progression from prototyping to final implementation saves a considerable amount of work. Let's next move on to design a product dataset for the Laszlo Market.

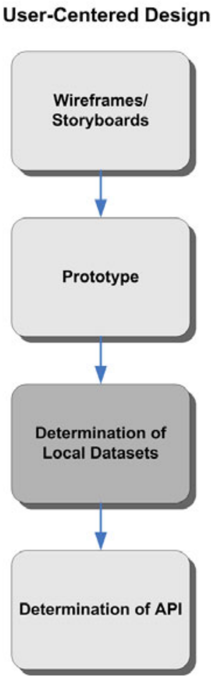


Figure 10.12 In our top-down development framework, we have advanced to the next stage of including local datasets in the prototype.

10.4.1 Designing a dataset

Designing a dataset involves a number of significant decisions. The data-bound nature of Laszlo objects causes data layout to impact the design of the application.

We'll start with a `dsProducts` dataset featuring a `products` parent node containing any number of `product` child nodes:

```
<dataset name="dsProducts">
   <products>
      <product/>
   </products>
</dataset>
```

Each product requires a unique identifier, known as a *stock-keeping unit* (SKU). We need to determine whether the `sku` should be represented as an attribute or a child node of the product. To retrieve a specific data node requires that it have a distinguishing characteristic or attribute; having a specific child node isn't a distinguishing characteristic. For this reason, Laszlo's XPath predicates have only a single equality operator, which works only with attributes.

Attributes are faster to process than child nodes, since there is no need to traverse the XML structure. When processing a large number of data nodes, the saving can be significant. However, one disadvantage of attributes is that they can contain no XML special characters. Since a product's description, technical specifications, and outline fields can contain embedded HTML tags—to italicize a word, for example—they must be represented as text nodes. The result is a `product` data element that looks like this:

```
<dataset name="dsProducts">
   <products>
      <product sku="SKU-001" title="The Unfold"
               price="19.99" image="dvd/spidermen_2.png">
          <description> A disaffected musician receives a phone …
          </description>
         <specs><p>Regional Code: 1 (US and Canada)<br>Languages: … </specs>
          <outline><b>The Unfold:</b><br/>A Swarm of Angels  … </outline>
      </product>
   </products>
</dataset>
```

This high-fidelity wireframe, shown in figure 10.13, provides the physical layout of the contents for each entry in the product list.

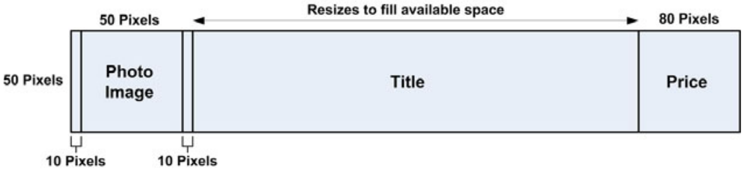


Figure 10.13 High-fidelity wireframe describing the physical layout of each item in the Product List window

To implement this wireframe, we'll create a `productlist` class to be contained in the existing window component. An instance of this class is displayed for each matching data node:

```
<window title="Product List" width="55%" height="50%">
   <productlist width="100%" height="75"
      datapath="dsProducts:/products/product"/>
   <simplelayout axis="y" spacing="2"/>
</window>
```

Since the `productlist` class contains the product list, the `datapath` must be attached to an instance of it. If the `datapath` were attached to the `window` container itself, then the entire Product List window would disappear when there were no data elements to display.

The implementation for the `productlist` class, shown in listing 10.8, contains the constituent elements specified by the wireframe. The class consists of these elements: a `view` for containing the photo image, `text` for the title, and `text` for the price. Since the two end pieces have a fixed size, and the title is variable sized, these layout requirements are tailor-made for a `stableborderlayout`.

Listing 10.8 Defining the `productlist` class

```
<class name="productlist" fontsize="12"
       fontstyle="bold" bgcolor="0xDDDDDD">
   <thumbnail height="75"
              datapath="image/text()">
      <method name="applyData" args="image">
         this.setSource(gController.IMAGESFOLDER + image);
      </method>
   </thumbnail>
   <text valign="middle" datapath="title/text()"/>
   <text width="80" valign="middle"
         text="$path{'price/text()'}">
      <method name="applyData">
         this.setText("$" + this.text);
      </method>
   </text>
   <stableborderlayout axis="x"/>
</class>
```

Resizes image to fit within 75 pixels

❶ Builds URL by adding image name

Displays title

❷ Postprocesses price to add $

When a matching XML element is encountered, all the objects are loaded with their matching data. The thumbnail class contains the product image, resized to a height of 75 pixels. This class uses an `applyData` method, ❶ and ❷, to perform postprocessing to attach a resource consisting of a global constant URL path prepended to the retrieved image name, and also to prepend a dollar sign to the price.

To resize an image to a thumbnail without distortion requires maintaining its aspect ratio.

10.4.2 Resizing images using aspect ratio

An image's *aspect ratio* is the ratio of width to height. If an image is stretched or compressed without maintaining its aspect ratio, the final image will distort. Although we initially allocated a space of 75 by 75 pixels, the natural shape of the resource included in listing 10.8 is rectangular, with width of 360 and height of 500, resulting in a width-to-height aspect ratio of 360 divided by 500 equaling 72 percent. Since the height is fixed at 75 pixels, the updated width value is the height multiplied by this calculated aspect ratio. The thumbnail class, shown in listing 10.9, handles image-resizing issues.

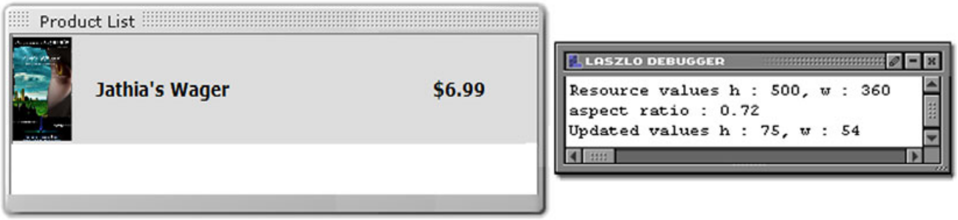
Listing 10.9 Using the aspect ratio to resize a dynamically loaded image

```
<class name="thumbnail" stretches="both" height="100">
   <handler name="onload">
      var iH = this.resourceheight;
      var iW = this.resourcewidth;             Determines
      var aspectratio = iW/iH;                 aspect ratio

      setHeight(this. height);                        Sets image to
      setWidth(this. height * aspectratio);           thumbnail size
      Debug.write("Resource values h : " + iH +
                  ", w : " + iW);
      Debug.write("aspect ratio : " + aspectratio);
      Debug.write("Updated values h : " + this.height +
                  ", w : " + this.width);
   </handler>
</class>
```

Loading an image file is a two-step process. An `ondata` event is generated when the name of the image file is read from XML data. This name is then used to access the image. An `onload` event is then sent after the image has been downloaded. At this point, the resource's `height` and `width` attributes are available. A resource also has a pair of `unstretchedheight` and `unstretchedwidth` attributes, which contain unalterable resource values. These values are useful for determining the state of the original dimensions and are used when a resource needs to be repeatedly resized.

Our example graphic, shown in figure 10.14, has a portrait layout with physical dimensions of 360/500, or 72 percent. Setting the thumbnail object's `height` attribute to 75 resizes its width to 54 pixels.

Despite accommodating the dimensions of external images, we still adhere pretty closely to the wireframe diagram. Now that we can display a single product item, we are ready to move on to the next phase, displaying multiple product rows.



```
Resource values h : 500, w : 360
aspect ratio : 0.72
Updated values h : 75, w : 54
```

Figure 10.14 The resized photo has the correct height with the width resized to accommodate the height.

10.5 Prototyping with grids

When products are added to the dsProducts dataset, more products are displayed than can fit in the Product List window. Because the stock window component doesn't provide scrolling, these additional products aren't displayed. Rather than supplementing the window with scrolling and other features, a better choice is to switch to the *grid* component. The grid is designed to quickly produce a prototype that contains a large range of built-in features for displaying tabular data, such as labeled headers, window scrolling, the ability to sort fields, alternating colored backgrounds, and selection highlights. As you'll see, using a grid can quickly produce a display with a polished appearance.

10.5.1 Using grids

A grid tag is intended to contain gridcolumn and gridtext tags. The latter are used only when necessary to allow editing of text data. Otherwise, the gridcolumn tag is used for displaying both images and text fields.

Listing 10.10 shows the definition of a productgrid class for our tabular product data. We want the data to be fixed within columns, so the resizable attribute is reset to false.

Listing 10.10 Using a grid in a class definition for the Product List window

```
<class name="productgrid" extends="grid" fontstyle="bold"
       fontsize="14" rowheight="79" bgcolor0="0xDDDDDD">

   <gridcolumn resizable="false" width="70" sortable="false">Image
      <thumbnail name="image" datapath="@image"
                 valign="middle" height="75">
         <attribute name="imageURL" value="$path{'@image'}"/>
         <method event="ondata">
            this.setResource(gController.IMAGESFOLDER + this.imageURL);
         </method>
      </thumbnail>
   </gridcolumn>

   <gridcolumn resizable="false" width="${parent.width-170}">Title
      <text valign="middle" datapath="@title"/>
   </gridcolumn>

   <gridcolumn resizable="false" width="100" sortable="false">Price
      <text valign="middle" datapath="@price">
         <method event="ontext">
            this.setText("$" + this.text);
```

❶ Sets alternating background color

Creates Image column ❷

Creates Title column

Creates Price column

```
            </method>
         </text>
      </gridcolumn>
   </class>
```

In the class tag, the bgcolor0 attribute ❶ provides a background color for even-numbered rows, and the rowheight attribute provides some padding between rows. We want fixed-size columns, so each column has a width with its resizable attribute set to false. Only the Title column ❷ is sortable; a sorted column is indicated by a downward arrow. Now we only need to invoke an instance of this productgrid class.

The Product List window is updated with a productgrid instance. Since the base grid component handles all layout-related issues for tabular data, the simplelayout tag can be deleted. The grid also needs to control the instantiation of the replication manager, so now the datapath's XPath expression matches the single XML expression dsProducts/products. The result of these changes is shown in listing 10.11.

Listing 10.11 Applying the productgrid class to the Product List window

```
<window title="Product List"
        x="${parent.browsesearch.width}"
        y="${parent.details.height}"
        width="55%" height="70%">
   <productgrid width="100%" height="100%"
                datapath="dsProduct:/products"/>
</window>
```

Sets width and height for scrolling

Sets data path to parent

Figure 10.15 shows the window display after the user has clicked the Title column to sort the items. The second item is currently selected, giving it a different background color.

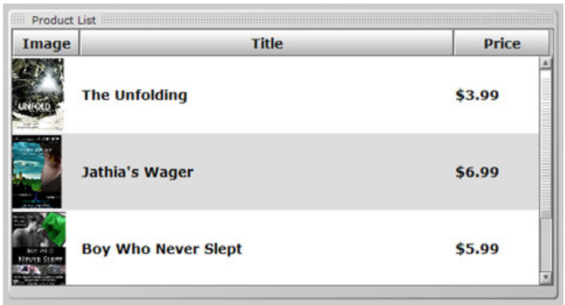


Image	Title	Price
	The Unfolding	$3.99
	Jathia's Wager	$6.99
	Boy Who Never Slept	$5.99

Figure 10.15 Grids are a convenient tool for prototyping, allowing the contents of a dataset to be displayed with a number of viewing features, including column headers, sortable columns, scrolling, alternating background colors, and selection.

Now that we're satisfied with the appearance of the Product List, let's explore the onselection event provided by the grid for processing a user's selection.

10.5.2 Processing a user selection

The Product List window is the starting point for interactions with the other windows. For example, a selected product is displayed in the Product Details window. It must be draggable to move it to the Shopping Cart or Media Player window. The grid component supports these interactions with the onselect event, in addition to the familiar onclick, onmousedown, and onmouseup events.

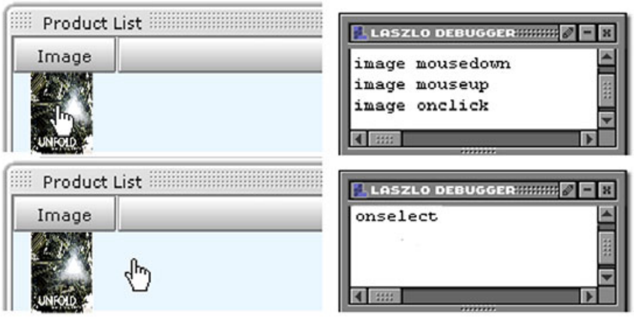
The onselect event is used to support the more complex multiple and range selection features found in most desktop applications. It follows the usual conventions; the Shift key is used to add a range of selections to the current selection, and the Ctrl key allows individual selections from the currently selected set to be toggled on or off.

The grid container handles selections, while its constituent fields handle mouse events. So the Title and Price columns generate onselect events, while the image in the Image column sends onclick and other mouse-related events. This is illustrated with the code in listing 10.12.

Listing 10.12 Event handling in a grid

```
<class name="productgrid" ...>
   ...
   <method event="onselect">
      Debug.write("onselect : " + this.getSelection()[0]);
   </method>
   <gridcolumn resizable="false" width="70" sortable="false">Image
      <thumbnail name="image" datapath="image" valign="middle"
                 height="75">
          ...
         <method event="onclick">
            Debug.write("image onclick");
         </method>
         <method event="onmousedown">
            Debug.write("image mousedown");
         </method>
         <method event="onmouseup">
            Debug.write("image mouseup");
         </method>
      </thumbnail>
   </gridcolumn>
</class>
```

Set to lowest-valued selection



```
image mousedown
image mouseup
image onclick
```

```
onselect
```

Figure 10.16
The top image illustrates how the mouse events are different, depending on where the mouse selection occurs. Any selections that occur in the image area generate mouse events instead of `onselect` events.

The `onselect` event handler allows us to capture and populate the Product Details window based on the user's selection. Figure 10.16 illustrates how the mouse events are different, depending on where the mouse selection occurs. Any selections occurring in the image area generate mouse events rather than `onselect` events.

Now that we can process a user selection from a list of choices, we'd like to display a more detailed summary of this selection in the Product Details window. But before we can go further, we need to learn more about navigating and manipulating the structure of the XML data tree. Although `datapath` expressions are fine for high-level operations, further development of the Laszlo Market requires lower-level operations, which are the subject of the next chapter.

10.6 Summary

The LZX model of interfacing with XML documents using declarative data paths is a powerful paradigm, resulting in a highly unified system. We can create an XML document that controls the appearance of the user interface. Just as a declarative system specifies only the relationships among entities, and relies on an operational system to enforce these relationships, a data path specifies a relationship between the data elements comprising an XML document and the visual objects representing them.

Once this relationship is established, there is no need to supplement it with procedural code. It doesn't matter if the attached dataset contains no matching data elements, a single matching data element, or multiple matching data elements. The Laszlo system carries out the correct behavior based on the data path definition; the attached visual object is not visible, it appears with a single object, or it appears with multiple objects created by a replication manager. This results in a highly flexible architecture; once the declarative statements are written, there is no need to modify them to address different data requirements.

The dataset is a powerful abstraction, allowing data to be directly represented in LZX. This almost completely eliminates any type of data access infrastructure, a common feature of most other programming languages. This lightweight access to data supports the top-down design-centric approach, and permits easy extensions to a prototype to work with mock datasets. We can add behavior to a prototype and nail down design and sizing requirements as early as possible.

Best of all, prototyping can be merged into the final application. There is very little throw-away code. Switching a data feed from a mock dataset to a networked dataset in an HTTP web server requires only the switching of a single attribute of the `dataset` object.

In this chapter, we supplied mock prototype data to the Laszlo Market application. Our major goal was to explore how datasets, data paths, and LZX objects work together to display data contained in an XML document. In the next chapter, we'll expand on the various uses of this data as we continue to add features to our prototype—features that allow us to manipulate data in a dataset.

11 Using dynamic dataset bindings

This chapter covers

- Modifying XML data elements
- Navigating through the XML data structure
- Using the master-detail design pattern

> *A mathematician is a device for turning coffee into theorems.*
>
> —Paul Erdós
> (perhaps from Alfréd Rényi), mathematician

The previous chapter dealt with how a data binding establishes a relationship between data and its bound visual object. This static relationship is limited, remaining fixed at a single location in a dataset. Additionally, although data bindings support bidirectional communications, communication from objects to datasets is limited to the `updateData` method. This chapter discusses how to move this binding to traverse a dataset and manipulate its contents and structure using a complementary set of class groups derived from the data pointer and data node base classes.

11.1 Linking data nodes and data pointers

The relationships among the data pointer and data node classes and a dataset are shown in figure 11.1. A data pointer establishes the context of the current location, while the data node classes map to individual elements of a dataset.

The data pointer classes, shown on the left side of the diagram, are derived from the `LzNode` class and are known as data pointer-derived classes. Since `LzNode` is their superclass, they are available as declarative tags. They provide direct access to modify an XPath expression or even to update the dataset. When a data pointer context is updated, it allows an XML data structure to be navigated and generates `ondata` events to initiate view-level actions.

The data node set of classes, derived from `LzDataNode` as shown on the right side of figure 11.1, are based on the W3C DOM XML specification to provide a direct representation of XML elements. The `LzDataNode` class serves as an abstract superclass for the `LzDataElement` and `LzDataText` classes, representing the data and text XML elements. We can use these classes to modify the values of data elements, delete data elements, or create new elements.

The data node and data pointer classes perform complementary roles, allowing XML structure manipulations to be performed both online and offline. Later, we'll see how these classes are best used in combination.

Finally, these two sets of classes wouldn't be complete without ways to convert from a data node to a data pointer and back again. These conversion features allow us to use data pointers and data nodes collaboratively. Since the `LzDataNode`-derived classes provide the building blocks pointed to by the `LzDataPointer`-derived classes, we'll start there.

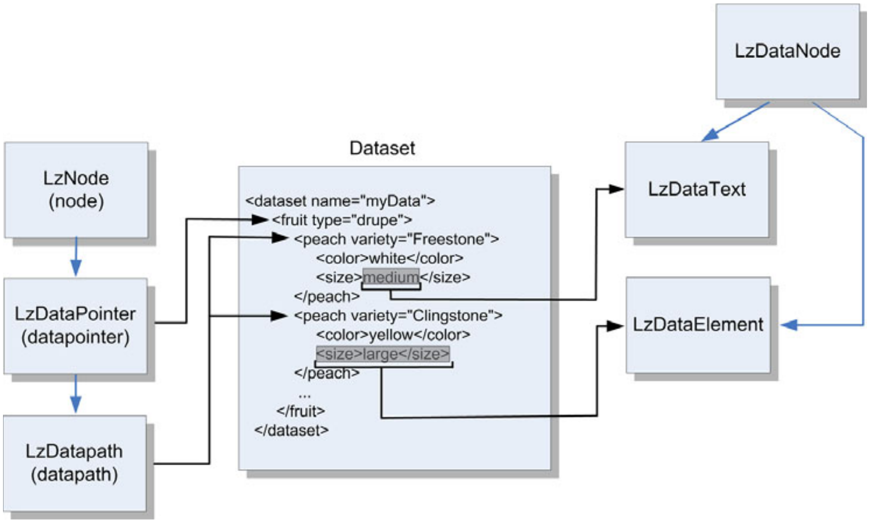


Figure 11.1 Laszlo provides two main approaches for accessing XML data, the data-pointer-oriented approach and the data-node-oriented approach, embodied in two corresponding sets of classes. The `LzDataPointer` class contains attributes and methods for translating from one approach to the other.

11.2 The LzDataNode classes

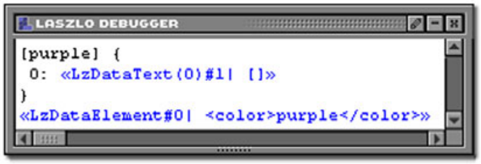
`LzDataNode` and `LzNode` are Laszlo's two base classes. While `LzNode` is the superclass for all declarative tags, `LzDataNode` is the base class for the data node classes representing the nodes of an XML dataset. Its lineage is more limited, since it's an abstract class for only the `LzDataElement` and `LzDataText` classes. These classes aren't derived from `LzNode`, so they can't be used as declarative statements and can't send events.

The `LzDataElement` and `LzDataText` classes map to the XML data and text elements stored in a dataset. We can use them to examine the contents of a returned `data` object:

```
<canvas debug="true">
   <include href="myData.lzx"/>
   <text name="main" datapath="myData:/fruit/plum/color">
      <handler name="ondata" args="data">
         Debug.write(data);
         Debug.inspect(data.childNodes);
      </handler>
   </text>
</canvas>
```

NOTE *Obtaining the myData dataset*—All the examples in this chapter use the `myData` dataset defined in listing 10.1.

Figure 11.2 shows the matching data returned by the XPath expression, consisting of an `LzDataElement` object containing the `<color>purple</color>` data element. This data element is a parent node that contains a child text element, an `LzDataText` object. The `data` attribute for this text element contains the text string `purple`.



```
[purple] {
 0: «LzDataText(0)#1| []»
}
«LzDataElement#0| <color>purple</color>»
```

Figure 11.2
The `datapath`'s data field returns a value of purple. This value comprises an LzDataElement object enclosing an LzDataText object, whose data field contains the value purple.

Now that we've broken down the composition of a dataset into its base classes, let's look at each of these classes in greater detail, starting with their abstract `LzDataNode` superclass.

11.2.1 The abstract LzDataNode superclass

The `LzDataNode` class defines the structure of an XML node. All its attributes are read-only, as shown in table 11.1, because they reflect the properties of a tangible data node. The `nodeType` attribute identifies this data node as being either a data or a text element. The `ownerDocument` attribute identifies its affiliated dataset. The `parentNode` attribute contains a reference to its parent's `LzDataNode` object. Let's now see how these base properties are extended by the `LzDataElement` and `LzDataText` classes that are derived from `LzDataNode`.

Table 11.1 `LzDataNode` attributes

Name	Data Type	Attribute	Description
`nodeType`	`number`	Read-only	The type of this node `ELEMENT_NODE` or `TEXT_NODE`
`ownerDocument`	`LzDataNode`	Read-only	The XML document or dataset associated with this node
`parentNode`	`LzDataNode`	Read-only	The XML parent node

11.2.2 Building datasets with LzDataElements

An LzDataElement object represents a data element in an XML dataset. This element can be either an end node or a parent node with children. A new LzDataElement object is instantiated with its own constructor. Its attributes are shown in table 11.2 and are passed as arguments to this constructor:

```
LzDataElement(nodeName, [attributes, childNodes]);
```

where

- nodeName is the name of the new node.
- attributes is a list of attributes for this node (optional).
- childNodes is an array for children of this node (optional).

Omitting the childNodes argument creates a single data node, while specifying an array childNodes produces a parent node with child nodes.

Table 11.2 LzDataElement attributes

Name	Data Type	Attribute	Description
attributes	object	Setter	List of attributes for this node
childNodes	LzDataNode[]	Setter	An array of children for this node
nodeName	string	Setter	The name of this node

LzDataElement isn't limited to creating single data elements, but can also be used to create complex data structures. In the next example, we'll add a top-level vegetable parent node to the myData dataset. We'll start with a root element named vegetable:

```
var ele = new LzDataElement ("vegetable", … )
```

then add an associative array of names and values to create its attributes:

```
{type:"Tuber", condition:"fresh"}
```

and finally add some child nodes, contained in a regular array populated with individual LzDataElement instances:

```
[new LzDataElement("carrot"), new LzDataElement("turnip")]
```

All these steps are shown in listing 11.1.

Listing 11.1 Inserting XML data into a dataset

```
<canvas width="100%" debug="true">
   <include href="myData.lzx"/>
   <method name="init">
      var ele = new LzDataElement ("vegetable",
                {type:"Tuber", condition:"fresh" },
                [new LzDataElement("carrot", {variety:"Danvers"}),
                new LzDataElement("turnip", {variety:"Brassica"})]);
      ele.setOwnerDocument(myData);
      myData.appendChild(ele);
      Debug.inspect(ele);
      Debug.inspect(myData.childNodes);
   </method>
</canvas>
```

❶ Creates new element

❷ Appends element to dataset

❸ Selects dataset owner

Laszlo can't interact with an XML data structure unless it's attached to a dataset. So after creating our new XML data structure ❶, there is a two-step procedure to attach it to a dataset. First, its ownerDocument attribute is set ❷ to the myData dataset. Next, the data element is inserted into the dataset ❸ with the appendChild method.

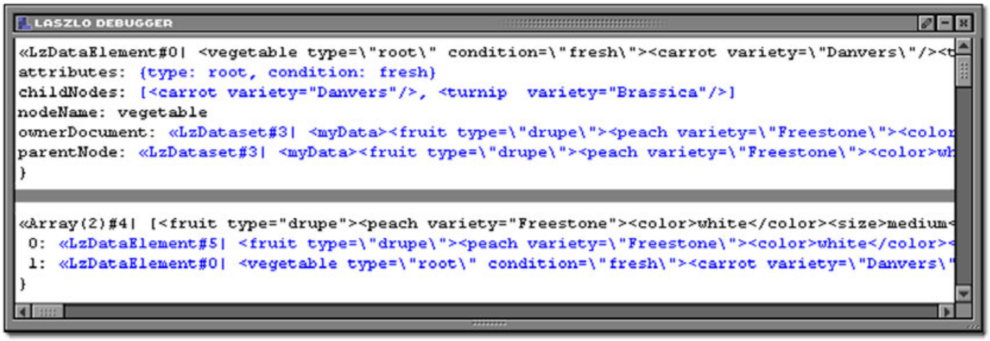
When this code is executed, the XML document contained in the myData dataset is updated to contain a vegetable sibling node alongside the fruit node, as you can see here:

```
<myData>
   <fruit type="Drupe"/>
   ...
   </fruit>
   <vegetable type="Tuber" condition="fresh">
      <carrot variety="Danvers"/>
      <turnip variety="Brassica"/>
   </vegetable>
</myData>
```

Since it's attached to the myData dataset, its root node is <myData>…</myData>.

We can verify our results by checking the dataset's contents in the debugger (see figure 11.3). We've added a cosmetic line to differentiate the two sections. The top part displays the attribute composition of the vegetable parent node, while the bottom part displays the top-level sibling nodes of the dataset.

Examining our new data element, we see that it is named vegetable; has two attributes, type and condition; and is the parent to two child nodes, carrot and turnip, where each is identified with a unique variety attribute. Its ownerDocument and parentNode attributes are of type LzDataset, to indicate its inclusion in



```
«LzDataElement#0| <vegetable type=\"root\" condition=\"fresh\"><carrot variety=\"Danvers\"/><t
attributes: {type: root, condition: fresh}
childNodes: [<carrot variety="Danvers"/>, <turnip  variety="Brassica"/>]
nodeName: vegetable
ownerDocument: «LzDataset#3| <myData><fruit type=\"drupe\"><peach variety=\"Freestone\"><color
parentNode: «LzDataset#3| <myData><fruit type=\"drupe\"><peach variety=\"Freestone\"><color>wh
}

«Array(2)#4| [<fruit type="drupe"><peach variety="Freestone"><color>white</color><size>medium<
 0: «LzDataElement#5| <fruit type=\"drupe\"><peach variety=\"Freestone\"><color>white</color><
 1: «LzDataElement#0| <vegetable type=\"root\" condition=\"fresh\"><carrot variety=\"Danvers\"
}
```

Figure 11.3 Checking the structure of the `LzDataElement` displays its attributes describing its composition. This verifies that the `vegetable` data element was correctly built and attached to the `myData` dataset.

the `myData` dataset. Examining the updated composition of this dataset, we see that it consists of an array with two nodes, `fruit` and `vegetable`.

11.2.3 Core methods of LzDataElement

Many of the `LzDataNode` methods overlap `LzDatapointer` methods to provide offline versus online versions. Table 11.3 lists the common methods. When discussing a topic involving common methods, we'll default to covering the navigational methods of the `LzDatapointer` classes and the node-creation methods of the `LzDataNode` classes.

The remaining LzDataElement methods are shown in table 11.4.

Table 11.3 Methods common to the `LzDatapointer`- and `LzDataNode`-based classes

`Datapointer` Classes	`DataNode` Classes	Description
`addNode`	`appendChild`	Adds a new child node below the current context
`deleteNode`	`removeChild`	Removes the referenced node
`deleteNodeAttribute`	`removeAttr`	Removes the name attribute from the current node
`getChild`	`getFirstChild` and `getLastChild`	Gets a specified child node
`getNodeAttribute`	`getAttr`	Returns the value of the current node's name attribute

Table 11.3 Methods common to the `LzDatapointer`- and `LzDataNode`-based classes *(continued)*

`Datapointer` Classes	`DataNode` Classes	Description
`getNodeName`	`NodeName (LzDataNode attribute)`	Gets the name of the node that the data pointer is pointing to
`getNodeText`	`data (LzDataText attribute)`	Returns a string that is a concatenation of the text nodes beneath the current element
`selectNext`	`getNextSibling`	Selects the next sibling node in the dataset
`selectPrev`	`getPreviousSibling`	Selects the previous sibling node in the dataset
`selectParent`	`parentNode (LzDataNode attribute)`	Moves up the data hierarchy to the next parent node in the dataset
`serialize`	`serialize`	Serializes the current element and its children to an XML string
`setNodeAttribute`	`setAttr`	Sets the name attribute of the current node to the `val` argument
`setNodeName`	`setNodeName`	Sets the name of the current element to the `name` argument
`setNodeText`	`setData`	Sets the current node's text to the `text` argument

Table 11.4 The remaining `LzDataElement` methods

Name	Description
`appendChild(newChild)`	Adds a child to this node's list of `childNodes`
`childOf(node1, node2)`	States if `node1` is an ancestor node of `node2`
`cloneNode(deep)`	Returns either a shallow or deep copy of this node
`LzDataElement(name, attributes, children)`	Represents a hierarchical data node
`getAttr(name)`	Returns the value for the given attribute
`getElementsByTagName()`	Returns a list of `childNodes` for this node that have a given name
`insertBefore(newChild, refChild)`	Inserts the given `LzDataNode` before another node in this node's `childNodes`

Table 11.4 The remaining `LzDataElement` methods *(continued)*

Name	Description
`setAttr(name, value)`	Sets the given attribute to the given value
`setAttrs(attrs)`	Sets the attributes of this node to the given object
`setChildNodes(children)`	Sets the children of this node to the given array
`setNodeName(name)`	Sets the name of this node

We'll now re-create the `vegetable` parent node, consisting of `carrot` and `turnip` child nodes, in listing 11.2, to demonstrate a cross section of `LzDataElement`'s methods.

Listing 11.2 Populating a dataset using `LzDataElement` methods

```
<canvas width="100%" debug="true">
   <include href="myData.lzx"/>

   <method name="init">
      var ele = new LzDataElement ("vegetable",
                {type: "Tuber", condition: "fresh" },
                [new LzDataElement("onion", {variety: "Valencia"}),
                new LzDataElement("potato", {variety: "Idaho"})]);
      nele = ele.cloneNode(false);
      Debug.write("cloneNode (shallow):" + nele);
      nele = ele.cloneNode(true);
      Debug.write("cloneNode (deep):" + nele);
      Debug.write("insert carrot before onion");
      var carrot = new LzDataElement("carrot");
      carrot.setAttr("variety", "Brassica");
      var onion = nele.getElementsByTagName("onion");
      nele.insertBefore(carrot, onion[0]);
      Debug.write("Change potato to turnip");
      var last = nele.getLastChild();
      last.setNodeName("turnip");
      last.setAttr("variety", "Danvers");
      Debug.write("Remove onion");
      var mid = nele.getFirstChild();
      mid = mid.getNextSibling();
      nele.removeChild(mid);
      Debug.write(nele.serialize());
   </method>
</canvas>
```

1 Performs shallow copy

2 Performs deep copy

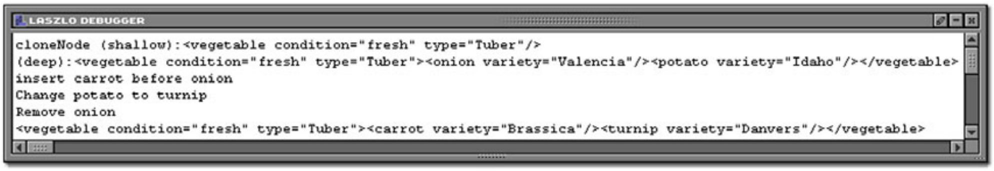
3 Inserts new carrot before onion

4 Updates potato to turnip

5 Removes onion

In listing 11.2, ❶ performs a *shallow clone* operation that only clones the current node, while ❷ performs a *deep clone* operation that copies all its descendents. The purpose of this step is to create a practice data structure with the appearance of the `vegetable` branch.

To convert this data structure ❸, a new `carrot` `LzDataElement` is instantiated and inserted before the `onion` element. The `getElementsByTagName` method returns an array of matching `onion` elements. We insert the first `onion` before the `carrot`. The next step ❹ is to update the existing `potato` element to be a `turnip`. Finally, ❺ the extra `onion` element is eliminated. To access this `onion` element, it's necessary to descend stepwise down to the child nodes and then to move laterally to the correct sibling. Each of the key steps are displayed in figure 11.4 to demonstrate the result of each operation.



```
cloneNode (shallow):<vegetable condition="fresh" type="Tuber"/>
(deep):<vegetable condition="fresh" type="Tuber"><onion variety="Valencia"/><potato variety="Idaho"/></vegetable>
insert carrot before onion
Change potato to turnip
Remove onion
<vegetable condition="fresh" type="Tuber"><carrot variety="Brassica"/><turnip variety="Danvers"/></vegetable>
```

Figure 11.4 Using various methods, we update the `myData` dataset to have a `vegetable` branch.

Although we have taken an unnecessarily long, roundabout way to add a `vegetable` branch, the purpose of this exercise was to demonstrate these data element methods.

11.2.4 Working with LzDataText text nodes

The `LzDataText` class represents a text element within an XML dataset, with these characteristics: it contains only text data, must be childless, and be a child of an `LzDataElement` object. As shown in table 11.5, it has only a single attribute, `data`, of type, `string`. This definition results in all XML data, including all numeric values, being interpreted as character strings.

Table 11.6 shows the `LzDataText` constructor and data setter methods.

Table 11.5 `LzDataText` attributes

Name	Data Type	Attribute	Description
`data`	`string`	Setter	The data held by this node

Table 11.6 `LzDataText` **methods**

Name	Description
`LzDataText(text)`	Constructs a text node in a set of data
`setData()`	Sets the string that this node holds

`LzDataText` objects are created through this constructor, which takes a text string as its argument:

```
var txt = new LzDataText("text string");
```

Listing 11.3 illustrates various ways to instantiate and initialize a text node. In this example, we'll create a `plum` data element that contains `color` and `size` data elements that each contains a text element.

Listing 11.3 Various ways to instantiate and initialize a text node

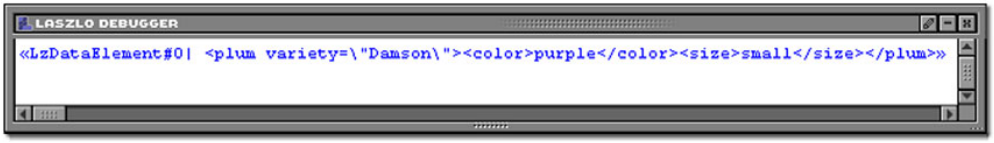
```
<canvas debug="true">
   <method name="init">
      var ele1 = new LzDataElement("color");
      ele1.setChildNodes([new LzDataText("purple")]);
      var ele2 = new LzDataElement("size");
      dt = new LzDataText();
      dt.setData("small");
      ele2.appendChild(dt);
      var ele = new LzDataElement("plum", {variety: "Damson"},
                                    [ele1, ele2]);
      Debug.write(ele);
   </method>
</canvas>
```

❶ **Instantiates color data element**

❷ **Instantiates size data element**

❸ **Instantiates parent node**

First ❶, a minimal `LzDataElement` object is instantiated and an `LzDataText` object is added as a child node. Next ❷, a minimal `LzDataText` object is instantiated and its text string is set, which is then appended to its parent `LzDataElement`. Finally ❸, the parent node, `plum`, is instantiated with an array containing the previous two data elements as its children. This produces one of the `fruit` data elements shown in figure 11.5.



```
«LzDataElement#0| <plum variety=\"Damson\"><color>purple</color><size>small</size></plum>»
```

Figure 11.5 This output shows the plum `LzDataElement` consisting of two data nodes, each containing a single text node.

These features provide the critical functionality for the LzDataNode classes. Their remaining methods concern navigation, which are also handled by the LzDatapointer and LzDatapath classes and are covered in section 11.3.

11.2.5 Building XML structures with power tools

The LzDataElement and LzDataText objects are intended for working on a single node. Constructing large XML structures requires the use of more powerful tools. The LzDataElement class contains three static methods, makeNodeList, valueToElement, and stringToLzData, designed for handling these large-scale tasks. We'll look at each of these tools.

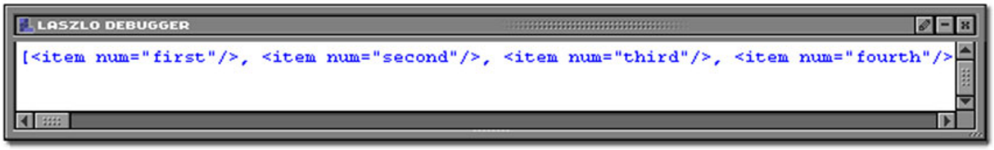
Creating sibling nodes with makeNodeList

The makeNodeList method provides a way to create a large number of identical data nodes. It's used most frequently to provide child nodes for a parent. Since these elements are stored in an array, we can iterate through them to make custom modifications with a parallel array containing unique values:

```
<canvas debug="true">
   <method name="init">
      <![CDATA[
      var ele = LzDataElement.makeNodeList(4, "item");
      var attr = ["first", "second", "third", "fourth"];
      for (i = 0; i < ele.length; i++) {
         ele[i].setAttr("num", attr[i]); }
      Debug.write(ele);
      ]]>
   </method>
</canvas>
```

Figure 11.6 shows that each of the newly created nodes has a distinct num attribute value.

Combining makeNodeList with a nested iterator is a useful technique for creating multilayered data structures.



```
[<item num="first"/>, <item num="second"/>, <item num="third"/>, <item num="fourth"/>
```

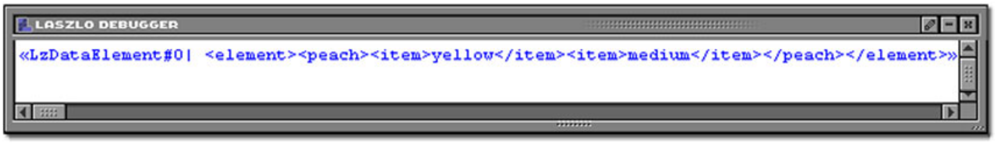
Figure 11.6 An arbitrary number of identical data nodes is easily created with makeNodeList.

Converting objects to XML with valueToElement

The `valueToElement` method converts JavaScript objects into XML data. These JavaScript objects can be composed of nested associative and regular arrays to create complex data structures. In the following example, an associative array is created consisting of a name and a value, where the value is a regular array consisting of strings. This associative array is then passed as an argument to the `valueToElement` method.

```
<canvas debug="true">
   <method name="init">
      var obj = {peach: ["yellow", "medium", ".60"]};
      var de = LzDataElement.valueToElement(obj);
      Debug.write(de);
   </method>
</canvas>
```

This code creates a `peach` parent node containing a series of `item` child nodes. However, since the root node name defaults to `element` and each data node name defaults to `item`, this data structure still requires extensive manual work to clean it up. Figure 11.7 shows the debug output.



```
«LzDataElement#0| <element><peach><item>yellow</item><item>medium</item></peach></element>»
```

Figure 11.7 The `valueToElement` method converts JavaScript objects into equivalent XML data structures. It suffers from the limitation of using a default node name (item).

Most situations require unique names for nodes, so this method is of limited use. But in special situations, it can still be quite useful. We have saved the most powerful and useful of the power methods for last.

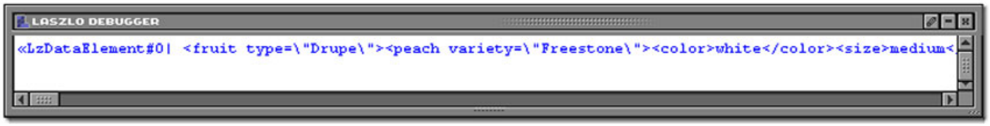
Building complete structures with stringToLzData

The simplest and fastest way to build an XML data structure is to use the `stringToLzData` method, which uses a string to build the data structure. Because a string can easily be updated with dynamic runtime information, it's easily modified to contain updated values. Whenever a large XML data structure must be dynamically built, `stringToLzData` is your tool of choice:

```
<canvas debug="true">
   <method name="init">
      <![CDATA[
      var str = "<fruit type=\"Drupe\">"
```

```
        str = str.concat("<peach variety=\"Freestone\">");
        str = str.concat("<color>white</color>");
        str = str.concat("<size>medium</size>");
        str = str.concat("</peach></fruit>");
        var node = LzDataNode.stringToLzData(str);
        Debug.write(node);
        ]]>
     </method>
  </canvas>
```

As shown in figure 11.8, this dynamically created XML data structure is complete, requiring no manual modifications.



```
«LzDataElement#0| <fruit type=\"Drupe\"><peach variety=\"Freestone\"><color>white</color><size>medium<
```

Figure 11.8 The `stringToLzData` method converts a string into an XML data structure. There is no limit to the length of the string, so an XML structure of arbitrary complexity can be produced.

This collection of power tools provides a comprehensive set of solutions that addresses many different situations requiring the runtime creation of an XML structure.

Let's now look at how to navigate through an XML data structure using `LzDatapointer` and `LzDatapath`.

11.3 Navigating with LzDatapointer and LzDatapath

In chapter 10, we saw how a data path establishes a link, specified by its XPath expression, between an LZX object and a data element in a dataset. There, the data path is specified as an attribute in a declarative tag. A limitation of this mechanism is the static XPath expression; its location is fixed to point only at a specific node. In this section, we'll see how the position can be dynamically updated at runtime.

Since they are descendants of `LzNode`, these classes can be expressed as `datapointer` and `datapath` declarative tags. This means that a data path isn't confined to be an attribute in a declarative tag, but can also be represented as its own declarative tag.

We'll start with an introduction to the `LzDatapointer` class, since it's the superclass for the data path.

11.3.1 Navigating with data pointers

To be useful, a data pointer must point at a data node through its `xpath` attribute:

```
<datapointer xpath="myData:/fruit"/>
```

While declarative tags contain a `datapath` attribute to support a data binding, a data pointer contains an `xpath` attribute to specify its XPath expression. An `xpath` attribute differs from a data path by only checking for a matching data element and then repositioning its data pointer or data path to that location; no data-binding operations or replication managers are instantiated.

A data pointer can also be dynamically created and positioned in JavaScript, like this:

```
var dp = new LzDatapointer();
dp.setXPath("myData:/fruit");
```

Whether invoked through a declarative tag or JavaScript, a data pointer is created that points to a single data element in a dataset. Data pointers have the limitation that their XPath expression can only return a single matching node. When its XPath expression returns multiple matching nodes, it results in a runtime error.

A data pointer can specify the set of attributes shown in table 11.7.

Table 11.7 Data pointer attributes

Name	Data Type	Attribute	Description
`context`	`XML data`	Read-only	The dataset associated with this XML data
`p`	`expression`	Read-only	The `LzDataNode` that the data pointer is pointing to
`rerunxpath`	`boolean`	Setter	If true, reevaluates the XPath expression when the dataset is edited; defaults to false
`xpath`	`string`	Setter	Contains the XPath expression for matching a data node

In the remainder of this section, we'll access and display an individual data node, explore navigation, and modify a dataset to demonstrate the various `LzDatapointer` methods.

11.3.2 Accessing data and text nodes

We'll start by demonstrating how to access and display attributes, node names, and node text for a single data node. Rather than list all data pointer methods in a single table, they will be grouped by a common theme. Each of these tables contains an `ondata` event field to indicate whether or not a method sends an `ondata` event to bound objects. It's important to be aware of which methods generate an `ondata` event, since a bound declarative object receives the event. Table 11.8 lists the methods for accessing node information.

Table 11.8 Data pointer methods for accessing node information

Name	`ondata` Event	Description
`getDataset()`	No	Returns a reference to the data pointer's dataset
`getNodeAttribute(name)`	No	Returns the value of the current node's `name` attribute
`getNodeAttributes()`	No	Returns the attributes of the node pointed to by the data pointer in an object whose keys are the attribute names and whose values are the attribute values
`getNodeCount()`	No	Counts the number of element nodes that are children of the node that the data pointer is pointing to
`getNodeName()`	No	Gets the name of the node that the data pointer is pointing to
`getNodeText()`	No	Returns a string that is a concatenation of the text nodes beneath the current element
`isValid()`	No	Tests whether or not this data pointer is pointing to a valid node

For the examples in this section, the annotation numbers in the code match the numbers displayed in their accompanying figure. This allows you to correlate output results with the code that produced it. Listing 11.4 demonstrates how to display all the characteristics for a data element. It consists of two data pointers set to different locations in the `myData` dataset. The `dp1` data pointer points to a second `peach` data element, while the `dp2` pointer points to both of the `peach` data elements. The methods listed in table 11.8 are used to display the characteristics of these `peach` instances.

Listing 11.4 Accessing and displaying XML attributes, node names, and node text

```
<canvas debug="true">
   <include href="myData.lzx"/>

   <datapointer id="dp1" xpath="myData:/fruit/peach[2]">
       <handler name="ondata" args="d">
          Debug.write("ondata :" + d);
       </handler>
   </datapointer>
   <datapointer id="dp2" xpath="myData:/fruit/peach">
      <handler name="ondata" args="d">
         Debug.write("ondata :" + d);
      </handler>
      <method name="init">
         Debug.write("dp1 is valid = " + dp1.isValid() +
                     "\ndp2 is valid = " + dp2.isValid());
         Debug.write("getDataset=", dp1.getDataset());
         Debug.write("getNodeName=" + dp1.getNodeName());
         Debug.write("getNodeAttribute=" +
                dp1.getNodeAttribute("variety"));
         Debug.write("getNodeAttributes=",
                dp1.getNodeAttributes());
         Debug.write("getNodeCount=" +
                dp1.getNodeCount());
         Debug.write("getNodeName=" + dp1.getNodeName());
         dp1.setXPath("myData:/fruit/peach[1]/color");
         Debug.write("getNodeText=" + dp1.getNodeText());
      </method>
   </datapointer>
</canvas>
```

❶ Generates runtime error

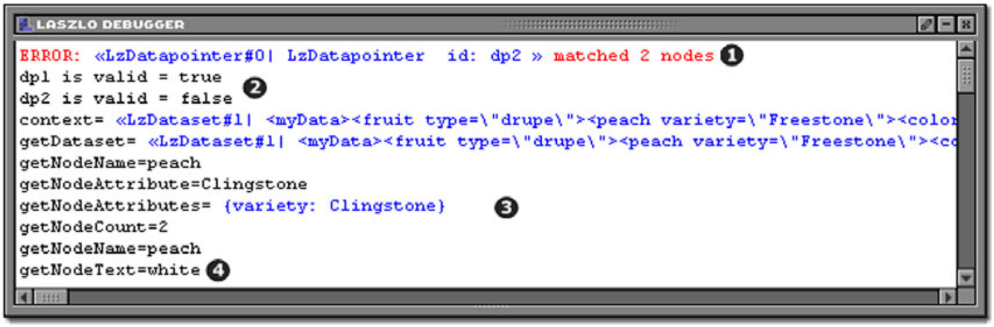
❷ Tests whether data pointer is valid

❸ Displays data node characteristics

❹ Displays text node

Because the XPath expression for the dp2 data pointer ❶ returns multiple nodes, it produces an error and fails a validity ❷ test. The dp1 data pointer ❸ is used to demonstrate how to access the various properties ❹ of a data element. Figure 11.9 shows the generated output.

Because all the methods listed in table 11.8 are used to read a data element's characteristics, they contain a no value in their ondata event column. As a result, no ondata debug messages are displayed in figure 11.9. Now that we can list the



```
ERROR: «LzDatapointer#0| LzDatapointer  id: dp2 » matched 2 nodes ❶
dp1 is valid = true
dp2 is valid = false ❷
context= «LzDataset#1| <myData><fruit type=\"drupe\"><peach variety=\"Freestone\"><colo
getDataset= «LzDataset#1| <myData><fruit type=\"drupe\"><peach variety=\"Freestone\"><c
getNodeName=peach
getNodeAttribute=Clingstone
getNodeAttributes= {variety: Clingstone}     ❸
getNodeCount=2
getNodeName=peach
getNodeText=white ❹
```

Figure 11.9 The top line in the debugger indicates that the dp2 data pointer is invalid, because it matches multiple XML data nodes. The other data pointer lists all the characteristics for its data element.

characteristics for a fixed location, let's see how to leverage these capabilities by traversing the nodes of a dataset.

11.3.3 Navigating a dataset

While both data node- and data pointer-derived classes support *cursor methods* for stepping through data nodes, only the data pointer classes send an ondata event when the location changes. The move doesn't even need to be to another data node, since moving to an invalid location generates an ondata event with a null data argument.

A data pointer supports lateral movement among sibling nodes with the selectNext and selectPrev methods, and vertical movement with the selectParent and selectChild methods. Each method takes an argument that specifies the number of nodes or levels to traverse.

Table 11.9 lists these navigation methods along with auxiliary "create duplicate pointer" and "compare pointer" methods, dupePointer and comparePointer, to round out the example.

In listing 11.5, two data pointers are created to traverse the sibling nodes in the fruit root node. An ondata event handler displays a debug message for each method generating an ondata event.

Table 11.9 Data pointer methods for navigating an XML structure

Name	ondata Event	Description
comparePointer(ptr)	No	Determines whether or not this pointer points to the same node as ptr
dupePointer()	No	Returns a new data pointer pointing to the same node, with a null XPath and a false rerunxpath attribute
selectChild(amnt)	Yes	Moves down the data hierarchy to the next child node in the dataset, if possible
selectNext(amnt)	Yes	Selects the next sibling node in the dataset, if possible
selectParent(amnt)	Yes	Moves up the data hierarchy to the next parent node in the dataset, if possible
selectPrev(amnt)	Yes	Selects the previous sibling node in the dataset, if possible
setXPath	Yes	Sets the pointer to the node returned by the XPath expression
xpathQuery	No	Returns the result of an XPath query without updating the pointer

Listing 11.5 Traversing sibling nodes

```
<canvas debug="true">
   <include href="myData.lzx"/>

   <datapointer name="dp1">
       <handler name="ondata" args="d">
          Debug.write("ondata :" + d);
       </handler>
   </datapointer>
   <datapointer name="dp2"/>

   <method name="init">
      Debug.write("setXPath to myData:/fruit/peach[1]");
      dp1.setXPath("myData:/fruit/peach[1]");
      Debug.write("xpathQuery " +
           dp1.xpathQuery
                ('myData:/fruit/cherry'));
      Debug.write("Create dupe ptr dp2 from dp1 : " +
            dp1.getNodeName());
      dp2 = dp1.dupePointer();
      Debug.write("Select sibling node from " +
            dp1.getNodeName() +
                  ", attribute=" + dp1.getNodeAttribute("variety"));
      dp1.selectNext(1);

      Debug.write("Select last sibling node from " +
            dp1.getNodeName());
      dp1.selectNext(2);
      Debug.write("Select previous sibling node from " +
                                                 dp1.getNodeName());
      dp1.selectPrev(1);
      Debug.write("Check if dp1 and dp2 pointers match : " +
                                            dp1.comparePointer(dp2));
      Debug.write("dp1=" + dp1.serialize());
      Debug.write("dp2=" + dp2.serialize());
      Debug.write("select parent of dp1");
      dp1.selectParent();
      Debug.write("Select child node from " +
           dp1.getNodeName());
      dp1.selectChild(1);
      Debug.write("dp1 and dp2 pointers now match : "
                                        + dp1.comparePointer(dp2));
   </method>
</canvas>
```

❶ Points at first peach data node

❷ Displays cherry node without updating

❸ Creates duplicate datapointer

❹ Selects next data node

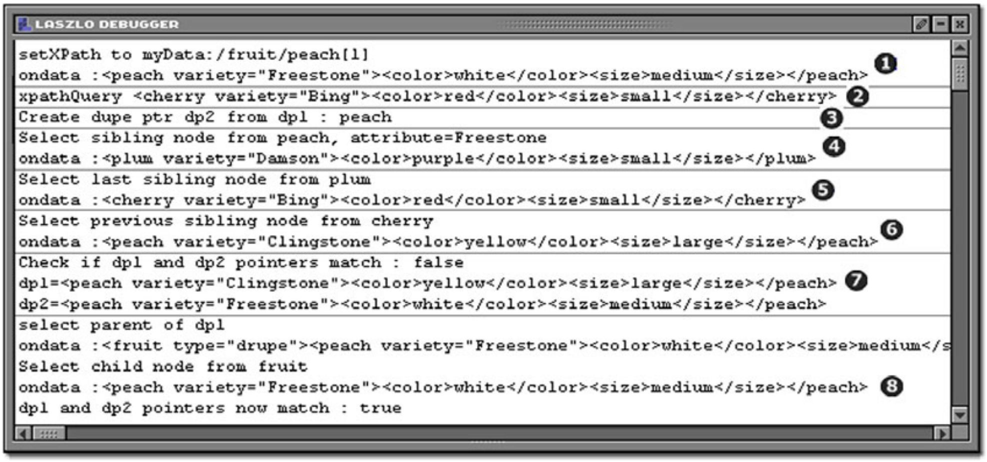
❺ Selects last data node

❻ Selects previous data node

❼ Compares two pointer locations

❽ Goes back to first peach data node

In listing 11.5, the dp1 data pointer is set ❶ to point at the first peach data node. The xpathQuery method is used ❷ to display the contents of the cherry data



```
setXPath to myData:/fruit/peach[1]
ondata :<peach variety="Freestone"><color>white</color><size>medium</size></peach> ❶
xpathQuery <cherry variety="Bing"><color>red</color><size>small</size></cherry> ❷
Create dupe ptr dp2 from dp1 : peach ❸
Select sibling node from peach, attribute=Freestone
ondata :<plum variety="Damson"><color>purple</color><size>small</size></plum> ❹
Select last sibling node from plum
ondata :<cherry variety="Bing"><color>red</color><size>small</size></cherry> ❺
Select previous sibling node from cherry
ondata :<peach variety="Clingstone"><color>yellow</color><size>large</size></peach> ❻
Check if dp1 and dp2 pointers match : false
dp1=<peach variety="Clingstone"><color>yellow</color><size>large</size></peach> ❼
dp2=<peach variety="Freestone"><color>white</color><size>medium</size></peach>
select parent of dp1
ondata :<fruit type="drupe"><peach variety="Freestone"><color>white</color><size>medium</s
Select child node from fruit
ondata :<peach variety="Freestone"><color>white</color><size>medium</size></peach> ❽
dp1 and dp2 pointers now match : true
```

Figure 11.10 Methods that generate an `ondata` event are indicated by a following `ondata` output line. The `data` value in the `ondata` output shows the current location in the dataset.

node, without updating the data pointer's location. As a result, no `ondata` event is generated. The `dupePointer` method is used ❸ to create a duplicate pointer to the `peach` location. In steps ❹, ❺, and ❻, the various cursor methods are demonstrated. Next, the data pointers are compared ❼ to determine if they point at the same location. Since they don't, `dp1` is moved up to its parent node and then down to its first child node. The two data pointers match, since `dp2` now points back to the first `peach` data node ❽. Figure 11.10 indicates those methods generating an `ondata` event by displaying an `ondata` message with the debug output.

We now know how to traverse and examine the contents of datasets using data pointers. The next step is to use these data pointers to manipulate a dataset's contents.

11.3.4 Creating and modifying datasets

We'll now modify our data structure by adding and deleting attributes, data nodes, and text nodes. Table 11.10 contains methods replicating the functionality available through the `LzDataNode`-derived classes, but since data pointer methods are used, `ondata` events are generated.

Table 11.10 Data pointer methods for manipulating an XML data tree

Name	ondata Event	Description
addNode(name, text, attrs)	Yes	Adds a new child node below the current context
addNodeFromPointer(dp)	No	Duplicates the node that dp points to, and adds it to the node pointed to by the data pointer
deleteNode()	Yes	Removes the node pointed to by the data pointer
deleteNodeAttribute(name)	No	Removes the name attribute from the current node
setFromPointer(dp)	Yes	Sets this data pointer to the location of the given data pointer
setNodeAttribute(name, val)	No	Sets the name attribute of the current node to val
setNodeName(name)	No	Sets the name of the current node to name
setNodeText(val)	No	Sets the current node's text to val; if the node already has one or more text children, the value of the first text node is set to val

In listing 11.6, we'll exercise the methods from table 11.10 by making a series of modifications to our myData dataset. The purpose is to see which methods generate ondata events.

Listing 11.6 Modifying an XML data hierarchy

```
<canvas width="100%" debug="true">
   <include href="myData.lzx"/>
                                                                ❶ Points
   <datapointer name="dp1" xpath="myData:/fruit/">                 to fruit
      <handler name="ondata" args="d">
         Debug.write("dp1 ondata : " + d);
      </handler>
   </datapointer>
   <datapointer name="dp2" xpath="myData:/fruit/plum/size">
      <handler name="ondata" args="d">                               Points to
         Debug.write("dp2 ondata : " + d);                           plum's
      </handler>                                                   ❷ child node
   </datapointer>

   <method name="init">
      Debug.write("Add new 'kumquat' node : " +
                dp1.addNode("kumquat", null,                 ❸ Adds
               {variety: "Nagami"}));                            kumquat      ❹ Changes
dp1.setXPath("myData:/fruit/*[5]");                                              name to
Debug.write("Update current node to be a \"kiwi\"");                             kiwi
dp1.setNodeName("kiwi");
dp1.setNodeAttribute("variety", "Arguta");
```

```
        Debug.write("Add a \"size\" child node to kiwi with
   addNodeFromPointer");
dp1.addNodeFromPointer(dp2);
Debug.write("dp1 now points to : ", dp1);
        Debug.write("set dp2 to dp1's location
            ➥with setFromPointer");
        dp2.setFromPointer(dp1);
Debug.write("Let's clean up by deleting
            ➥this \"kiwi\" node");
        dp2.deleteNode();
    </method>
</canvas>
```

❺ Adds dp2's child node to kiwi

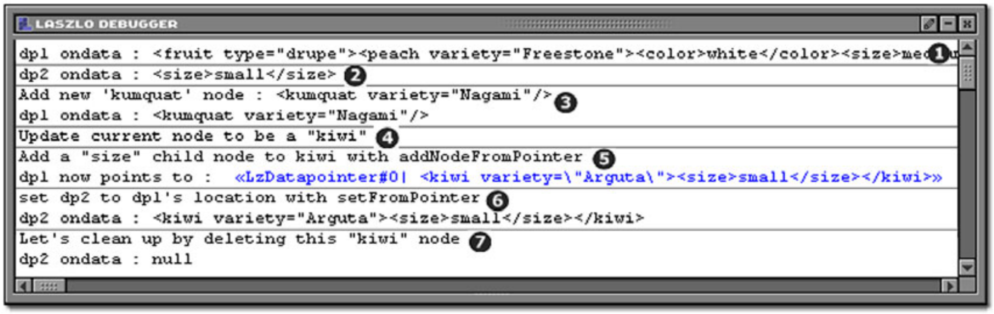
❻ Sets dp2 to point to dpl's location

❼ Cleans up by deleting kiwi

We start with two data pointers, dp1 and dp2, which initially point at the fruit node ❶ and at the plum's size node ❷. We'll then modify the `myData` dataset, by adding a new kumquat node with a Nagami variety attribute ❸ and set dp1 to point to it. Next, we'll change this node's name to kiwi and its variety to Arguta ❹, add dp2's size node to it ❺, set dp2 to point at this location ❻, and finally clean up by removing this node ❼.

Examining the debug output in figure 11.11, we see that adding or deleting a node generates an `ondata` event. This occurs even when the data pointer's location isn't changed. But an `ondata` event is not generated when the `addNodeFromPointer` method is used. This is quite likely a Laszlo oversight, so you shouldn't depend on this behavior since it could change in future releases to match its brethren.

We have shown examples of all the data pointer methods except for `setPointer`, which we'll see in section 11.3.6 when we discuss converting between the `LzDataNode` and `LzDatapointer` approaches. But first let's take a short look at the `datapath` class. Most of a data path's attributes and methods are concerned with optimization issues, so those issues are postponed until chapter 17, when we examine large datasets.



```
dp1 ondata : <fruit type="drupe"><peach variety="Freestone"><color>white</color><size>med❶u
dp2 ondata : <size>small</size> ❷
Add new 'kumquat' node : <kumquat variety="Nagami"/>❸
dp1 ondata : <kumquat variety="Nagami"/>
Update current node to be a "kiwi" ❹
Add a "size" child node to kiwi with addNodeFromPointer ❺
dp1 now points to :  «LzDatapointer#0| <kiwi variety=\"Arguta\"><size>small</size></kiwi>»
set dp2 to dp1's location with setFromPointer ❻
dp2 ondata : <kiwi variety="Arguta"><size>small</size></kiwi>
Let's clean up by deleting this "kiwi" node ❼
dp2 ondata : null
```

Figure 11.11 Listing 11.6 demonstrates various methods used to create and modify datasets with data pointers.

11.3.5 Working with the datapath tag

The `datapath` class is derived from the `datapointer` class, adding the capability to handle XPath statements that match multiple data nodes. This declarative data path is the same object that we previously used as an attribute of visible objects. For example,

```
<view datapath="myData:/fruit/plum"/>
```

is equivalent to

```
<view>
   <datapath xpath="myData:/fruit/plum"/>
</view>
```

which can be taken one step further to

```
<view>
   <datapath name="dp ">
      <method name="init">
         dp.xpath("myData:/fruit/plum");
      </method>
   </datapath>
</view>
```

However, there are restrictions on where and how data paths can be used. Here's an example that shows how declaring a data path as a top-level tag can potentially lead to problems:

```
<canvas>
   <include href="myData.lzx"

   <datapath xpath="myData:/fruit/plum"/>       <- Returns a single match: legal
   <datapath xpath="myData:/fruit/peach"/>      <- Returns multiple matches: illegal
</canvas>
```

Since there is only a single `plum` data node in the `myData` dataset, the `datapath` tag is equivalent to a `datapointer` tag, and can be declared at the top level. But because there are multiple `peach` data nodes, a top-level `datapath` tag attempts to create a replication manager that controls a matching number of cloned instances of its parent container. In this case, the parent container is the `canvas` tag. The canvas is the root node and contains the application, so there can't be multiple instances of it. As a result, a top-level data path tag isn't allowed, producing a runtime error.

LZX data-access classes are used most effectively in concert, that is, by combining the `LzDataNode`-derived and `LzDatapointer`-derived approaches. But to do this, we need a way to easily convert from one approach to the other.

11.3.6 Converting between data pointers and data nodes

The LzDatapointer class contains two mechanisms, shown in figure 11.12, for converting from data pointer to data node representation and vice versa. The data pointer's p attribute allows any LzDataNode-derived object to be retrieved from a data pointer. The inverse operation is accomplished through the data pointer's setPointer method; a data pointer can be set to point to the data node represented by any LzDataNode-derived object.

LzDatapointer's p attribute serves as a bridge from data pointer objects to data node objects. Since the data type of the p attribute is LzDataElement, this provides access to all the LzDataElement attributes and methods. This provides a way for LzDatapointer objects to easily access all the LzDataNode properties. Here's an example of this integrated approach, where a data pointer uses its p attribute to directly access data node attributes and methods:

```
<datapointer name="dp" xpath="myData:/fruit">
   <method name="init">
      var name = dp.p.nodeName;
      var ele  = dp.p.getElementsByTagName("plum");
   </method>
</datapointer>
```

The setPointer method provides the reverse conversion from data node objects to data pointer objects. This method takes a data node as an argument and sets a

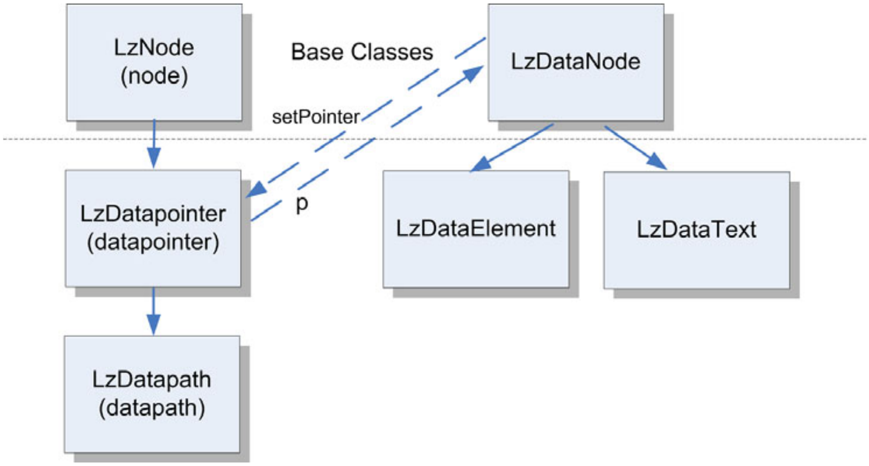


Figure 11.12 The p attribute allows an LzDataNode-derived object to be retrieved from a data pointer. The data pointer's setPointer method allows the reverse; a data pointer can be set to point to the data node represented by an LzDataNode-derived object.

data pointer to point to its location. Because this changes the location of the data pointer, it generates an ondata event. See listing 11.7.

Listing 11.7 Converting from data nodes to data pointers

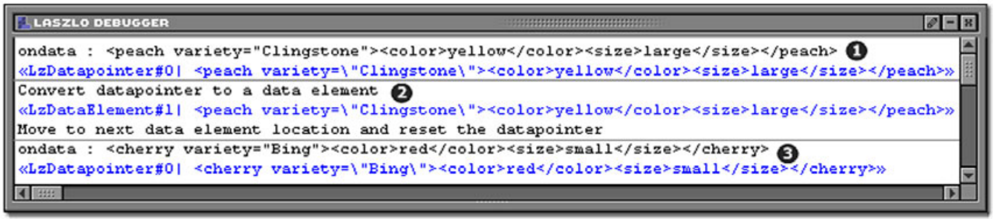
```
<canvas width="100%" debug="true">
   <include href="myData.lzx"/>

   <datapointer name="dp" xpath="myData:/fruit/peach[2]">
      <handler name="ondata" args="d">
         Debug.write("ondata : " + d);
      </handler>
   </datapointer>

   <method name="init">
      Debug.write(dp);
      var ele = dp.p;
      Debug.write("Convert datapointer to a data element");
      Debug.write(ele);
      ele = ele.getNextSibling();
      Debug.write("Move to next data element location and reset the
  datapointer");
      dp.setPointer(ele);
      Debug.write(dp);
   </method>
</canvas>
```

❶ **Converts from data pointer to dataelement**

❷ **Moves to next sibling data node**

❸ **Converts from data element to data pointer**

The data pointer dp is initially set ❶ to point to the second peach data node. Its p attribute is used to set ❷ its location to an LzDataElement object called ele. The LzDataElement's getNextSibling method moves to the next sibling data element called cherry. Finally, this data element is passed ❸ as an argument to the setPointer method to reset the data pointer to point to the cherry data node. In figure 11.13, you can see that ondata events are generated when the data pointer is initially set and again when its position is updated.



```
ondata : <peach variety="Clingstone"><color>yellow</color><size>large</size></peach> ❶
«LzDatapointer#0| <peach variety=\"Clingstone\"><color>yellow</color><size>large</size></peach>»
Convert datapointer to a data element ❷
«LzDataElement#1| <peach variety=\"Clingstone\"><color>yellow</color><size>large</size></peach>»
Move to next data element location and reset the datapointer
ondata : <cherry variety="Bing"><color>red</color><size>small</size></cherry> ❸
«LzDatapointer#0| <cherry variety=\"Bing\"><color>red</color><size>small</size></cherry>»
```

Figure 11.13 A data pointer can be used to retrieve an instance of the data node to which it points, and a data node can be used to set a data pointer.

NOTE *Collaborative working relationship between data pointers and nodes*—You will generally find it easiest to manipulate XML data structures by using a data pointer's XPath capabilities to navigate to a selected location and then using its `p` attribute to return the data node for the location. The data node methods can be used to modify the data structure without causing spurious `ondata` events. Finally, the data node can be used as an argument to a data pointer's `setPointer` method to get back to working with a data pointer.

These conversion tools provide an avenue for moving easily between the data pointer and data node domains in a constructive manner.

11.3.7 Checking updates with rerunxpath

An XPath expression provides a link between the view and model layers, so it must correctly respond to updates to the dataset and to the XPath expression itself. Here is a list of the different situations that an XPath expression can handle:

- The data in the data node is updated or removed.
- A data node is deleted.
- A data node is added, resulting in the XPath expression returning multiple matches.
- The XPath expression itself is updated.

In each case, the data pointer must point to the correct location and, for the first three situations, an `ondata` event must be sent to the bound object.

However, there is one situation that the XPath expression can't handle. This occurs with a wildcard predicate and the addition or deletion of a sibling node, changing its offset value. An example should make this clear. Suppose we have

```
xpath="myData:/fruit/*[1]/name()"
```

Inserting a new data node changes the location. With an expression like

```
xpath="myData:/fruit/*[2]/name()"
```

deleting the first data node would change the XPath location. Neither of these situations generates an `ondata` event, unless the `rerunxpath` attribute is set. Listing 11.8 demonstrates this situation.

Listing 11.8 Using `rerunxpath` to send an `ondata` event

```
<canvas>
   <include href="myData.lzx"/>

   <view name="main" x="10" y="10">
```

```
      <text name="uptodate"/>
      <text name="outofdate"/>
      <button text="update Dataset">
         <handler name="onclick">
            var last = ptr.xpathQuery("*[1]");
            ptr.p.removeChild(last);
         </handler>
      </button>
      <simplelayout axis="y" spacing="3"/>
   </view>

   <datapointer name="ptr" xpath="myData:/fruit"/>
   <datapointer xpath="myData:/fruit/*[1]/name()"
                      rerunxpath="true">
      <handler name="ondata" args="d">
         main.uptodate.setText(d);
      </handler>
   </datapointer>
   <datapointer xpath="myData:/fruit/*[1]/name()">
      <handler name="ondata" args="d">
         main.outofdate.setText(d);
      </handler>
   </datapointer>
</canvas>
```

❶ **Removes first data node from dataset**

❷ **Evaluates XPath multiple times**

❸ **Evaluates XPath only once**

The data pointer's xpathQuery method is used ❶ to obtain a copy of the first data element peach. This data element is then removed, which changes the meaning of * [1]. The first data node is now plum. But since the XPath expression has already been parsed, it isn't aware of this change unless it's reevaluated. Setting rerunxpath forces the XPath expression to be reevaluated every time the contents of the dataset change. At ❷ and ❸, two data pointers are declared, one with rerunxpath set and the other with it reset. The rerunxpath attribute ensures that its XPath expression is reevaluated when the peach data node is removed, updating the XPath expression to point to the plum data node. It also generates an ondata event, updating the text object to display plum. The other data pointer is blind to all of these proceedings and continues to display *peach*, as figure 11.14 shows.

The rerunxpath attribute shouldn't be set needlessly, because that forces the XPath expression to be reevaluated on each access, which can impact performance. As we've seen, it's only needed in a limited number of situations.

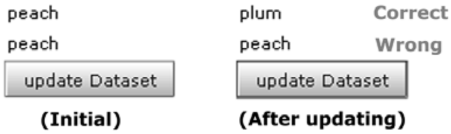


Figure 11.14 This output from listing 11.8 shows that the top text object has its rerunxpath attribute set so it accurately reflects changes to the XML dataset.

Before applying this material to the Laszlo Market, we need to examine the operation of the replication manager in the next section.

11.4 Advanced replication manager issues

Chapter 10 introduced us to how a replication manager supports multiple element matches. Now let's examine the internal operation of the replication manager and its associated clones and nodes arrays. We'll start by contrasting the timing sequence of a data path that returns a single match versus a data path that returns multiple matches, thereby instantiating a replication manager.

The initialization sequence for an object with a data path that returns a single match is displayed in figure 11.15. The completion of each step is indicated by the generation of an event. This sequence occurs synchronously in a bottom-up order where the `datapath` tag is first initialized, followed by the initialization of its parent object and its reception of an initial `ondata` event.

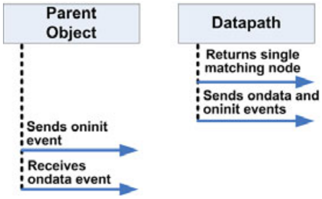


Figure 11.15 A `datapath` object that only returns a single match will complete its initialization, and afterward its parent object will be initialized and receive an initial `ondata` event.

The multistep initialization sequence for a replication manager and its clones is displayed in figure 11.16. When a data path returns multiple matches, a replication

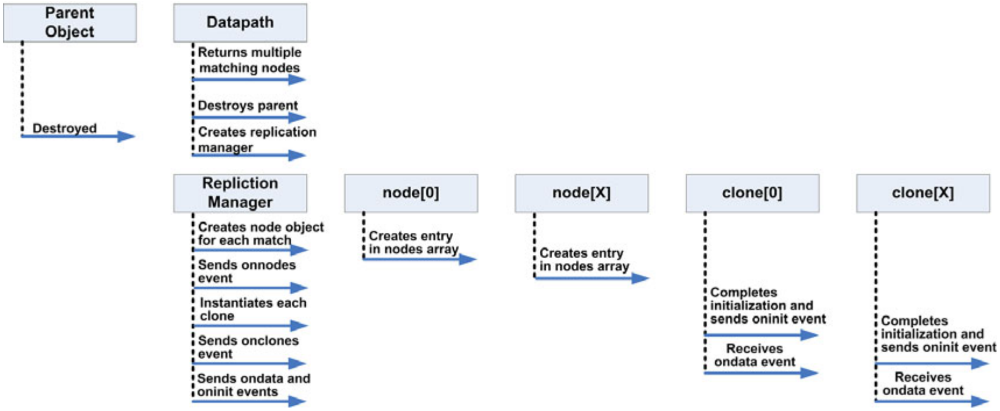

Parent Object
Destroyed
Datapath
Returns multiple matching nodes
Destroys parent
Creates replication manager
Repliction Manager
Creates node object for each match
Sends onnodes event
Instantiates each clone
Sends onclones event
Sends ondata and oninit events
node[0]
Creates entry in nodes array
node[X]
Creates entry in nodes array
clone[0]
Completes initialization and sends oninit event
Receives ondata event
clone[X]
Completes initialization and sends oninit event
Receives ondata event

Figure 11.16 When a data path returns multiple matching data nodes, its first action is to destroy the parent object and replace it with a newly created replication manager that is responsible for governing the creation of a matching set of node and clone objects. When all of the nodes are created, an `onnodes` event is sent. When all of the clones are created, an `onclones` event is sent. Finally, an `ondata` event is sent to each of these clones.

manager is created and its parent object is destroyed and replaced with a matching number of clones. A replication manager's initialization sequence requires an additional set of onnodes and onclones steps.

1 Build the nodes array (onnodes event).
2 Instantiate each clone and build the clones array (onclones event).
3 Initialize the LzReplicationManager object (oninit event).

The onnodes event is sent when a nodes array containing the matching data nodes array is created. The onclones event is sent when each of the clones has begun instantiation and when a clones array containing the instantiated clone objects is created.

Although the execution of these three steps is synchronous, the instantiation of each clone is asynchronous. In other words, the replication manager starts the instantiation of each clone, but doesn't wait for each clone to complete its initialization. The size and complexity of a clone can significantly affect its timing, allowing it to complete prior to or after the replication manager.

Listing 11.9 demonstrates the differing initialization sequences for a data path that can return either a single or multiple data nodes. When a button is clicked, it dynamically updates an object's datapath XPath expression, switching it from returning a single node to multiple nodes. Although the same set of declarative handlers are used for both cases, each has a different context, so these event handlers are used in different ways. To show their relationship, two annotation sequences are used; the first 1a 1b 1c shows the progression for a single match, and the second 2a 2b 2c for multiple matches.

Listing 11.9 Timing issues with the replication manager

```
<canvas debug="true">
   <include href="myData.lzx"/>

   <view name="first">
      <datapath name="dp" xpath="myData:/fruit/plum">
         <handler name="oninit" args="d">
            Debug.write("datapath oninit: ", d);
         </handler>
         <handler name="onnodes" args="d">
            Debug.write("onnodes: ", d);
        </handler>
        <handler name="onclones" args="d">
           Debug.write("onclones: ", d);
        </handler>
      </datapath>
```

1a Returns single match

2c Sends onclones event

2a Returns multiple matches

2b Sends onnodes event

```
         <handler name="oninit" args="d">
            Debug.write("oninit : ", d);
         </handler>
         <handler name="ondata" args="d">
            Debug.write("ondata : " + d);
         </handler>
      </view>
      <button x="10" y="80" text="Push">
         <handler name="onclick">
            first.dp.setXPath("myData:/fruit/peach");
         </handler>
      </button>
   </canvas>
```

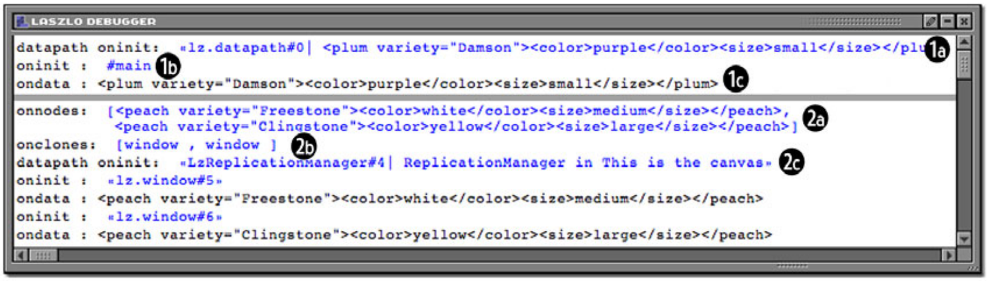
1b **Completes parent object initialization**

1c **Receives ondata event for parent object**

The simple object initialization sequence is displayed in the upper portion of the debug window shown in figure 11.17, while the multistep initialization sequence is displayed in the lower portion.

The replication manager's first action is to create a nodes array 1b containing the matching data nodes. Next, the cloned objects are instantiated, resulting in 2b an onclones event. Finally, the initialization of the replication manager is displayed 3b.

In the following sections, we'll see how the onnodes and onclones events can be used in an application.



```
datapath oninit:  «lz.datapath#0| <plum variety="Damson"><color>purple</color><size>small</size></plu
oninit :  #main
ondata : <plum variety="Damson"><color>purple</color><size>small</size></plum>
onnodes:  [<peach variety="Freestone"><color>white</color><size>medium</size></peach>,
           <peach variety="Clingstone"><color>yellow</color><size>large</size></peach>]
onclones:  [window , window ]
datapath oninit:  «LzReplicationManager#4| ReplicationManager in This is the canvas»
oninit :  «lz.window#5»
ondata : <peach variety="Freestone"><color>white</color><size>medium</size></peach>
oninit :  «lz.window#6»
ondata : <peach variety="Clingstone"><color>yellow</color><size>large</size></peach>
```

Figure 11.17 The top half shows the initialization timing of a data-path-bound view object returning a single match. The bottom half shows the timing for multiple matches that requires the instantiation of a replication manager. This debug output has been digitally manipulated for better presentation.

11.4.1 Filtering with onnodes

Requirements to filter data retrieved from a dataset are very common. Often, for example, a common text string or some HTML tags must be added or stripped to correctly display data. Although an `ondata` or `ontext` event handler could be used, this requires a separate event handler for each data element. Since the `onnodes` event handler has access to every matching data element, this allows all data modifications to be handled in a single location. Listing 11.10 shows an example.

Listing 11.10 Filtering data nodes with `onnodes`

```
<canvas>
   <dataset name="sampleDS">
      <item>Line 1## sample line of text</item>
      <item>Line 2## yet another line</item>
   </dataset>

   <node>
      <datapath xpath="sampleDS:/item">
         <handler name="onnodes">
            for (var i = 0; i &lt; nodes.length; i++) {
               var line = nodes[i].getLastChild().data;
               var pos = line.indexOf("##");
               if (pos)
                  nodes[i].getFirstChild().data = line.substr(pos+3);}
         </handler>
      </datapath>
   </node>
   <text resize="true" datapath="sampleDS:/item/text()"/>
   <simplelayout/>
</canvas>
```

Contains extra line numbers

Removes line number prefix

Displays data without line numbers

In the previous section, we saw that within the cloning process, the first step was to create the matching nodes and then send an `onnodes` event. Because this step occurs so early, it provides a centralized location to filter incoming data. In this case, the input data nodes contain an undesired artifact, line numbers, from another system. Our `ondata` filter will remove these line numbers so they won't be displayed. Now we are assured that all subsequently cloned objects will receive clean data.

Let's now move on to the next set of initializations in the cloning process: the `onclone` events.

11.4.2 Checking clone instantiation with onclones

A declarative tag's oninit event normally signals that this object has completed initialization and is ready for use. However, the asynchronous nature of clone instantiation turns checking for a clone's initialization completion into a two-step process. First we need to receive an onclones event to indicate that all the clones have been instantiated. Then we need to wait for the last clone to send its oninit event. Only then are we assured that all cloned objects have completed initialization and are ready for use.

Listing 11.11 demonstrates how to perform this check for clone completions and illustrates a pitfall of attempting to using clones before their initialization has completed.

Listing 11.11 Ensuring that clone initialization is complete

```
<canvas>
   <dataset name="tabDS">
      <item>one</item>
      <item>two</item>
      <item>three</item>
      <item>four</item>
   </dataset>

   <button y="40" text="Add Tabelements">
      <method event="onclick">
         A.tabA.setDatapath('tabDS:/item');
         B.tabB.setDatapath('tabDS:/item');
      </method>
   </button>
   <tabslider name="A" y="10" width="100" height="120" spacing="2">
      <tabelement name="tabA" text="$path{'text()'}">
         <datapath>
            <method event="onclones">
               parent.select(this.getCloneNumber(2));
            </method>
         </datapath>
      </tabelement>
   </tabslider>
   <tabslider name="B" y="10" width="100" height="120" spacing="2">
      <tabelement name="tabB" text="$path{'text()'}">
         <datapath>
            <method event="onclones">
               this.doneDel = new LzDelegate(this, "waits")
               this.doneDel.register(clones[clones.length - 1],
                                     "oninit")
            </method>
            <method name="waits">
```

Dynamically applies data path

Selects list entry immediately

Waits for last clone's oninit event

```
               parent.select(this.getCloneNumber(2));
            </method>
         </datapath>
      </tabelement>
   </tabslider>
   <simplelayout inset="10" axis="x" spacing="20"/>
</canvas>
```

Selects completed list entry

Both lists displayed in figure 11.18 are initially empty, because an XPath expression hasn't been supplied. An XPath expression is dynamically applied to both of the tabsliders, when the user clicks the Add Tabelements button.

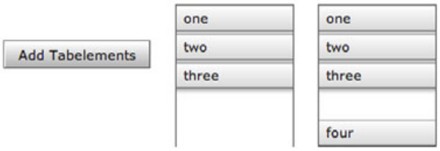


Figure 11.18 This contrasts the effects of waiting for the final `onclones` event to complete versus immediately proceeding with processing. In the left figure, the selected cloned object hasn't completed its initialization and isn't ready for display.

In the first tabslider, shown on the left, the parent immediately attempts to open a tabelement before clone initialization has completed. In the second tabslider, a delegate has been registered to call the `waits` method when the last clone has completed its initialization and sends its `oninit` event. The `waits` method then performs the selection.

Initialization for cloned instances is only slightly more complex than for a single instance; instead of checking for a single `oninit` event, the `oninit` event of the last cloned instance must be checked.

We're now ready to apply this chapter's material to the Laszlo Market, using the master-detail design pattern to coordinate the display of multiple windows.

11.5 Master-detail design pattern

In chapter 10, we created the initial layout for the `dsProducts` dataset and used it to supply products for a Product List window. Now we want to display a full description along with technical details for each product. But the Product List window doesn't contain enough room to display all this information. Instead a separate window needs to be coordinated to display auxiliary information for a selected item in the Product List window. This situation occurs frequently enough to be addressed with a design pattern known as *master-detail.*

In the master-detail design pattern, a *master window* contains a summary or list of items. When a particular item is selected, additional information for that item is displayed in a *detail window*. This relationship can scale to include multiple detail windows, where each window specializes in a particular aspect of a selected item.

Since the content of each window is controlled through a data binding, coordinating the contents of these windows is accomplished through a shared data pointer. The master window updates this data pointer's location in a dataset, to reflect the user's current selection. This results in a data event being sent to each of the detail windows, which use this new data location to update the displayed contents for each of their windows.

This example of vertically oriented communication between objects is analogous to the constraint mechanism's horizontally oriented communications in coordinating the aircraft formation seen in chapter 2. In both cases, the result is a coordinated action driving the presentation of visible objects.

11.5.1 Implementing master-detail in Laszlo Market

In the Laszlo Market, the Product List window serves as the master and the Product Details window serves as the detail. The detail window displays description and technical information that conforms to the current product selection within the master window. Changing the selection in the master window causes the information displayed in the Product Details window to also be updated.

This relationship is based upon a shared data pointer. We'll declare a data pointer at the top level, so it can be easily shared by different objects:

```
<datapointer name="productdp"/>
```

Any class can create an ondata event handler that references the productdp data pointer as the publisher of its ondata events. Within the master window, whenever this data pointer is repositioned to point at a new data node, it sends an ondata event to every class referencing this data pointer in its event handler. Here's a simple example:

```
<class name="details">
   <handler name="ondata" reference="productdp">
    Debug.write("productdp ondata");
   </handler>
</class>
```

Now any number of detail windows receive ondata events from our master window.

To send these ondata events, a master window positions the data pointer to point at the data node corresponding to a user's selection by using the data pointer's setFromPointer method. In table 11.10, we saw that using this method to change the data pointer's location results in an ondata event being sent to all event listeners:

```
<class name="productgrid" extends="grid" … >
    …
    <handler name="onselect">
       productdp.setFromPointer(this.getSelection()[0]);
    </handler>
</class>
```

When there are multiple listeners, the order in which they receive this event is indeterminate, but the processing of the ondata event handlers for all listeners proceeds synchronously. As a result, all event-related processing will complete before processing resumes in the master window's onselect event handler.

In a detail window, the ondata event handler uses the location of the productdp data pointer as a base context to access its particular data locations. Simple XPath expressions can then be used to navigate from this context to specific data locations, similar to navigating a normal file system. Listing 11.12 shows how to do this.

In our case, there is only one Product Details window, but we can easily add more windows.

Listing 11.12 The master-detail design pattern in the Laszlo Market

```
<class name="details" extends="window">
   <attribute name="prod_title" type="string" value=""/>
   <attribute name="description" type="string" value=""/>

   <handler name="ondata" reference="productdp">
      if (productdp.data) {
         top.setVisible(true);
         var selectedImage =
               productdp.getNodeAttribute("image");
         this.setAttribute("prod_title",
               productdp.getNodeAttribute("title"));
         productdp.setXPath("description");
         this.setAttribute("description",
               productdp.getNodeText());
         productdp.setXPath("..");
         top.col_1.image.setResource(
               gController.IMAGESFOLDER + selectedImage);
      }
      else this.top.setVisible(false);
   </handler>
</class>
```

Checks for data to display

Moves among attributes and child nodes

Displays a blank screen

11.5.2 When to use a static layout

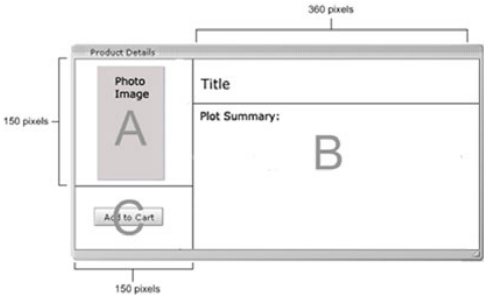


Figure 11.19 Providing static dimensions for the photo image causes the Add to Cart button to remain stationary until the image has been loaded.

In chapter 6 we discussed the advantages of a flexible layout when working with pre-loaded image assets compiled into the application. This allows all display elements to be loaded and displayed simultaneously. However, dynamically loaded images introduce timing differences. Now a static layout is needed to ensure the layout doesn't reconfigure as the images appear. In particular, we want the Add to Cart button to be fixed in place. This problem can be fixed by specifying a static alignment for the image so the Add to Cart button remains stationary regardless of whether the image has been loaded. Figure 11.19 illustrates the benefits of this approach.

While the other parts of the window can reconfigure themselves to accommodate sizing changes, the left column, consisting of the photo image and the button, has a fixed spatial layout. As a result, when the window is resized the title and summary information still wrap, providing a measure of flexibility. Listing 11.13 shows the steps for laying out the contents of this details window.

Listing 11.13 Using a static layout to fix the Add to Cart button in position

```
<class name="details" extends="window" fontsize="14" fontstyle="bold">
   <attribute name="prod_title" value="title" type="text"/>
   <attribute name="description" value="desc" type="string"/>

   <view name="top" height="${immediateparent.height}"
            width="${immediateparent.width}" visible="false">
      <view name="col_1">
         <thumbnail name="image" x="10" y="10" height="150"/>
         <button x="10" y="170" text="Add to Cart"/>
      </view>
      <view name="col_2" x="150" width="350">
         <text text="${classroot.prod_title}" width="100%"
               resize="true" fontsize="24" multiline="true"/>
         <view width="100%">
            <text text="Plot Summary:"/>
            <text text="${classroot.description}" width="100%"
                  fontstyle="plain" fontsize="12" multiline="true"/>
```

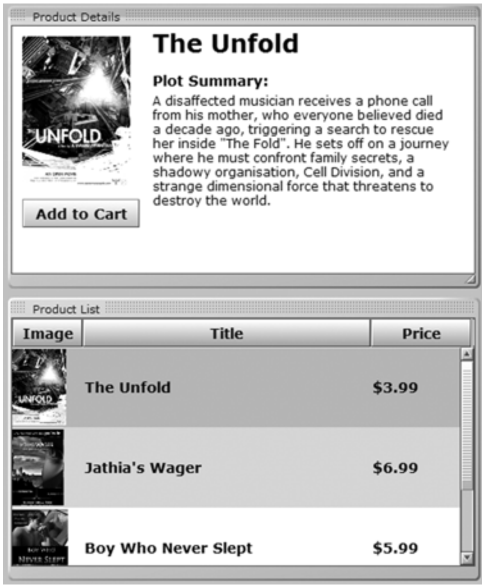
Contains images and button

Contains title and summary

```
            <simplelayout axis="y"/>
         </view>
         <simplelayout axis="y" spacing="10"/>
      </view>
      <simplelayout axis="x" spacing="10"/>
   </view>
   <handler name="ondata" reference="productdp">
      …
   </handler>
</class>
```

The Product List and Product Details windows, shown in figure 11.20, conform to the master-detail design pattern. Whenever a selection is made in the Product List window, indicated by a darker background, the corresponding product description appears in the Product Details window. This relationship holds for all of the products in the Product List window.

The master-detail design pattern is extensible, so it's a relatively simple matter to accommodate additional detail windows. For example, if we decide to separate Plot Summary and Technical Specs into separate windows, we can simply add a window-based class that references the `productdp` data pointer in its `ondata` event handler to be another detail window.



The Unfold

Plot Summary:
A disaffected musician receives a phone call from his mother, who everyone believed died a decade ago, triggering a search to rescue her inside "The Fold". He sets off on a journey where he must confront family secrets, a shadowy organisation, Cell Division, and a strange dimensional force that threatens to destroy the world.

Image	Title	Price
	The Unfold	$3.99
	Jathia's Wager	$6.99
	Boy Who Never Slept	$5.99

Figure 11.20 The relationship between the Product List (master) window and the Product Details (detail) window is shown here. The current selection is the Unfold video, indicated by the darker background. A detailed display is provided for this selection in the Product Details window.

11.6 Summary

This chapter builds on the XML data path material in the previous chapter, which introduced new concepts and demonstrated the high-level behavior of data-path-bound visible objects. We explored the underlying classes that provide this data-binding functionality. Our first exposure to data-path-bound visual objects limited us to a static connection to a specific data node. Building on this, we saw how to manipulate the data elements of an XML hierarchy and traverse its tree. We also modified the XML tree structure itself by adding, modifying, or deleting data nodes. Finally, we introduced different ways to control the replication process involved with data paths and the replication manager. Completing this chapter, you have seen most of what you need to use and manipulate data paths. This sets the stage for a later chapter on optimization.

Applying the master-detail design pattern to the Laszlo Market provides the building blocks for additional capabilities to be presented in the next chapter. With our current prototype, we have limited ourselves to only reading from a dataset. In the next chapter, we'll begin to modify the contents of different datasets by adding and deleting data nodes. These dataset-related changes provide a higher level of inter-window functionality in the Laszlo Market.

12 Scoreboarding the shopping cart

This chapter covers

- Building a grid window
- Creating a scoreboarding shopping cart
- Interfacing to the mouse scroll wheel
- Interfacing to the right mouse button
- Creating an image icon dragger

> *Whoever said money can't buy happiness simply didn't know where to go shopping.*
>
> —Bo Derek, actress

One goal of the Laszlo Market is easy selection of items for the shopping cart. At the same time, there are a multitude of windows and various keyboard and mouse-based input devices to be supported. One unifying mechanism with a standard, easily accessible interface is needed to update the shopping cart. For this, we turn to the *scoreboard*. Although a scoreboard isn't a formal design pattern, scoreboarding techniques are widely used across many interactive applications. The purpose of a scoreboard is to collect and tabulate information through a standard interface easily accessible by a wide range of input devices.

The most obvious application of a scoreboard is in a game with scores kept for players—the number of monsters killed. There are many different types of scoreboards. We'll be using a *totals* scoreboard that keeps track of the number of purchased products and their total price. There are also *actions* scoreboards that record a customer's actions—for example, to support the undo function in an application.

In the Laszlo Market, the shopping cart serves as our scoreboard to record purchase requests and deletions from input sources. It needs to tabulate all interactions affecting the contents of the shopping cart. Input can come from any of these sources:

- The Add to Cart button
- Drag-and-drop operations for adding items
- Manual input
- Right-click mouse input
- Drag-and-drop operations for deleting items

The shopping cart maintains a list of products to be purchased, their quantity, and a total price. It also identifies duplicate products and updates the quantity to ensure that each product is unique. The total price is recalculated when the shopping cart contents are updated.

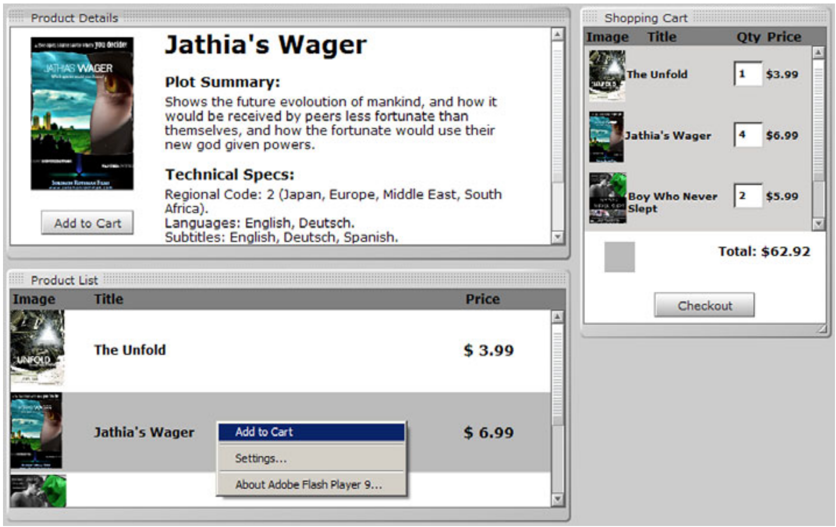
This chapter begins by defining a scoreboard. But before we apply scoreboarding techniques to the Market, we'll reimplement the Product List window with a lower-level window component. Next, we'll update the shopping cart to support a standard scoreboard interface. Finally, we'll integrate our input sources to interface with the shopping cart.

12.1 How a scoreboard works

The scoreboard shares many similarities with the master-detail design pattern. Where the master-detail pattern *displays* a user selection, a scoreboard *records* a user selection. The master-detail pattern scales to support multiple display windows, while the scoreboard pattern performs the inverse by scaling to support multiple reporting sources. Like master-detail, a single dataset is used by all participants to contain their input values. But a scoreboard introduces another dataset to record each input. In our case, a totals scoreboard tabulates the current number of purchases and the total price. Other types of scoreboards could also easily be supported.

A scoreboard can be applied to many situations. Its simplest use involves a button that just triggers a collection. Manual entry modifies shopping cart contents through the quantity field. A more sophisticated interface involves the drag-and-drop network we built in chapter 9, where products are added by dragging them to a shopping cart. Purchased products can be deleted by dragging and dropping into the trash. Finally, to support all input sources, a user can right-click and choose Add to Cart to add a product to the shopping cart. All these input sources interact with the shopping cart through a common interface.

When we've finished building our shopping cart, it will contain the functionality shown in figure 12.1.



Jathia's Wager

Plot Summary:

Shows the future evoloution of mankind, and how it would be received by peers less fortunate than themselves, and how the fortunate would use their new god given powers.

Technical Specs:

Regional Code: 2 (Japan, Europe, Middle East, South Africa).
Languages: English, Deutsch.
Subtitles: English, Deutsch, Spanish.

Image	Title	Qty	Price
	The Unfold	1	$3.99
	Jathia's Wager	4	$6.99
	Boy Who Never Slept	2	$5.99

Total: $62.92

Image	Title	Price
	The Unfold	$ 3.99
	Jathia's Wager	$ 6.99

Figure 12.1 The Shopping Cart window uses a scoreboarding technique to allow input sources to easily report product updates. The right-click input source is currently displayed.

But before we incorporate this scoreboard functionality in our application, let's rebuild our Product List window using a simple window class.

12.2 Reimplementing the Product List window

In chapter 10, we created the Product List window using the grid component, since it offered a convenient way to list the contents of a static dataset. Although the grid component provides these basic features, it's important to understand how to add these features yourself. We'll now explore how to add the following grid-related features to a view or window:

- Title headers for a window
- Column sorting
- Alternating background colors for rows
- Window scrolling
- A mouse-based scroll wheel
- Selections

Lower-level components, such as views and windows, provide flexibility that will be needed in subsequent chapters, when we start working with dynamic datasets and dealing with issues such as optimization.

We'll start with the title headers.

12.2.1 Creating the title header

We'll define a new class productwin to replace the productgrid instance. The only difference in its interface is that the data path location is now set in the productwin:

```
<canvas>
   ...
   <window name="productlist"
           title="Product List"
           width="55%" height="50%"
           x="${parent.browsesearch.width}"
           y="${parent.details.height}">
      <productwin name="pwin" width="100%" height="100%"/>
   </window>
   ...
</canvas>
```

Next we create its subsidiary classes, starting with the columnheader class that provides the column headers appearing in the title header:

```
<class name="columnheader" extends="text"
       bgcolor="gray" fontstyle="bold" fontsize="12"/>
```

The columnheader class is used in the titleheader class for the Image, Title, and Price column headers. The outer header columns have a fixed width, and the middle column has a variable length set by the stableborderlayout tag:

```
<class name="titleheader">
   <columnheader width="80" text="Image"/>
   <columnheader text="Title"/>
   <columnheader width="100" text="Price"/>
   <stableborderlayout axis="x"/>
</class>
```

The titleheader instance has a height of 30 pixels and a width that stretches across the window:

```
<canvas>
   <include href="library.lzx"/>
   <window name="productlist" title="Product List"
           width="55%" height="50%">
      <titleheader width="100%" height="30"/>
   </window>
   ...
</canvas>
```

The title header is now complete. The next step is to populate the rows for the table.

12.2.2 Populating a table row

We previously saw how the rows of a table are created through a single instance replicated through its bound data path. The productrow class has an attached data path that matches each product, producing a replication manager to govern the matching number of productrow clones:

```
<productrow width="100%"
            datapath="dsProducts:/products/product"/>
```

Listing 12.1 shows the productrow layout that matches our earlier grid-based layout.

Listing 12.1 Populating the Product List window with rows

```
<class name="productrow">
   <thumbnail name="image" datapath="image/text()"
              width="80" maxheight="75" maxwidth="75">
      <method name="applyData" args="d">
         this.setSource( gController.IMAGESFOLDER + d);
      </method>
   </thumbnail>
   <text valign="middle" datapath="title/text()"/>
   <text width="100" valign="middle" fontstyle="bold"
```

Specifies thumbnail image

Contains title

```
         fontsize="14" datapath="price/text()">
      <method name="applyData" args="d">
         if (d.length == 4) this.setText( "$ " + d );
         else this.setText( "$" + d );
      </method>
   </text>
   <stableborderlayout axis="x"/>
   <handler name="ondata" args="d">
      var pos = this.datapath.xpathQuery('position()') * 1;
      if (pos % 2 == 0) this.setAttribute('bgcolor', 0xcccccc);
      else this.setAttribute('bgcolor', 0xffffff);
   </handler>
</class>
```

Specifies price

Alternates row background colors ❶

To produce the alternating background colors, we've added an ondata event handler. Each matching product data node generates a data event that is sent to this event handler ❶. In this handler, the XPath position function is used to obtain the offset of each row, modulo two, to specify an alternating set of background colors. Now that we have rows to display, the next step is to make the table columns sortable.

12.2.3 Sorting table columns

Each column header that contains a headerid is sortable. The first time the column head is clicked, it is sorted in ascending order; on the next click, sorting is performed in descending order. Subsequently, it switches between ascending and descending sorts.

```
<class name="columnheader" extends="text" fontstyle="bold">
   <attribute name="headerid" type="string" value=""/>
   <handler name="onclick">
      if (this.headerid != "") {
         main.productlist.pwin.sortBy(this.headerid); }
   </handler>
</class>
```

Sorts column by specified headerid

Let's assume we're sorting by title, so the headerid is set to title. When a column header is clicked, our sortBy method is called with the sorted column's headerid. This results in setOrder sorting the rows by the XPath title attribute. The sort order can be set to either ascending or descending with the setComparator method:

```
<class name="productwin">
   <attribute name="lastsort"/>
   <method name="sortBy" args="sortpath">
      var listdp = this.products.datapath;
      listdp.setOrder('@' + sortpath);
```

```
         if (this.lastsort == sortpath) {
            listdp.setComparator('descending');
            sortpath = ''; }
         else
            listdp.setComparator('ascending');
         this.lastsort = sortpath;
      </method>
      …
   </class>
```

A limitation of this sorting is that it's a dictionary sort, which can't sort numbers equal to or greater than 10. In chapter 17, you'll learn how to specify a comparator function to perform arithmetic sorting. Now that we have a sorted display, the next step is to add a scrollbar to this window.

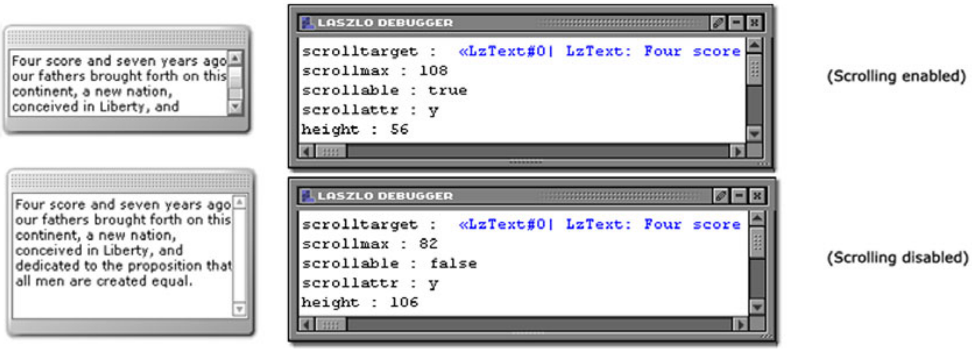
12.2.4 Basics of a scrollbar

Any view or window object can have a scrollbar attached to it, allowing text and images to be scrolled. A scrollbar can be directly added to a window to display static text. Listing 12.2 shows a window with an enabled scrollbar that scrolls through static text.

Listing 12.2 Adding a scrollbar for statically defined text

```
<canvas debug="true">
   <window height="100">
      <text width="150" multiline="true">
         Four score and seven years ago our fathers brought forth on
         this continent, a new nation, conceived in Liberty, and
         dedicated to the proposition that all men are created equal.
      </text>
      <scrollbar>                    <—— Attaches scrollbar
         <handler name="oninit">
            Debug.write("scrolltarget : ", this.scrolltarget);
            Debug.write("scrollmax : " + this.scrollmax);
            Debug.write("scrollable : " + this.scrollable);
            Debug.write("scrollattr : " + this.scrollattr);
            Debug.write("height : " + immediateparent.height);
         </handler>
      </scrollbar>
   </window>
</canvas>
```

Figure 12.2 compares an enabled scrollbar to a disabled scrollbar. The difference between their states is the `height` attributes: 100 versus 150 pixels. In the first case, the inner window's height is less than the scrolled text (`scrollmax`) to be displayed,



Four score and seven years ago our fathers brought forth on this continent, a new nation, conceived in Liberty, and

```
scrolltarget :  «LzText#0| LzText: Four score
scrollmax : 108
scrollable : true
scrollattr : y
height : 56
```

Four score and seven years ago our fathers brought forth on this continent, a new nation, conceived in Liberty, and dedicated to the proposition that all men are created equal.

```
scrolltarget :  «LzText#0| LzText: Four score
scrollmax : 82
scrollable : false
scrollattr : y
height : 106
```

Figure 12.2 Two examples of correctly functioning scrollbars are shown to illustrate the values of the critical scrollbar attributes. In the top example, the value of the `scrollmax` attribute is greater than the height of the window, so `scrollable` is true and scrolling is enabled. In the bottom example, the value of the `scrollmax` attribute is less than the height, so scrolling is disabled. See listing 12.2.

which produces an enabled scrollbar; in the second case, the inner window's height is greater than its contents, so scrolling is disabled.

Table 12.1 displays the complete list of scrollbar attributes. The attributes that determine whether scrolling is enabled are `scrolltarget`, `scrollmax`, and `scrollable`. The `scrollable` attribute specifies whether the scrollbar is enabled or disabled. The `scrolltarget` attribute specifies the child view being scrolled. The `scrollmax` attribute gives the height of the parent clipped view. The `scrollattr` attribute specifies whether this is a vertical or horizontal scrollbar. The `stepsize` attribute controls the number of pixels traversed when the scrollbar is moved; it can be changed to a larger granularity for large documents.

Table 12.1 Scrollbar attributes

Name	Data Type	Tag or Script	Attribute Type	Default	Description
`axis`	`string`	Tag	Final	`y`	Axis may be x or y.
`disabledbgcolor`	`number`	Both	Setter	`null`	If defined, is used as the scrollbar's background color when it is disabled.
`scrollable`	`boolean`	Both	Read-only	`true`	True if the `scrolltarget` is bigger than the containing view, making the scrollbar active; the scrollbar appears disabled when scrollable is false.

Table 12.1 Scrollbar attributes *(continued)*

Name	Data Type	Tag or Script	Attribute Type	Default	Description
scrollattr	string	Both	Final		The attribute of the scrolltarget that is modified by the scrollbar.
scrollmax	number	Tag	Final		Height or width of the scrolltarget.
scrolltarget	expression	Both	Final		The view that is controlled by the scrollbar.
stepsize	number	Both	Setter	10	The distance the scrolltarget moves when the scrollbar is moved or when the step method is called.

We'll next look at the more general case of adding a scrollbar to a window containing dynamic content.

Adding a scrollbar to dynamic content

When a display contains dynamic content, it's necessary to set up a combination of a clipped parent view working in conjunction with a scrolled child view. This technique works for both windows and views to support the dynamic behavior produced by replicated objects. A scrolling display consists of two views in a parent-child relationship: the parent view, with its clip attribute set to true, clips a larger child view. Since the parent view is clipping another view, its dimension parameters must be explicitly specified.

Listing 12.3 shows a vertical scrollbar. The child view's y and height attributes are controlled by the scrollbar, so the child view only needs to specify its width. With a horizontal scrollbar, the reverse is true; the child view must specify its height and can't specify its width.

Listing 12.3 Applying a scrollbar to a view

```
<canvas>
   <dataset name="dsAddress">
      <line>Four score and seven years</line>
      <line>ago our fathers brought </line>
      <line>forth on this continent, a new</line>
      <line>nation conceived in Liberty,</line>
      <line>and dedicated to the</line>
      <line>proposition that all men are</line>
      <line>created equal.</line>
   </dataset>
   <view width="200" height="100" clip="true">
      <view width="100%">
         <text width="100%"
            datapath="dsAddress:/line/text()"/>
         <simplelayout/>
      </view>
      <scrollbar/>
   </view>
</canvas>
```

Clips child view

Follows clipping parent

Replicates text object

Executing this example produces the scrolled view in figure 12.2. If you encounter any problems with a broken scrollbar, examine the key attributes in table 12.1. They are powerful tools for diagnosing scroll-related problems.

Once we add a scrollbar to a window, we naturally want to use the mouse's scroll wheel to control it.

Adding a scroll wheel

We can easily add scroll-wheel control by setting the clipped parent view to be focusable and having the scrollbar's `focusview` attribute point back to this parent. The clipped parent view must be focusable and have focus to make the mouse wheel operational. The scrollbar's `focusview` attribute must be set to this clipped parent, since it has focus. Once focus is established in the display area, it must be set to the clipped `content` parent node.

Listing 12.4 updates the previous example with these focus settings. If you have problems making your scroll wheel operational, it's probably because window focus hasn't been correctly established. Remember that some browsers (Firefox and Safari) require that the browser's display area first be clicked to transfer focus from the browser to its display area.

Listing 12.4 Adding a scroll wheel to a window

```
<canvas>
   ...
   <window name="content" width="200"
           height="100" clip="true"
           focusable="true">
      <view width="100%">
         <text width="100%" datapath="dsAddress:/line/text()"/>
         <simplelayout/>
      </view>
      <scrollbar focusview="$once{parent}"/>
   </window>
```

Clips child view

Refers to clipping parent

```
   <handler name="oninit">
      LzFocus.setFocus(content);
   </handler>
</canvas>
```

Sets focus to content window

There is no limit to the number of scrollbars that can be controlled through the scroll wheel. It's only necessary to follow a similar procedure for each scrolling window.

Supporting a scroll wheel in the Laszlo Market

The `productwin` class can easily be updated to support a scroll wheel. Listing 12.5 shows the updated class definition.

Listing 12.5 Adding scrolling to the Laszlo Market

```
<class name="productwin">
   <simplelayout axis="y"/>
   <titleheader width="100%"/>
   <view name="container" width="100%" height="${parent.height-30}"
         clip="true" focusable="true">
      <view width="100%">
         <productrow width="100%"
                     datapath="dsProducts:/products/product"/>
         <simplelayout axis="y" spacing="2"/>
      </view>
      <scrollbar focusview="$once{parent}"/>
   </view>
   <handler name="oninit">
      LzFocus.setFocus(productlist.container);
   </handler>
</class>
```

❶ Makes clipping parent view focusable

Points back to parent view

Sets focus

We've added scroll-wheel support ❶ by adding the `clip` and `focusable` attributes to the parent view. The next piece of functionality required by our window is selection support. The window must support both selection and drag-and-drop operations, so it must handle both `onselect` and mouse-related events.

12.2.5 *Creating a selection manager*

A selection manager provides a way to manipulate a list of choices. It can add to, modify, or clear the selections in the list. Additionally, the selection manager supports the Ctrl and Shift modifiers to perform multiple and range selections. Table 12.2 lists the `selectionmanager` attributes to access selections in a list.

Table 12.2 `LzSelectionManager` attributes

Name	Data Type	Attribute	Description
`sel`	`string`	Settable	The name of the method on an object to call when the object's select state changes; the method is called with a single boolean argument. The default value for this field is `setSelected`.
`selected`	`array`	Read-only	An array representing the current selection.
`toggle`	`boolean`	Settable	If true, a reselected element loses the selection.

Table 12.3 contains the `LzSelectionManager` methods to change the selected state for standard, multiple, and range selections.

Table 12.3 `LzSelectionManager` methods

Name	Description
`clearSelection()`	Unselects any selected objects
`getSelection()`	Returns an array representing the current selections
`isMultiSelect()`	Determines whether an additional selection should be multiselected or should replace the existing selection
`isRangeSelect()`	Determines whether an additional selection should be range-selected or should replace the existing selection
`isSelected(o)`	Tests select state of input
`select(o)`	Called with a new member to be selected
`unselect(o)`	Deselects the given object

Listing 12.6 contains a list of text objects available to be selected. This example demonstrates a cross section of the `selectionmanager` attributes and methods that include choosing single, multiple, or range selections and then clearing those selections.

Listing 12.6 Creating a selection manager for the Laszlo Market

```
<canvas>
   <include href="myData.lzx"/>

   <view name ="fruitlist"
         datapath="myData:/fruit"
         bgcolor="0xDDDDDD">
```

```
      <selectionmanager name="selector"
                        toggle="true">
         <handler name="oninit">
            this.sel = "our_selector";
         </handler>
      </selectionmanager>
      <text name="txt" datapath="*/name()"
         onclick=
            "parent.selector.select(this)">
         <method name="our_selector"
                 args="selected">
            Debug.write("isMultiSelect: " +
                           parent.selector.isMultiSelect());
            Debug.write("isRangeSelect: " +
                           parent.selector.isRangeSelect());
            Debug.write("isSelected: " +
                           parent.selector.isSelected(this));
            Debug.write("=================");
            if (selected) {
               var txtColor = 0xFFFFFF;
               var bgcolor = 0x999999;}
            else {
               var txtColor = 0x000000;
               var bgcolor = 0xDDDDDD;}
            this.setBGColor(bgcolor);
            this.setAttribute('fgcolor', txtColor);
         </method>
      </text>
      <simplelayout axis="y"/>
   </view>
   <view>
      <button onclick="fruitlist.selector.clearSelection()"
         text="Clear all selections"/>
      <view>
         <edittext width="20">
            <handler name="onblur">
               parent.clear.setAttribute("offset", this.getText());
            </handler>
         </edittext>
         <button x="30" name="clear"
                 text="Clear a selection">
            <attribute name="offset" type="number"/>
            <handler name="onclick">
fruitlist.selector.unselect(fruitlist.txt.clones[this.offset]);
            </handler>
         </button>
      </view>
      <simplelayout inset="10" axis="y" spacing="3"/>
   </view>
   <simplelayout axis="x" spacing="3"/>
</canvas>
```

❶ Declares selection manager

❷ Invokes selection method

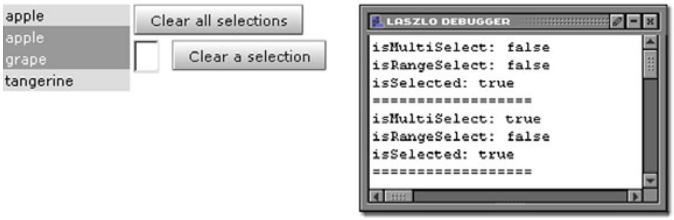
Contains selection method

Sets selection background color

Clears all selections

Clears single selection

The `toggle` attribute ❶ allows selections to be toggled on and off with successive mouse clicks. The `sel` attribute changes the name of the default `select` method from `setSelected` to `our_selector`. The text object's `onclick` event is used ❷ to call the `select` method with an argument to allow selections to be toggled. This argument can only be used to support toggling when the `toggle` attribute is set. Figure 12.3 shows the results of these selections.



```
isMultiSelect: false
isRangeSelect: false
isSelected: true
==================
isMultiSelect: true
isRangeSelect: false
isSelected: true
==================
```

Figure 12.3 The initial selection results in the `isSelected` method returning true. Holding down the Ctrl key on the subsequent selection adds to this initial selection to create multiple selections.

Now that we have some familiarity with selections, we are ready to use them in the Product List window.

Adding selections to the Product List window

To add selection capabilities to the Product List window, the `selectionmanager` is placed on the same level as a replicating set of `productrow` instances. The `selectionmanager`'s `select` method is indirectly invoked when a `productrow` receives an `onclick` event:

```
<class name="productwin">
   …
   <view name="scroll" width="100%">
      <selectionmanager name="selector"/>
      <productrow width="100%"
                  datapath="dsProducts:/products/product"
                  onclick="parent.selector.select(this)"/>
      <simplelayout axis="y" spacing="2"/>
   </view>
   …
</class>
```

Controls product rows

Replicates product row

We still need to add a `setSelected` method for this `selectionmanager` to call (note listing 12.7). When a user changes a selection, the `selectionmanager` calls this method twice: once to turn off the previous selection and once to turn on the new selection. The `selected` argument specifies whether or not the view is selected. Since the background color alternates, the background of a selection is saved in the `lastcolor` attribute.

Listing 12.7 Updating the `productrow` class to support selection

```
<class name="productrow">
   <attribute name="lastcolor" type="color"/>
   …
   <method name="setSelected" args="selected">
      if (selected) {
         productdp.setPointer(this.data);
         this.setAttribute("lastcolor", this.bgcolor);
         setAttribute("bgcolor", 0xBBBBBB);}
      else {
         productdp.setPointer(null);
         setAttribute("bgcolor", this.lastcolor); }
   </method>
</class>
```

Stores original color

Sets product pointer

"Unsets" product pointer

Since the `productdp` data pointer is still used to generate data events, no changes to the `details` class are needed to display selections in the Product Details window.

Figure 12.4 shows the final results of reimplementing our Product List window.

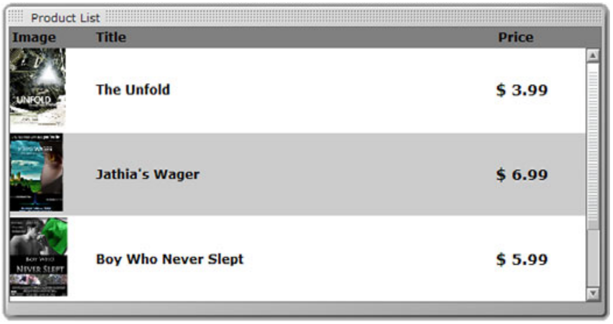


Image	Title	Price
	The Unfold	$ 3.99
	Jathia's Wager	$ 6.99
	Boy Who Never Slept	$ 5.99

Figure 12.4 Our Product List window supports sorting, scrolling, and selection for the mouse scroll wheel.

Now that we've completed building the Product List window, it is ready to be connected to the shopping cart. But first we need to add the scoreboarding features to the shopping cart.

12.3 Building the scoreboarding shopping cart

We'll use the scoreboard principle to implement our shopping cart. Because the scoreboard collects information from a number of sources, the shopping cart requires a backing `dsCart` dataset to store this information. This dataset is initially empty and is populated during execution. The shopping cart controls access to its backing dataset. This is a *totals* scoreboard, so it ensures that product items stored in the shopping cart are unique—each new product creates a new item in the shopping cart—while each previously entered product increments an existing item's quantity. Then, the shopping cart's contents are tabulated to calculate a total price.

12.3.1 Designing the Shopping Cart window

Figure 12.5 shows the wireframe used for the Shopping Cart window layout. The window has two sections. The top section contains multiple rows each with a height of 50 pixels and divided into four displayed columns. It also contains a scrollbar that is enabled when enough items are entered. The bottom section has a height of 90 pixels and consists of a 30-pixel gray square that serves as a trashcan indicator, a total price field, and a change-state button labeled "Check out."

Listing 12.8 shows the top section, represented by the shoppingcart object, with the bottom section in a view container.

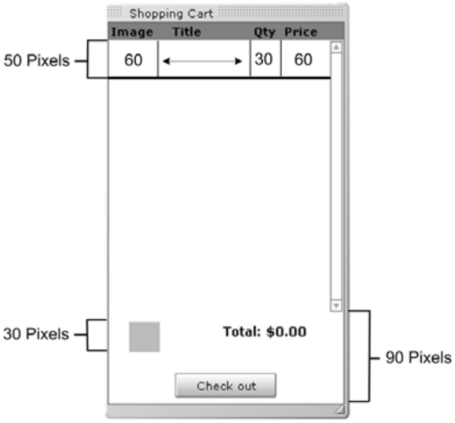


Figure 12.5 Wireframe of the Shopping Cart window

Listing 12.8 Layout for the shopping cart

```
<canvas>
   <dataset name="dsCart">
      <items/>
   </dataset>
   …
   <window name="shoppingcart" title="Shopping Cart"
           resizable="true" x="75%" width="25%" height="65%">
      <shoppingcart name="shopcart" width="100%"
                    height="${immediateparent.height-90}"
                    datapath="dsCart:/"/>
      <view width="100%" height="30" y="${immediateparent.height-80}">
         <view name="trash" width="30" height="30"
                  bgcolor="0xBBBBBB"
                  clickable="true"/>
         <text fontsize="12" fontstyle="bold" resize="true"
               text="${'Total: $' + main.shoppingcart.
                           shopcart.totals}"/>
```

Replicates purchased items

Displays trashcan icon

Displays total purchase value

```
            <simplelayout inset="20" axis="x" spacing="80"/>
         </view>
         <!--  States containing the button -->
      </window>
   </canvas>
```

The shoppingcart is similar to productwin, consisting of the shop_titleheader, shoppingcart, and shoprow classes. The columnheader class is reused for the shop_titleheader class:

```
<class name="shop_titleheader">
   <columnheader width="60" text="Image"/>
   <columnheader text="Title" options="releasetolayout"/>
   <columnheader width="30" text="Qty"/>
   <columnheader width="60" text="Price"/>
   <resizelayout axis="x"/>
</class>
```

Since the Title field has a releasetolayout option, it's stretched by the resizelayout tag to fill all remaining space within the header—the other fields have fixed widths.

In listing 12.9, shoprow is declared with a data path, thereby replicating it for each matching item in the dsCart dataset. Depending on the number of replicated objects, the display can be larger than the window height. We've added a scrollbar for displaying all items.

Listing 12.9 Replicated item rows of the shopping cart

```
<class name="shoppingcart" bgcolor="0xD5D4D3" fontstyle="bold">
   <attribute name="totals" value="0.00" type="string"/>
   <simplelayout axis="y"/>
   <shop_titleheader width="100%"/>    <—— Displays titles
   <view name="container" width="100%"
         height="${parent.height - 20}" clip="true">
      <view width="100%" name="scroll">
         <shoprow datapath="dsCart:/item"
                                   width="100%"/>   | Displays
         <simplelayout axis="y" spacing="2"/>       | purchased items
      </view>
      <scrollbar/>    <——  Scrolls
   </view>               | purchased items
</class>
```

The shoprow's data path causes the shoprow object to be replicated and also establishes a context, providing objects in the shoprow class with easier access to their data nodes. Listing 12.10 shows the layout of each row.

Listing 12.10 Layout of each row in a shopping cart

```
<class name="shoprow">
   <thumbnail y="10" maxheight="50" maxwidth="50"
              width="60" datapath="@image">
      <attribute name="imageURL" value="$path{'@image'}"/>
      <method name="applyData">
         this.setSource(gController.IMAGESFOLDER + this.imageURL);
      </method>
   </thumbnail>
   <text valign="middle" text="$path{'title/text()'}"
         fontsize="10" multiline="true"
         options="releasetolayout"/>
   <edittext name="qty" valign="middle" width="30"
             fontsize="10" datapath="qty/text()"/>
   <text valign="middle" width="60" fontsize="10"
         text="$path{'price/text()'}">
      <handler name="ontext">
         this.setText("$" + this.text);
      </handler>
   </text>
   <resizelayout axis="x"/>
</class>
```

Retrieves thumbnail image

Stretches title text

Prepends "$" to price

This provides the basic skeleton for the rows of the shopping cart. Later we'll return to this shoppingcart class to enhance it and incorporate it in the drag-and-drop network.

12.3.2 Implementing scoreboarding techniques

Now we're ready to implement the shopping cart's scoreboarding techniques. All input sources that report product requests for inclusion into the shopping cart will use the method

```
updateShopcart(data pointer to product data node)
```

which contains a data pointer argument that points to a single product data node. Input sources simply have to report new products without concern about internal shopping cart operations. The shopping cart determines whether this product represents a new item or an update to an existing item, and any actions such as totaling its contents.

Similarly, certain input sources can delete items from the shopping cart by using the method

```
deleteItem(data pointer to an item data node)
```

In the next section, we'll see an initial example of an input source using these methods to report a product request to the shopping cart.

12.3.3 Reporting add-to-cart operations

Updating the Product Details window to report product updates to the shopping cart only requires adding the `updateShopcart` method. When the user clicks the Add to Cart button, the product is added to the shopping cart. The `productdp` data pointer is the same pointer used earlier to implement the master-detail design pattern between the Product List and Product Details windows:

```
<class name="details_window" extends="window">
   …
   <button x="10" y="170" text="Add to Cart"/>
   <handler name="onclick">
      main.shoppingcart.shopcart.updateShopcart(productdp);
   </handler>
</class>
```

All the information contained in a product data node is now accessible to the shopping cart.

12.3.4 Building the shopping cart

The `updateShopcart` method performs the internal operations of adding items to the shopping cart. Since the shopping cart may already contain items, we need to determine whether an item for this product already exists. If it does, its quantity field is incremented. Otherwise, a new item data node representing the product is added to the `dsCart` dataset.

A `deleteItem` method removes an item from the `dsCart` dataset. When this method is called, the value of the quantity field is irrelevant since the item's data node is simply deleted. Listing 12.11 shows the code for these operations.

Listing 12.11 Updating an item in the shopping cart

```
<class name="shoppingcart">                                        ❶ Points to parent
   <datapointer name="dptr" xpath="dsCart:/items"/>                  items node
   <attribute name="targets" value="$once{[]}"/>
   …
   <method name="updateShopcart" args="dp">
      <![CDATA[                                            ❷ Finds SKU for
      var curr = dp.xpathQuery("@sku");                       new item
      var exist = dptr.xpathQuery("item/@sku");
      if (exist != null) {
         if (typeof exist != "object") {
            targets[0] = exist;
            exist = targets; }
         for (i = 0; i < exist.length; i++) {
            if (exist[i] == curr) {
               dptr.setXPath("dsCart:/items/item[@sku='" + curr + "']");
               var qty = dptr.getNodeAttribute("qty");
               dptr.setNodeAttribute("qty", ++qty);
               main.shoppingcart.shopcart._updateTotals();
               return; }}}
      var ele = new LzDataElement("item");
      ele.setAttr("sku", dp.getNodeAttribute("sku"));
      ele.setAttr("title", dp.getNodeAttribute("title"));
      ele.setAttr("image", dp.getNodeAttribute("image"));
      ele.setAttr("qty", 1);
      ele.setAttr("price", dp.getNodeAttribute("price"));
      dptr.p.appendChild(ele);
      main.shoppingcart.shopcart._updateTotals();
      return;
      ]]>
   </method>
   <method name="deleteItem" args="dp">
      dp.deleteNode();
      this.updateTotals();
   </method>
</class>
```

❸ Builds SKU array for existing items

❹ Checks for matching item

❺ Updates quantity and total price

❻ Creates a new item in dataset

A local data pointer ❶ is instantiated to point at the `items` parent node to establish a context. The SKU is retrieved ❷ from the product information supplied by the data pointer. Next, we collect the SKU values ❸ from all items stored in the `dsCart` dataset. The `xpathQuery` method returns a string when only a single item is contained in this dataset, and an array for multiple items. We want to simplify our code to only use arrays. So ❹ we'll check the returned value's type: if it's a string, it's converted into an array with a single element.

The items ❺ are searched for an SKU matching the new product. If a match is found, the matching item in the `dsCart` dataset is updated. If a matching SKU isn't found, it's necessary to create ❻ a new `item` data node, use the `product` data pointer to initialize its contents and set its quantity to 1, and append this new node to the `items` parent node. Finally, the shopping cart's contents are tabulated to produce a total value.

The `updateShopcart` method either updates or adds new `item` data nodes to the `dsCart` dataset. Because the shopping cart's contents are bound to the `dsCart` dataset's data nodes, the shopping cart's window reflects the contents of this dataset. This approach provides a clear separation between presentation and the underlying model; only the model is updated with the data-binding linkages automatically updating the display.

We'll next look at implementing the `updateTotals` method.

Updating the totals

Listing 12.12 shows the tabulation of the items in a `dsCart` dataset to produce a total price.

Listing 12.12 Updating the totals field

```
<class name="shoppingcart" … >
   …
   <method name="updateTotals">
      <![CDATA[
      var total = 0;
      var qty, price;
      dptr.setXPath("dsCart:/items");
      var num = dptr.getNodeCount();
      var prices = dptr.xpathQuery("item/price/text()");
      var qty    = dptr.xpathQuery("item/qty/text()");
      if (num == 1)  total = prices * qty;
      else {
         for (i = 0; i < num; i++) {
            total += (prices[i] * qty[i]); } }
      if (!isNaN(total)) {
         var str = new String(total);
         if (str.length - str.lastIndexOf(".") == 2) total += "0";
         if (str.length - str.lastIndexOf(".") == 1) total += "00";
         this.setAttribute("totals", total); }
      if (total == 0) this.setAttribute("totals", "0.00");
      ]]>
   </method>
</class>
```

❶ Sets to first item

Gets number of items in cart

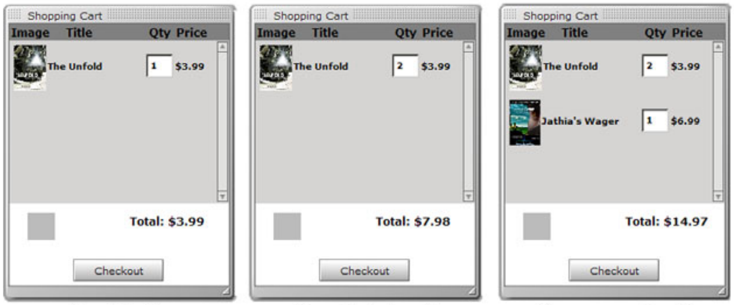
Handles number primitive case

Handles array case

❷ Formats total price

The shopping cart's local data pointer is set ❶ to point to the `items` parent node. We know the number of items, so we can multiply the unit price and the quantity for each item and add that to the total sum. The `getNodeCount` method returns a number primitive for a single item and an array object for multiple items, requiring each case to be handled separately. We ensure ❷ that the total has—down to the penny—two-digit accuracy.

Figure 12.6 shows the general operation of the shopping cart. On an initial Add to Cart button click, a new item appears in the shopping cart. Additional button



(initial) (matching) (new entry)

Figure 12.6 The first time the user clicks Add to Cart, a new item appears in the shopping cart. Subsequent clicks increment the quantity field. The user can select another product and click Add to Cart to make another product appear in the shopping cart.

clicks recognize the product and increment the quantity field. When another product is displayed and added, another item is added. After each operation the Total field is updated.

These methods can be used to support all other types of input operations.

12.3.5 Manually updating the quantity field

Adding multiple units of a product to the shopping cart is a two-step procedure. First, the product is entered into the shopping cart, and then its quantity input field is updated. Since this operation occurs within the shopping cart, it's permissible to directly access the `dsCart` dataset. Once the quantity is updated, users must press Enter or Tab to complete entering their value. The `onblur` event handler leverages the `doEnterDown` method to support the Tab key:

```
<class name="shoprow">
   …
   <edittext name="qty" valign="middle" width="30"
             onblur="this.doEnterDown()"
             doesenter="true" fontsize="10" datapath="@qty">
      <method name="doEnterDown">
         this.datapath.setNodeAttribute("qty", this.getText());
         main.shoppingcart.shopcart.updateTotals();
      </method>
   </edittext>
   …
</class>
```

Is invoked by Tab or Enter

Gets value from text field; updates total

We now have an infrastructure supporting a drag-and-drop network, as well as other input methods such as the right-click method. We now need to update the drag-and-drop network to work with a product data pointer.

12.3.6 Supporting drag-and-drop

We'll now extend the drag-and-drop network, introduced in chapter 9, to work with the shopping cart. To implement the scoreboard pattern, all drag-and-drop operations must have a payload containing a data pointer referencing the currently selected product. These product attributes will be used to decorate our draggable icon—with an image icon and title—for the dragged product. All drag-and-drop operations must still use the shopping cart's updateShopcart and deleteItem methods to conform to the scoreboard pattern.

We'll demonstrate both types of drag-and-drop operations, copy and move, in the Laszlo Market. A copy drags a product from the Product List window to the Shopping Cart window to add an item to the shopping cart. A move drags an item from the shopping cart to the trashcan—represented as a gray square—to delete it from the shopping cart. In each case, what really occurs is the addition or deletion of an item child node in the dsCart dataset.

To support these operations, we need to update the draggable class.

Creating an image icon dragger

When we initially created our drag-and-drop network, back in chapter 9, the drag source was just a gray square that handled an onmousedown event to initiate a dragging operation by calling the dragger's startdrag method. Now we have a cloned set of productrow objects, corresponding to each product row, where each row consists of a thumbnail image, with title and price text-based objects. To initiate the dragging operation, each thumbnail must handle onmousedown events to call the dragger's startdrag method. This allows each thumbnail image to initiate a drag-and-drop operation.

We have several different products, so we'll update the dragger's appearance to reflect the product being dragged. To have the dragger appear as an image icon with a title, its attributes are updated with data from the product data pointer.

The existing code for our drag-and-drop network only needs to be augmented with a datapath argument to the startdrag method to provide access to the currently selected product data node. This data path represents the data payload for the dragger:

```
<class name="productrow" fontstyle="bold" fontsize="14">
   <thumbnail name="image" datapath="@image" valign="middle"
              width="80" maxheight="75" maxwidth="75">
      …
      <handler name="onmousedown">
         targets[0] = "product_target";
         targets[1] = "media_target";
```

```
            dragger.startdrag(targets, this.datapath);
         </handler>
      </thumbnail>
      …
   </class>
```

Although the dragger's attributes could be updated with the product data through a method, a more flexible approach is to register the data path with the dragger. However, because the dragger didn't create this data path, it's passed as an argument. It doesn't yet possess the vertical communication infrastructure to handle ondata events. Instead, the dragger must obtain this ability by declaring its own data path. Although the product's data path can't be used to send events to the dragger, it can be used as an argument to set the dragger's data path using the setFromPointer method. When this method executes, it generates an ondata event that the dragger receives to point its data path to the product data fields.

Listing 12.13 shows how the default gray square icon has been enhanced into a more colorful icon containing the image and title text of the dragged product.

Listing 12.13 Enhancing the drag-and-drop network with an image icon

```
<class name="draggable" visible="false">
    …
    <attribute name="label" type="string" value=""/>
    <datapath/>                                         Instantiates a datapath
    <simplelayout spacing="2"/>
    <thumbnail name="image" maxheight="75" maxwidth="75" width="75"/>
    <text name="label" bgcolor="white" x="10" align="left"
          width="75" multiline="true"/>
    <method event="ondata">
       var selectedImage = this.datapath.xpathQuery("@image");
       var title = this.datapath.xpathQuery("@title");
       label.setAttribute("text", title);
       image.setSource(gController.IMAGESFOLDER + selectedImage);
    </method>                                           Contains image icon fields

    <method name="startdrag" args="target, data">
       <![CDATA[
       this.datapath.setFromPointer(data);              Sends ondata event
       …
       ]]>
    </method>
</class>
```

Figure 12.7 shows how the draggable icon has been updated with the title and image elements from the product data node.

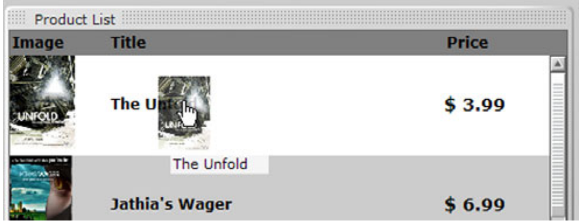


Image	Title	Price
	The U	$ 3.99
	Jathia's Wager	$ 6.99

Figure 12.7
The drag-and-drop network is provided with an informative draggable image icon.

Now that the drag portion of our application is working, let's turn our attention to integrating the shopping cart with the drop.

Performing a drag-and-drop copy operation

Dragging a product from the Product List window and dropping it into the shopping cart is a drag-and-drop copy operation, since the original product isn't altered. The previous section dealt with initiating the drag operation from the Product List window; now we'll handle the drop operation in the shopping cart.

The shopping cart must be registered with the `product_target` tracking group. In the `startdrag` method, this tracking group is activated for the shopping cart to receive `onmousetrack` events. In the `onmousetrack` handler, we'll update the shopping cart with the dragger's contents by calling `updateShopcart` with the dragger's product data pointer payload. The `shoppingcart` class is enhanced as follows:

```
<class name="shoppingcart">
   <handler name="oninit">
      LzTrack.register(this, 'product_target');
   </handler>
   <handler name="onmousetrackup">
      this.updateShopcart(dragger.datapath);
   </handler>
   …
</class>
```

Let's review the sequence of events that occur when a new product is added to the shopping cart. The dragged icon is dropped into the shopping cart, the shopping cart receives an `onmousetrackup` event, and `updateShopcart` adds a new item node in the `dsCart` dataset. This item is immediately displayed in the Shopping Cart window. Meanwhile the dragger finishes completing its `enddrag` method, which turns off the visibility of the dragger. This sequence of events happens quickly enough to appear as though the product is dragged directly into the shopping cart window.

In the next section, with only a few changes we'll implement a move operation that results in the deletion of the original view.

Performing a drag-and-drop move operation

Dragging an item from the shopping cart into the trash is a move operation since it deletes the item from the Shopping Cart window. The differences between a copy and a move operation appear only in the drop. To drag and drop items from the shopping cart to the trash requires adding a `trash_target` tracking group to the shopping cart:

```
<class name="shoppingcart">
   …
   <handler name="oninit">
      LzTrack.register(this, "product_target")
      LzTrack.register(this, "trash_target");
   </handler>
   …
</class>
```

The drag-and-drop operation is initiated by handling the `onmousedown` event for a thumbnail image in the `shoprow` class. A data path for the selected item's data node is supplied to the dragger object's `startdrag` method with the target:

```
<class name="shoprow">
   <datapath/>
   <thumbnail y="10" height="50" datapath="image">
      <attribute name="imageURL" value="$path{'text()'}"/>
      <handler name="ondata">
         this.setSource(gController.IMAGESFOLDER + this.imageURL);
      </handler>
      <handler name="onmousedown">
         dragger.startdrag("trash_target", this.datapath);
      </handler>
   </thumbnail>
   …
</class>
```

So far, this is no different from the drag-and-drop copy. The difference comes in how the drop is handled.

Receiving a drag-and-drop move

The `shoppingcart` and `trash` objects are sibling nodes within the Shopping Cart window. Since the dragger is a top-level object, it is always globally accessible. Listing 12.14 shows the steps in a drag-and-drop move.

Listing 12.14 Receiving a drag-and-drop move operation

```
<canvas>
   …
   <window name="shoppingcart" title="Shopping Cart"
           resizable="true" x="75%" height="65%" width="25%">
      <shoppingcart name="shopcart" width="100%"
                    height="${immediateparent.height-60}"
                    datapath="dsCart:/items"/>
      <view y="${immediateparent.height-80}" width="100%" height="30">
         <view name="trash" width="30" height="30"
               bgcolor="0xBBBBBB" clickable="true">
            <handler name="oninit">
               LzTrack.register(this, "trash_target");
            </handler>
            <handler name="onmousetrackout">
               this.setAttribute("bgcolor", "0xBBBBBB");
            </handler>
            <handler name="onmousetrackover">
               this.setAttribute("bgcolor", "0x000000");
            </handler>
            <handler name="onmousetrackup">
               this.setAttribute("bgcolor", "0xBBBBBB");
               main.shoppingcart.shopcart.deleteItem(dragger.datapath);
            </handler>
         </view>
         <text fontsize="12" fontstyle="bold" resize="true"
               text="${'Total: $' +
                  main.shoppingcart.shopcart.totals}"/>
         <simplelayout inset="20" axis="x" spacing="80"/>
      </view>
      …
   </window>
   …
</canvas>
```

Points to items parent node

Displays trash icon

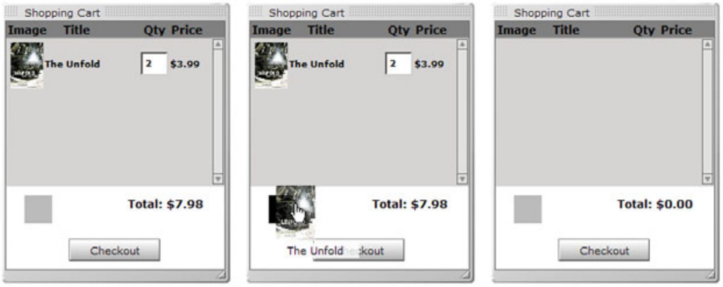
Registers trash target

Sets background color on mouseover

Displays total price

When an `onmousetrackup` event is received, the shopping cart calls its `deleteItem` method to delete this data node from the `dsCart` dataset. Once the node is deleted, the `shoprow` object that was bound to this node also vanishes from the Shopping Cart window. Immediately afterward, `enddrag` completes by turning off the dragger's visibility. As before, this sequence of events occurs quickly enough to appear as a single disposal action. Finally, the total price for the items in the shopping cart is updated. Figure 12.8 shows this sequence of events.

We'll complete our scoreboard by exploring the last input method available to a user for updating a shopping cart: the right-click method.



(initial) (deleting) (afterwards)

Figure 12.8 A drag-and-drop move operation to the trashcan deletes the item from the shopping cart.

12.3.7 Supporting the right mouse button

The right mouse button provides access to a menu where commands can be selected. This menu is context-sensitive, supplying a different selection of commands depending on its location. Support for the right mouse button is provided by the `LzContextMenu` class. Table 12.4 lists the methods available to create context menus that are callable through the right mouse button.

Table 12.4 `LzContextMenu` methods

Name	Description
`makeMenuItem(callback)`	Creates a new menu item for an `LzContextMenu`
`addItem(LzContextMenuItem)`	Adds a menu item to a menu
`hideBuiltInItems()`	Removes Laszlo-related default menu items
`clearItems()`	Removes all custom items from a menu
`getItems()`	Returns a list of custom items

An `LzContextMenuItem` is instantiated with two arguments, `caption` and `delegate`, which supply the displayed menu item label and its supporting method. Each of these arguments can be updated with the `setCaption` and `setDelegate` methods, shown in table 12.5. This menu item can be disabled by calling the `setEnabled` method with a value of false.

Listing 12.15 shows the methods for the `LzContextMenu` and `LzContextMenuItem` objects. Context menus are only available under Flash 8, so it's necessary to

Table 12.5 `LzContextMenuItem` methods

Name	Description
`setCaption(text)`	Sets the text string that is displayed for the menu item.
`setEnabled(boolean)`	If false, the menu item is grayed out and does not respond to clicks.
`setDelegate(delegate)`	Sets the delegate that is called when the menu item is selected.

set the `url` option to `lzr=swf8` to get context menus to work correctly. A context menu only appears within the view—a context—where it is defined.

Listing 12.15 Adding context menus to the Laszlo Market

```
<canvas>
   <view width="100" height="100" bgcolor="#cccccc" >
      <attribute name="item2"/>
      <method event="oninit">
         var cm = new LzContextMenu();
         var item1 = cm.makeMenuItem('my item1',
                               new LzDelegate(this, "myItem1"));
         cm.addItem(item1);
         var item2 = new LzContextMenuItem('my item2',
                               new LzDelegate(this, "myItem2"));
         this.setAttribute("item2", item2);
         item2.setSeparatorBefore(true);
         cm.addItem(item2);
         this.setContextMenu(cm);
      </method>
      <method name="myItem1">
         var item2 = this.getAttribute("item2");
         item2.setEnabled(false);
      </method>
   </view>
</canvas>
```

Creates context menu

Creates first context menu item

Creates second context menu item

Adds separator between menu items

Sets this context menu

Disables “my item2” menu item

When an `LzContextMenu` is added, it replaces any previous custom menu items with newly created menu items. Figure 12.9 shows the default menu on the left, and our context menu on the right. This was designed so that when the user chooses `my item1`, it disables the second menu item selection.

Once again, we can easily update the Laszlo Market code to accommodate input from the right mouse button.

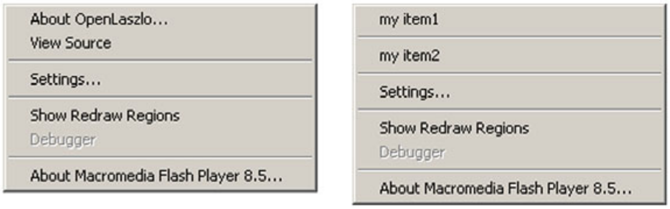


Figure 12.9 This output compares the default menu with our updated custom menu.

Updating the shopping cart with the right mouse button

To allow products to be added to the shopping cart with the right mouse button, we'll update the context menu for a particular product row with an Add to Cart item when that product is selected. Pressing the right mouse button displays an Add to Cart menu item at the top. Choosing this menu item adds the product to the shopping cart. When the product is unselected, the default menu appears and this item is no longer listed.

Listing 12.16 shows that since `LzContextMenu` is not derived from the `LzNode` tag, it can't be used declaratively. Instead we create a declarative attribute and set its value with a `once` constraint containing an instantiation of the `LzContextMenu` to ensure that only a single instantiation of this object exists. Then, an Add to Cart menu selection is added to this context menu.

Listing 12.16 Providing an Add to Cart item to the context menu

```
<class name="productrow" fontstyle="bold" fontsize="14">
   ...
   <attribute name="cm"
              value="$once{ new LzContextMenu() }"/>
   <attribute name="del"
              value="$once{ new LzDelegate(this,
                                'addtoCart') }"/>
   <method name="setSelected" args="selected">
      if (selected) {
         productdp.setPointer(this.data);
         this.setAttribute("lastcolor", this.bgcolor);
         this.setAttribute("bgcolor", 0xBBBBBB);
         var item = cm.makeMenuItem("Add to Cart", this.del);
         cm.addItem(item);
         this.title.setContextMenu(cm); }
      else {
         productdp.setPointer(null);
         setAttribute("bgcolor", this.lastcolor);
         cm.clearItems();
         this.title.setContextMenu(cm); }
   </method>
```

Instantiates LzContentMenu once

Instantiates LzDelegate

Accesses addtoCart method

Sets context menu

Clears items from menu

```
    <method name="addtoCart">
       main.shoppingcart.shopcart.updateShopcart(productdp);
    </method>
    …
 </class>
```

Updates shopping cart

This completes our scoreboard implementation. Its goal is to provide a standard interface to collect information from all major input devices. This allows products to be easily added to the shopping cart.

12.4 Summary

This chapter introduces a scoreboarding technique that provides a standard method for different input sources, ranging from the Add to Cart button to more sophisticated methods such as drag and dropping, to easily report product requests. These input sources only need to report product requests and don't have to be involved with any of the shopping cart's internal operations, such as whether a product is new or just updates the quantity of an existing product.

The Product List window was rebuilt using lower-level window components to supply the flexibility to interface to the shopping cart. We next developed the Shopping Cart window and added the scoreboarding methods used by input sources to report their requests. These methods contain the logic for determining whether new or existing products should be entered into the shopping cart's dataset. We updated each of the input sources to use this standard interface and to support the drag-and-drop and right-click operations.

In the next chapter, we'll look at how Laszlo implements animation. To do this, we'll take the current skeleton of the Laszlo Market and embellish it with animated effects, thus providing a sense of physical mass.

Part 4

Integrating DHTML and Flash

One goal of the Laszlo Market is identical operation and appearance across the Flash and DHTML platforms. Up to this point, we have been working with the core Laszlo operations. Since every platform must support this core, platform differences haven't been an issue. However, in part 4 we work with the multimedia features that differentiate the Flash and DHTML platforms. Chapter 13 covers the basics of Laszlo animation, which are identical for both platforms. Chapter 14 covers design strategies for replicating the differentiating features using alternative solutions available in each platform. Chapter 15 covers those features that can't be replicated, requiring a hybrid DHTML/Flash application. By the end of this part, we will have created a hybrid application that provides the same operating characteristics and appearance across both platforms.

13 Enhancing the user experience

This chapter covers

- Resizing panes
- Creating multistate buttons
- Animating sequences
- Animating complex groups
- Using animation effectively

> *Illusions are art, for the feeling person, and it is by art that you live, if you do.*
>
> —Elizabeth Bowen, Irish novelist

This chapter deals with ergonomic features for making users more comfortable with an application's interface. These range from splash screens for impatient users, resizable panes and buttons for a consistent look, and animation to suggest intuitive physical metaphors. In each case, the desired result is to enhance a user's experience through an intuitive interface rather than attempting to impress with flashy effects.

The chapter starts with a simple splash screen, whose function is simply to fill time until the application completes its initialization. We next move to buttons and some rules for their appearance. Buttons should have a consistent appearance to signify their membership in a group. Buttons should announce themselves by changing their appearance when the mouse cursor passes over or is clicked on them. We'll demonstrate how to create consistent resizable buttons with a multistate display that reacts to mouse cursor movement.

Next, we'll move to Laszlo's animator tags and demonstrate how animation enriches an application's presentation. We'll start with simple animation and develop some useful principles. Our objective is to use animation to endow objects with properties that correspond to physical laws. Such objects with familiar behavior are useful building blocks in creating intuitive application interfaces. For example, we'll see how temperature can be used as a metaphor to assist users with drag-and-drop operations.

13.1 Animating transitions

A splash screen is a necessary evil. Hopefully your applications will always start up fast enough that people can use them immediately. Otherwise, a splash screen is needed to say "please wait a moment." For anxious users accustomed to fast-loading web pages, a splash screen indicates that the application is not frozen, but just in the process of making itself presentable.

13.1.1 Using Laszlo's default splash screen

Laszlo's `splash` tag supplies a default splash screen. Figure 13.1 shows the Laszlo-powered progress bar that appears during startup:

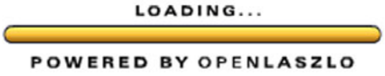


Figure 13.1 Laszlo's default splash image features a "POWERED BY OPENLASZLO" progress bar.

```
<canvas>
   <splash/>
</canvas>
```

The progress bar in the splash screen is an animated movie image. Although it doesn't accurately reflect the loading state of an application, the video does stop running when the application completes initialization.

While this works nicely during development, you'll eventually want a customized splash screen for your application.

13.1.2 Customizing a splash screen

A splash screen can be customized by specifying a view with an attached image resource. However, only a few view attributes are available because the splash screen is displayed during system initialization. An SWF or animated PNG file can be used to supply the resource for a splash screen, but animated GIF files shouldn't be used, since they aren't supported by Laszlo's SOLO mode:

```
 <canvas>
   <splash>
      <view resource="images/loading.swf"/>
   </splash>
</canvas>
```

Figure 13.2 shows a custom splash screen with a progress bar.

The `persistent` attribute can be used to retain the splash screen after the application has completed loading and initialization. This allows the splash screen to be used as a background image for the application.

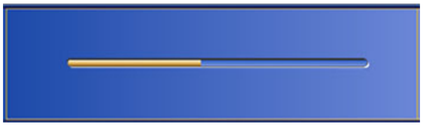


Figure 13.2 A splash screen can be customized with an attached image resource.

Once an application has completed initialization and startup, it must identify its operating controls. Traditionally, web applications have used mouse rollover events for this. Laszlo adopts this approach, which can be used with any clickable view. For clarity, we'll refer to any clickable view as a button. In the next section, we'll cover general methods for creating these buttons.

13.2 Building resizable buttons

Buttons should be easy to use and have a consistent appearance. A button should accommodate a variable-length text string, spacing itself appropriately. This should work for simple buttons as well as those that change state—`onmouseup`, `onmouseover`, and `onmousedown`—to reflect the mouse cursor's position. These buttons also need to respond to keyboard input by changing state. Finally, some buttons need to resize both horizontally and vertically. These offer the ultimate in resizability and are known as *nine-piece buttons*.

After completing this section, we'll have a comprehensive set of classes for addressing any button-related issue that could arise in application development.

13.2.1 The problem with simple buttons

OK, you have a button image and need a straightforward way to display it. The simplest approach is to create a view and just attach the image as a resource:

```
<canvas>
   <view name="poof" resource="images/button.png"/>
<canvas>
```

This works fine, as we see in figure 13.3, but this simple approach has a number of limitations.

The problem is that it tightly couples the physical dimensions of the image resource to the view declaration. During application design, physical dimensions inevitably need tweaking. The only way to change the physical dimensions of the image resource is to set its `stretch` attribute:

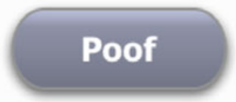


Figure 13.3 We start with a designer's rendition of the Poof button, attached to a view as a resource.

```
<canvas>
   <view name="poof" stretches="width" resource="images/button.png"/>
<canvas>
```

Figure 13.4 shows that, because the image's aspect ratio isn't maintained, the button distorts.

This simple approach requires going back to the designer to modify the original artwork every time the button size or text is changed. Clearly, this approach is not flexible enough for the real world of numerous design changes. Furthermore, buttons generally appear as a set; so it's important to maintain a consistent appearance within the set. What is needed is a mechanism to decouple the physical dimensions of the button from the artwork, to allow the artwork to be dynamically resized.

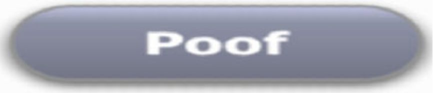


Figure 13.4 A distorted image results from directly attaching a resource to a view and resizing.

13.2.2 Building resizable buttons

The first step in building a general-purpose button is to decouple the text from the image. The ideal button automatically stretches itself to accommodate its text label.

To achieve this, we separate the button into three parts: left, middle, and right. Figure 13.5 shows the three image slices created by a designer.

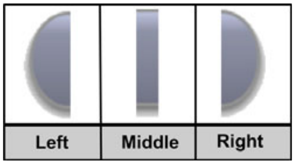


Figure 13.5 A stretchable button has three separate parts: the left and right ends and a single-pixel middle section.

A three-part button contains two fixed-size end pieces and a stretchable middle piece. This makes it ideal for configuration by a `stableborderlayout` tag. Although the middle image is only one pixel wide, its `stretches` attribute is set to `width` so it stretches without distortion. Since we don't want the `text` tag to stretch, its `options` attribute is set to `ignorelayout`. Button labels are generally horizontally and vertically centered, so the `align` and `valign` attributes are set to `center` and `middle`. Finally, we specify a default style, size, and color for the font. Of course, you can substitute your own choices. Putting these pieces together produces this example:

```
<canvas>
   <class name="sizetext" extends="text">
      <attribute name="height"
         value="${this.fontsize * 1.5}"/>
   </class>
   <class name="unibutton" >
      <attribute name="text" type="string"/>
      <stableborderlayout axis="x" />
      <view resource="images/button/btn_left.png"/>
      <view resource="images/button/btn_center.png"
            stretches="width"/>
      <view resource="images/button/btn_right.png"/>
      <sizetext options="ignorelayout" align="center"
                valign="middle" text="${classroot.text}"
                resize="true" font="Tahoma" fontsize="14"
                fgcolor="0xFFFFFF" fontstyle="bold"/>
   </class>
   <simplelayout axis="x" spacing="25"/>
   <unibutton width="100" text="Checkout"/>
   <unibutton width="200" text="Return to Store"/>
</canvas>
```

❶ Prevents text clipping

Setting the height of the text object to one-and-a-half times the font size ❶ ensures that the object's height is sized correctly for this font. The result is a consistent set of buttons, as shown in figure 13.6.

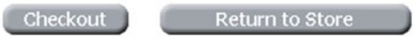


Figure 13.6 A stretchable button is created from three parts. This allows a single button class to be used with a variable-length label to produce buttons that have a consistent appearance.

This three-part technique is a template for producing buttons or panes that correctly resize for variable-length text messages. The approach can be extended to produce multistate buttons.

13.2.3 Building multistate buttons

Figure 13.7 A gradient is used to produce the illusion of a shine in a button.

Now that we have a resizable button, the next step is to add multiple visual states to indicate the mouse cursor state. These buttons support three mouse states—`mouseout`, `mouseover`, and `mousedown`—each with a different appearance. The `mouseout` state indicates that the mouse cursor is positioned outside the button; `mouseover` announces that the mouse cursor is over the button; and `mousedown` indicates that the mouse button has been clicked. Because it's popular to make buttons shiny (and who doesn't love a shiny button), a typical multistate button uses a *gradient* image to produce the illusion of a shine. In graphics, a gradient refers to a gradual transformation from one color to another. An example gradient is shown in figure 13.7.

Designers use a gradient along with shadow effects to differentiate among mouse states. Generally, a lighter button gradient is used in the `mouseover` state with a darker gradient for `mousedown`. Although the differences are often subtle, they are easily seen in a complete button image.

Each slice of the three-part button is enhanced to contain a *multiframe resource* consisting of three frames whose order is set to reflect the current state of the mouse cursor. See figure 13.8.

A button's mouse states are given priority in the order of `mouseup`, `mouseover`, and `mousedown`. The `mouseover` state has priority over `mousedown`, because when the mouse button is pressed, its operation can be canceled by dragging the mouse outside the button. These priorities require that the resource frames be specified in the following order:

- `mousedown`
- `mouseup`
- `mouseover`

In order for the mouse rollover states to operate correctly, the mouse-state images (frames) in a resource declaration must appear in this order:

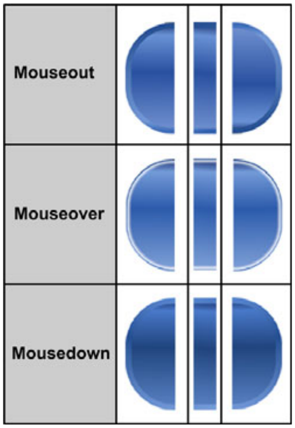


Figure 13.8 Normal, lighter, and darker images, from top to bottom, indicate the state of the button.

```
<library>
   <resource name="basic_button_left">
      <frame src="images/basic_button/btn_left_down.png"/>
      <frame src="images/basic_button/btn_left_up.png"/>
      <frame src="images/basic_button/btn_left_over.png"/>
   </resource>
   <resource name="basic_button_center">
      <frame src="images/basic_button/btn_center_down.png"/>
      <frame src="images/basic_button/btn_center_up.png"/>
      <frame src="images/basic_button/btn_center_over.png"/>
   </resource>
   <resource name="basic_button_right">
      <frame src="images/basic_button/btn_right_down.png"/>
      <frame src="images/basic_button/btn_right_up.png"/>
      <frame src="images/basic_button/btn_right_over.png"/>
   </resource>
</library>
```

When the frames appear in this order, their relationship can be given in the expression contained in the fnum attribute, where frames have a one-based offset:

```
<class name="basic_button" focusable="true">
   <attribute name="fnum"
           value="${ min? ( mdown ? 3 : 2 ) : 1 }"/>
   <attribute name="min" value="false"/>
   <attribute name="mdown" value="false"/>

   <method event="onmouseover">
      this.setAttribute("min", true );
   </method>
   <method event="onmouseout">
      this.setAttribute("min", false );
   </method>
   <method event="onmousedown">
      this.setAttribute("mdown", true );
   </method>
   <method event="onmouseup">
      this.setAttribute("mdown", false );
   </method>
   …
</class>
```

The fnum attribute contains a two-part equation whose meaning is shown in table 13.1. The fnum attribute uses a tertiary expression to determine the returned value. Although this expression might appear somewhat complex, it's an elegant way to set a constraint from a selection of values. In layman's terms, this expression states that when the mouse is over the button, it is necessary to further determine whether the mouse button is down or up. When the mouse isn't over the button, the frame containing the image corresponding to the mouseup state is displayed. If the first condition is not met, the second condition is irrelevant, which is shown in table 13.1 with dashes.

Table 13.1 Mouse state versus frame displayed

Mouse State	min	mdown	Frame Displayed
onmouseover	true	–	Need to check value of mdown
onmouseout	false	–	mouseup
onmousedown	–	true	mousedown
onmouseup	–	false	mouseup

The displayed frame image is controlled through the fnum attribute:

```
<view frame="${classroot.fnum}" resource="basic_button_left"/>
<view frame="${classroot.fnum}" stretches="width"
      resource="basic_button_center"/>
<view frame="${classroot.fnum}" resource="basic_button_right"/>
```

Although this appears complicated, the good news is that, since we are creating a button template, it only needs to be set up once in a class definition. Afterward, you can create an endless series of new button instances from this template, each with a consistent appearance and operation.

Handling focus and default

When a button has focus, it receives by default all keystroke events—onkeydown and onkeyup. A button is only concerned with the Enter key (32 ASCII) or the spacebar (13 ASCII). When either key is pressed, an event is sent to the onmousedown event handler, causing display of the mousedown frame image. When the key is released, the onmouseup event handler sends an event to display the mouseup frame. Listing 13.1 shows how to integrate these actions with a button.

Listing 13.1 Integrating the mouse states with focus and activation-by-keystroke

```
</library>
   <class name="basic_button" focusable="true">
      <attribute name="_fnum"
         value="${ min? ( mdown ? 3 : 2 ) : 1 }"/>
      <attribute name="min" value="false"/>
      <attribute name="mdown" value="false"/>
      <attribute name="text" type="string"/>
      <handler name="onmouseover">
         this.setAttribute("min", true );
      </handler>
      <handler name="onmouseout">
         this.setAttribute("min", false );
      </handler>
      <handler name="onmousedown">
         this.setAttribute("mdown", true );
      </handler>
      <handler name="onmouseup">
         this.setAttribute("mdown", false );
      </handler>
      <handler name="onkeydown" args="k">
         if (k == 13 || k == 32) {
            this.onmousedown.sendEvent();
      </handler>
      <handler name="onkeyup" args="k">
         if (k == 13 || k == 32){
            this.onmouseup.sendEvent();
      </handler>
      <view frame="${classroot.fnum}"
            resource="basic_button_left"/>
      <view frame="${classroot.fnum}" stretches="width"
            resource="basic_button_center"/>
      <view frame="${classroot.fnum}"
            resource="basic_button_right"/>
      <stableborderlayout axis="x"/>
   </class>
</library>

<canvas>
   <include href="resources.lzx"/>
   <include href="basic_button.lzx"/>

   <simplelayout axis="x" spacing="3"/>
   <basic_button width="100" text="Checkout"/>
   <basic_button width="200" text="Return to Store"/>
</canvas>
```

❶ **Simulates mousedown with keystrokes**

❷ **Simulates mouseup with keystrokes**

The onkeydown handler ❶ is triggered when the user presses any key. It sends an event to trigger the onmousedown handler to display the mousedown visual state. The onkeyup handler ❷ is triggered when the user releases any key. It sends an event to display the mouseup visual state.

13.2.4 Building resizable nine-piece panes

Although the width available for the label of a three-piece button is variable, the font size is still constrained by the height. A nine-piece pane can stretch both width and height to accommodate any font size. However, for buttons it's generally easier to just maintain different-sized button classes—for example, `small_button`, `medium_button`, and `large_button`. Otherwise, there would be 27 mouse-state images to manage.

Nine-piece panes are generally used only for views containing variable-sized images. The objective is to allow a window's dimensions to change during development without having to redo the original artwork. To accomplish this, the corners have a fixed size while the other sections stretch. Because stretching occurs along both axes, the contents of a pane must be carefully selected to minimize the effects of distortion. The best case is a pane with a border or frame, since its orientation matches the direction being stretched. Figure 13.9 shows an example.

The center section is doubly distorted, along both axes, so it must either be a solid color or be overlaid with another image. Listing 13.2 generates the image seen in figure 13.9.

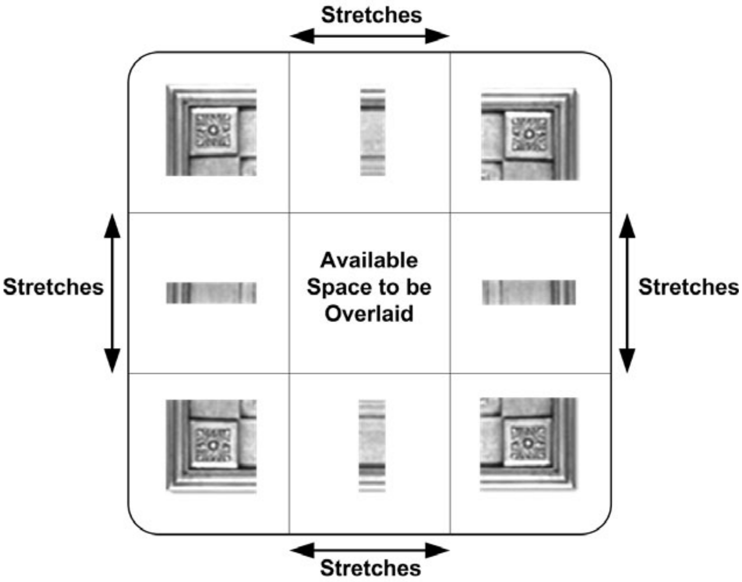


Figure 13.9 Borders and frames are good candidates for a nine-piece pane since the noncorner pieces are oriented to minimize the distortion of stretching.

Listing 13.2 Implementing a nine-piece stretchable pane

```
<canvas>
   <class name="sizetext" extends="text">
      <attribute name="height" value="${this.fontsize * 1.5}"/>
   </class>

   <class name="megabutton">
      <attribute name="text" type="string"/>
      <view height="${classroot.height}">          Creates
         <view width="${classroot.width}">     <-- first row
            <view resource="images/ninepiece/frame_01.png"/>
            <view resource="images/ninepiece/frame_02.png"
                  stretches="width"/>
            <view resource="images/ninepiece/frame_03.png"/>
            <stableborderlayout axis="x"/>
         </view>                                Creates
         <view width="${classroot.width}"     <-- second row
               stretches="height">
            <view resource="images/ninepiece/frame_04.png"/>
            <view resource="images/ninepiece/frame_05.png"
                  stretches="width"/>
            <view resource="images/ninepiece/frame_06.png"/>
            <stableborderlayout axis="x"/>
         </view>                                   Creates
         <view width="${classroot.width}">     <-- third row
            <view resource="images/ninepiece/frame_07.png"/>
            <view resource="images/ninepiece/frame_08.png"
                  stretches="width"/>
            <view resource="images/ninepiece/frame_09.png"/>
            <stableborderlayout axis="x"/>
         </view>
         <stableborderlayout axis="y"/>
      </view>
      <sizetext options="ignorelayout"
              align="center" valign="middle"
              text="${classroot.text}" resize="true"
              width="${classroot.width}" font="Tahoma"
              fontsize="24" fgcolor="0xBBBBBB"
              fontstyle="bold"/>            <-- Creates centered text
   </class>
   <megabutton width="200" height="200" fontsize="12" text="Poof"/>
   <megabutton width="250" height="250" fontsize="40" text="Poof"/>
</canvas>
```

When slicing up an image, a designer must ensure that the subimages match correctly. Corner images are square; the top and bottom pieces are one pixel wide with a height matching a corner side; the side pieces are one pixel high with a width matching a corner side; and the center piece is one pixel square. See figure 13.10.

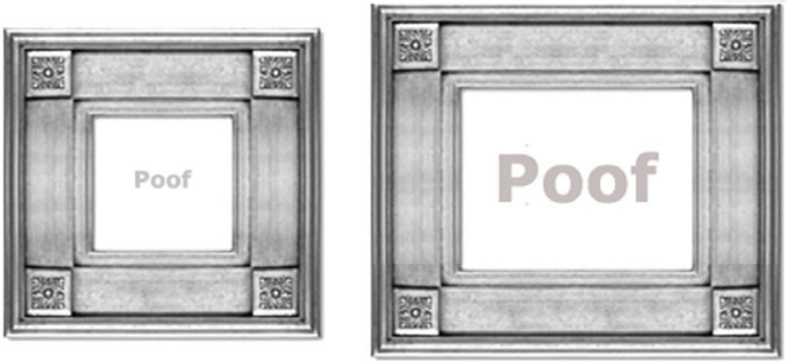


Figure 13.10 A nine-piece pane can be resized both horizontally and vertically to accommodate a center image of any size.

We've now come to the last item in our discussion about user feedback and controls: mouse-window interactions. Let's return to the modal window, which blocks activity in other windows until it is closed. The obvious application is the user login protocol.

13.3 Modal windows and button interactivity

We introduced the modal window in chapter 9 where we used it for registration in the Laszlo Market. Because unregistered users shouldn't be able to access the rest of the application, this dialog window blocks interactions with other windows until the user's input is received and the modal window closed. While a modal window is displayed, all other views should appear paused, but not locked up and frozen. Although all operations for these views are disabled, the mouse rollover states should still be operational. In other words, the application is only paralyzed—that is, it can't move—but it's not frozen—that is, its eyes are still blinking. To do this, the mouse and keyboard events are filtered by a modal manager's `passModeEvent` method to pass only a controlled set of events to nonmodal views.

The `passModeEvent` method determines the combination of events passed and views receiving these events. In listing 13.3, only the `onmouseover` and `onmouseout` events are passed to the Show Window button, and only the `onclick` event is passed to the Close Window button.

Listing 13.3 Managing a modal window

```
<canvas>
   <simplelayout axis="y" inset="10" spacing="3"/>
   <button name="show" text="Show Window">
      <method event="onclick">
         win.open();
         Debug.write("######## makeModal ##########");
      </method>
      <method event="onmouseover">
         Debug.write("Show Window : onmouseover");
      </method>
      <method event="onmouseout">
         Debug.write("Show Window : onmouseout");
      </method>
   </button>
   <button name="close" text="Close Window">
      <method event="onclick">
         Debug.write("Close Window : onclick");
         win.close();
      </method>
      <method event="onmouseover">
         Debug.write("Close Window : onmouseover");
      </method>
      <method event="onmouseout">
         Debug.write("Close Window : onmouseout");
      </method>
   </button>
   <window x="150" name="win"
           visible="false"
           width="250" closeable="true">
      <text multiline="true" width="${parent.width}">
         This window is now modal and has been placed
         on the modal stack. Only the window and its
         subviews can receive mouse and keyboard events.
         This window has a passModeEvent method that
         allows the onmouseover and onmouseout mouse
         events through to the "Show Window" button
         and the onclick event through to the "Close
         Window" button.
      </text>
      <method name="open">
         LzModeManager.makeModal(this);
         super.open();
      </method>
      <method name="close">
         LzModeManager.release(this);
         super.close();
      </method>
```

1 Shows Window button

Closes Window button

Declares closeable window

Makes window modal, opens it

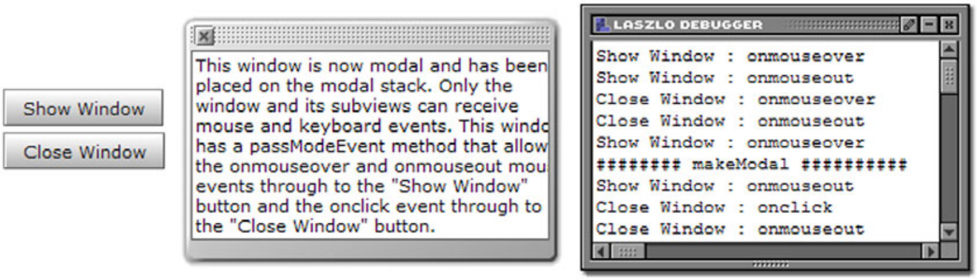
Releases modal property, closes window

```
        <method name="passModeEvent"
                args="event, view">
           if (event == "onmouseover" ||
               event == "onmouseout") {
              switch (view.name) {
                 case 'show': return true; }}
           if (event == "onclick") {
              switch (view.name) {
                 case 'close': return true; }}
        </method>
     </window>
  </canvas>
```

❷ Updates receivable mouse events

When the modal window isn't displayed, both buttons respond to mouseover and mouseout events. When the user clicks the Show Window button ❶, the ordinary win window is converted into a modal window and explicitly opened with a call to the superclass's open method. Immediately after, Laszlo calls the passModeEvent method ❷ to allow filtered events to pass through to its specified views. In particular, onmouseover and onmouseout events are passed to the Show Window button, but only onclick events can pass to the Close Window button.

Figure 13.11 shows that initially both buttons receive all mouse events. After the user clicks the Show Window button, the modal window displays. At that point, only the Show Window button receives onmouseover and onmouseout events. But the Close Window button can still receive an onclick event to close this modal window. The window's close method is invoked, which releases the window's modal properties and explicitly closes the window. Once the window has been closed, it releases modal control and once again the Close Window button responds to onmouseout events.



This window is now modal and has been placed on the modal stack. Only the window and its subviews can receive mouse and keyboard events. This windc has a passModeEvent method that allow the onmouseover and onmouseout mou events through to the "Show Window" button and the onclick event through to the "Close Window" button.

```
Show Window : onmouseover
Show Window : onmouseout
Close Window : onmouseover
Close Window : onmouseout
Show Window : onmouseover
######## makeModal ##########
Show Window : onmouseout
Close Window : onclick
Close Window : onmouseout
```

Figure 13.11 The Show Window button causes the modal window to be displayed, while the Close Window button closes it. The modal window allows the Show Window button to respond to `onmouseover` and `onmouseout` events. The Close Window button responds only to `onclick` events.

This concludes our discussion of individual resource frames. These static images are used by interface control components to display different states, signaling to users their ability to perform actions. In chapter 14, we'll show you how to customize the appearance of these components. In the following section, we move from static single frames to a sequence of frames to create animated effects.

13.4 Basics of animation

In this section, we'll describe the basics of Laszlo animation and provide a style guide on how to use animation effectively. Our goal for using animation in an effective manner is that it should have a tangible purpose. Animated sequences that don't serve a fundamental need—that only dazzle—quickly become annoying. This type of gratuitous animation is often disparaged as "eye candy." But what constitutes a tangible purpose? One answer is to simply ask, "What is the goal of this animated sequence?" In this section, we outline some of the goals that can be achieved with animation.

You have already seen animation used in the Laszlo Market in constructive ways; it's been used to support transitions between static screens and to perform drag-and-drop operations to the shopping cart and trash.

In particular, we want to use animation to achieve an emotional response from users by providing a sense of familiarity. This goal is achieved by building psychological links to the physical world. Rather than trying to exactly mimic the reality, it's better to supply subtle hints in unexpected ways. These hints frequently work as visual jokes or puns on the relationship between the physical and virtual worlds. They provide a sensation of familiarity within the uncharted waters of interface navigation to give users a higher comfort level. In short, we want to make the application fun to use. Although this is more of an art than a science, we'll show you some useful general rules.

NOTE *A recommendation on how to read this chapter*—Although we have illustrated this section with informative diagrams, the true nature of animation is impossible to capture in the static confines of a book. To get the most value from this material, you should run OpenLaszlo and observe these applications firsthand. Each example has been designed to provide a particular visual sensation. Merely reading and following the source code won't provide the visual feedback necessary to gain the full value of the concepts.

13.4.1 Selling visual illusions

For a number of psychological and physical reasons, flat animation appears lifeless and fails to emotionally engage the viewer. To squeeze a three-dimensional activity into a two-dimensional space, we must sell the visual illusion with a heightened sense of reality. Think of the theater, where actors wear makeup and act dramatic so their characters won't appear lifeless and wooden. Similarly, animation sometimes needs to be a little "over the top" in order to reach the "back rows" of your audience. Various tricks can be used to inject a sense of life into this two-dimensional medium—techniques as simple as using drop shadows in static images.

Why do people like drop shadows?

Although few websites require a sense of depth, *drop shadows*—those ubiquitous dark shadows that make images appear to float on the screen—are common. Although a design purist might decry this overuse, most people accept drop shadows as an enriching design element. Like salt or pepper, a little bit goes a long way.

People feel most comfortable working in the real world, a world with three dimensions, governed by the physical laws of nature. When confined within a two-dimensional screen, our minds search for visual clues to provide a sense of depth, because it provides a link back to physical reality. Drop shadows provide such clues.

Drop shadows are used for psychological effect, rather than for esthetics. In other words, they sell the visual illusion of depth; their overall esthetic value is less important. This provides a wide latitude for using drop shadows to create a sense of realism. Even though this might result in shadows where the sun won't ever shine, the goal justifies the means. The ultimate goal is to make users comfortable using an application. If this means using an exaggerated sense of depth, this is an acceptable trade-off.

Another link to the real world is a sense of physicality or mass, which can be suggested through animation.

Implying mass with animation

Animation occurs in a context since, by default, objects float weightlessly. Rather than settle for this default, a better choice is to supply a context governed by the physical laws of the real world. To achieve this, we must create the illusion that animated objects have mass and density. These illusionary attributes provide objects with base properties that are governed by a set of contrived physical laws. For example, we could imply a difference in mass by having larger objects fall faster than smaller objects (let's just ignore Galileo). Another contrived rule might be that dark-colored objects have greater density than light-colored objects.

It's not that the individual rules are so important in themselves but that a consistent set of rules exists. The result for a user should be a psychological link to the physical world.

For these properties to be clearly communicated, they must be exaggerated. An imagined composition of an object can establish some extremes; an object can be imagined to be composed of soft rubber that is easily deformed, or of steel that is hard to deform. Objects shouldn't send an ambiguous message by softly collapsing into one another; rather, they must send a clear message by squashing, bouncing, or slamming shut. In the following sections, we'll illustrate these principles with examples using the animator and animatorgroup tags.

13.4.2 Using animators and animatorgroups

In traditional animation, master animators draw the *keyframes* containing the most important moments of an action. Junior animators then draw frames connecting these keyframes with *between frames*. The act of interpolating between keyframes is called *tweening*. Combined with a few simple mathematical functions, it is the central concept in most timeline-based animation.

The role of the animator tag is to create the tweened steps between a pair of start and end keyframes. The transition between these keyframes is described by varying the value of an attribute over a range during a time period. The scope of this attribute can encompass coordinate values, size, colors or any other quantifiable entity. We have previously used a JavaScript version of the animation method, so a natural question is, "What are the benefits of a declarative animator?"

There are many such benefits. First, the animator object can be reused, while the animation method requires a new animator instantiation for each use. This can affect performance if a significant number of animators are used. Second, a declarative animator can more easily respond to events. And finally, declarative animators can be grouped into an animatorgroup to perform synchronized animation. Let's start with a simple example of animation.

Consider a dropping ball. The simplest approach to animate this is to describe a sequence with start and end positions. Although we can't decompose the animation into individual frames, let's suppose that keyframe 1 marks the start and keyframe 10 marks the end position. In this animated sequence, the ball falls a distance of 100 units along the x coordinate in a sequence of 10 keyframes displayed at a constant rate. We can express this in an informal notation to record the ball's position:

```
keyframe  1: ball is at position = (0, 0)      (top)
keyframe 10: ball is at position = (100, 0)    (bottom)
```

If we perform simple tweening, in frame 5 the ball would be halfway between the top and bottom positions, at position (50, 0). In this scenario, the ball moves the same amount in every frame, until it hits the bottom. This motion can be described with an `animator` tag like this:

```
<canvas>
   <view resource="images/earth.png" x="70" y="10"
         width="150" height="150" stretches="both">
      <animator id="simpleAnim" start="false" target="${parent}"
                from="30" to="${canvas.height-200}"
                duration="1000" attribute="y"/>
   </view>
   <view name="controls" y="${canvas.height-30}">
      <button onclick="simpleAnim.doStart()" text="Start" />
   </view>
</canvas>
```

The `doStart` method starts the animated sequence. Figure 13.12 shows this simple linear motion of a ball, shown as the earth, in three positions: start, middle, and end.

Although the ball does indeed drop from the top to the bottom, the end result fails to engage us. We see the ball move within the specified duration, but it doesn't give us a visceral sensation of falling. If you've ever suffered a large fall or taken a physics course, you perceive at a subconscious level that this isn't how falling works. When you start to fall, you begin slowly and continually gather speed until you reach the bottom. While every falls begins the same way, certain factors determine how it ends. These factors provide the nuances that help sell the illusion that an object is falling. Let's start with the beginning of a fall.

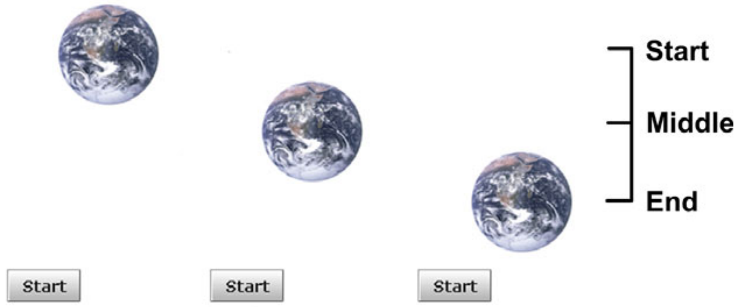


Figure 13.12 The linear motion of a ball moving through space doesn't provide a convincing depiction of falling.

Easing motion

Traditional animation uses the notion of *ease* to deal with the endpoints of motion. *Easing in* means slowly accelerating until a top speed is reached, while *easing out* means decelerating to a stop. An animated sequence can ease in, ease out, ease both, or be linear. An important byproduct of easing is the illusion of mass.

To glide an object to a halt, the animator's `motion` attribute is set to either `easeout` or `easeboth`. This makes the element slowly accelerate at its start and later decelerate to a stop:

```
<canvas>
   <view resource="images/earth.png" x="100" y="0"
         width="150" height="150" stretches="both">
      <animator id="simpleAnim" start="false"
           target="${parent}"
           motion="easeboth" from="30"
           to="${canvas.height-200}"
           duration="1000" attribute="y"/>
   </view>
   <view name="controls" y="${canvas.height-50}">
      <button onclick="simpleAnim.doStart()" text="Start"/>
   </view>
</canvas>
```

Although easing provides an illusion of mass, the results still aren't very convincing. The biggest problem is that a single animator doesn't allow us to manipulate the ease time but only the duration of the sequence. This provides the appearance that all objects have a uniform mass. Normally, larger objects are expected to have a greater inertia and require a longer ease period. To reach the next level of complexity with animation, we need to advance to the `animatorgroup` tag.

The next effects needed for realistic animation are *squash* and *bounce*. A fall can end several ways: it can glide to a halt, it can destroy itself, or it can squash and bounce. Since objects that degrade—crumble to pieces—aren't reusable, we'll concentrate on the squash-and-bounce scenario.

13.5 Complex animated effects

An *animator group* specifies and controls a group of animators. Animator groups are used to build complex movements by combining several animations, either sequentially or concurrently. An animator group uses declarative properties to maintain a hierarchical tree of nested animator groups and animators. Although this might seem to provide limitless animation complexity, animators do consume a significant amount of resources and when used excessively produce degraded "chunky" animation. Table 13.2 lists the attributes for the `animator` and `animatorgroup` tags.

Table 13.2 `animator` and `animatorgroup` attributes

Name	Data Type	Tag or Script	Attribute Type	Description
`attribute`	`object`	Both	Setter	The name of the attribute whose value is animated.
`duration`	`string`	Both	Setter	The duration of the animation in milliseconds.
`from`	`number`	Both	Setter	The start value for the animation.
`motion`	`string`	Both	Setter	Valid values are `linear`, `easein`, `easeout`, `easeboth`, and `any`.
`paused`	`boolean`	Both	Setter	Pauses an executing animator.
`process`	`string`	Both	Setter	Valid values are `simultaneous` and `sequential`.
`relative`	`boolean`	Both	Setter	If true, the value is relative to the initial value of the attribute; if false, it is absolute.
`repeat`	`number`	Both	Setter	The number of times to repeat the animation.
`start`	`boolean`	Both	Setter	Requires the animator to call `start`.
`target`	`object`	Both	Setter	The object to animate.
`to`	`number`	Both	Setter	The final value for the targeted attribute.

With an animator group, the ease periods can be individually specified and combined sequentially. One problem with ease is that its effects are often too subtle and can be easily overlooked. To effectively imply mass, the effect needs to be exaggerated.

13.5.1 *Simulating a squashed ball*

A better way to enhance the illusion of mass is to provide an object with a squashed appearance upon colliding with another object or boundary. Listing 13.4 illustrates this illusion of mass through squashing. It features a top-level animator group `anim` that contains two sequential animators. The first animates the descent of the ball. When it completes, a seamless transition to the next animation sequence produces a squashed appearance for the ball. Since we don't want an abrupt transition from one state to the next, the `motion` attribute is used to help smooth this transition. The squashing motion is produced by the compound effect of animating across four attributes: `width`, `height`, `x`, and `y`.

Listing 13.4 Creating an illusion of mass by squashing

```
<canvas>
   <view name="ball" resource="resources/earth.png"
         x="100" y="0" width="150" height="150" stretches="both"/>
   <animatorgroup name="anim" target="ball"
                  start="false" process="sequential">
      <animator attribute="y" to="${canvas.height-150}"
                duration="900" motion="easein"/>
      <animatorgroup name="squash" duration="100"
                     motion="easein" process="simultaneous">
         <animator attribute="width"
                   to="180"/>
         <animator attribute="height"
                   to="120" motion="easeout"/>
         <animator attribute="x"                       ❶ Sets attributes
                   to="-15" relative="true"/>             for squashing
         <animator attribute="y"
                   to="30" relative="true"
                   motion="easeout"/>
      </animatorgroup>
      <animatorgroup name="unsquash" duration="100"
                     motion="easein" process="simultaneous">
         <animator attribute="width"
                   to="150"/>
         <animator attribute="height"
                   to="150" motion="easein"/>
         <animator attribute="x"                       ❷ Sets attributes for
                   to="15" relative="true"/>              unsquashing
         <animator attribute="y"
                   to="-30" relative="true"
                   motion="easein"/>
      </animatorgroup>
   </animatorgroup>
   <view x="10" name="controls" y="${canvas.height-40}">
      <button onclick="anim.doStart()">Start</button>
   </view>
</canvas>
```

To achieve this squashed appearance, the ball's width is increased and its height is decreased ❶. Since the squashed ball is wider, it expands along the x-axis, thus requiring the x coordinate to be adjusted to the right to keep the center of the ball on its y-axis. The `relative` attribute is set to true because the `x` and `y` attributes are relative to the animation and not to the canvas.

After the ball is squashed, it must be unsquashed ❷ to spring back to its original shape. To unsquash the ball, we add the `unsquash` animator group to the animator sequence.

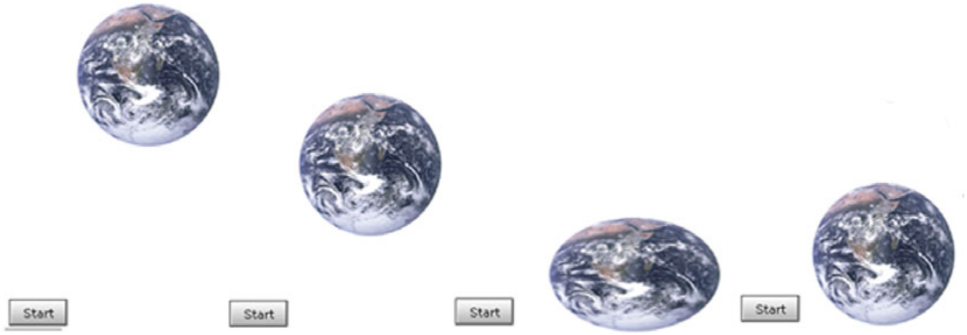


Figure 13.13 To provide the illusion that an element has mass, it becomes squashed when it reaches its termination point.

It's a worthwhile exercise to change the size of the ball and experiment with other settings in the `squash` animator group to find values that produce the most realistic effect. The final result, shown as a time sequence in figure 13.13, is quite convincing and produces the illusion that the ball has a composition type.

Generally, there are only two composition types: elastic or inelastic. So we are always dealing with either rubber or steel balls. The most important thing is to maintain consistency within a composition type. An object should squash and bounce against all surfaces consistently.

13.5.2 Interactive animation

So far we have been working with what is known as *prescripted animation*, whereby an animation runs from start to completion without interruption. An important Laszlo goal is to design applications that are responsive to a user's actions. The ideal Laszlo user interface is always alive, reactive, and fluid. If we don't want the ball to hit the floor, there should be a way to stop it.

The obvious question is, "What happens when a user interrupts an animation sequence?" The simplest solution is to just pause and freeze the motion. However, this gives the impression that the system has locked up. In many cases, the best response is to animate back to the last starting point or to a default home position. This provides users with the sensation that they have interrupted an animation sequence, but the system is still operating normally.

One situation where it is preferable to pause an animation sequence is when the animation contains state information. For example, an animator may contain attributes with dynamically set values, such as trajectory or speed, in which case it must be paused to maintain the current state.

Pausing and restarting animation

The example in listing 13.5 demonstrates how to pause and restart an animated sequence. The `repeat` attribute is used to run the animation sequence in a repeated loop. Clicking a Pause/Unpause button causes the ball to pause and to restart on the subsequent click.

Listing 13.5 Pausing and restarting an animation sequence

```
<canvas>
   <view resource="resources/earth.gif" id="ball"
         x="100" y="0" width="150" height="150" stretches="both"/>
   <animatorgroup id="anim" target="ball" start="true"
           paused="false" process="sequential"
           repeat="Infinity">
      <animator attribute="y" to="${canvas.height-200}"
                duration="900" motion="easein"/>
      <animatorgroup name="bouncer" duration="1000"
                     process="simultaneous">
         <animator attribute="y" to="0" motion="easeout"/>
      </animatorgroup>
   </animatorgroup>
   <view name="controls" y="${canvas.height-40}">
      <button onclick="anim.pause()">Pause/Unpause</button>
   </view>
</canvas>
```

As you can see in figure 13.14, the earth image is paused in its downward path. When it restarts, it continues on this path.

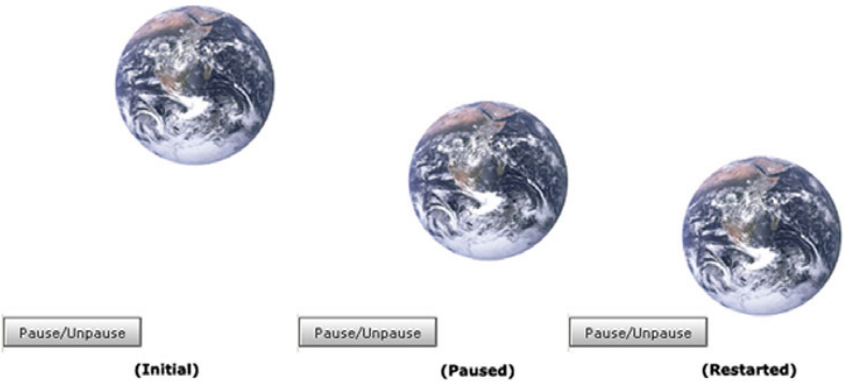


Figure 13.14 When an animation is paused, it stops in its current position.

Resetting an animation state

As we suggested earlier, a good way to handle a pause is to move the object back to a home position. This is a reset operation rather than a pause. To implement a reset, we can establish two states: *loop* and *reset*. Loop is the default state, displaying a continuously bouncing ball.

An isloop attribute controls the application of the reset state. When the Pause/Unpause button is clicked, the value of isloop is flipped, causing the reset state to execute an animator that resets the ball back to its starting position at the top of the screen. Listing 13.6 shows how to define loop and reset states.

Listing 13.6 Implementing a reset operation with loop and reset states

```
<canvas>
   <view resource="resources/earth.gif"
         id="ball" x="100" y="0" width="150"
         height="150" stretches="both"/>
   <state name="loop" apply="true">
      <animatorgroup id="anim" target="ball"
           start="true" process="sequential"
           repeat="Infinity">
         <animator attribute="y"
            to="${canvas.height-200}"
            duration="900" motion="easein"/>
         <animator attribute="y" to="0"
            duration="1000"
            motion="easeout"/>
      </animatorgroup>
   </state>
   <state name="reset">
      <animator attribute="y" to="0"
                duration="1000"
                motion="easeout"/>
   </state>
   <view name="controls"
         y="${canvas.height-40}">
      <button text="Pause/Unpause">
         <attribute name="isLoop"
                    value="true"/>
         <handler name="onclick">
            if (this.isLoop) {
               loop.remove();
               reset.apply(); }
            else { loop.apply();
                   reset.remove(); }
            this.isLoop = !this.isLoop;
         </handler>
      </button>
   </view>
</canvas>
```

Displays earth ball

Produces bouncing motion

Resets ball to initial position

Controls ball playing state

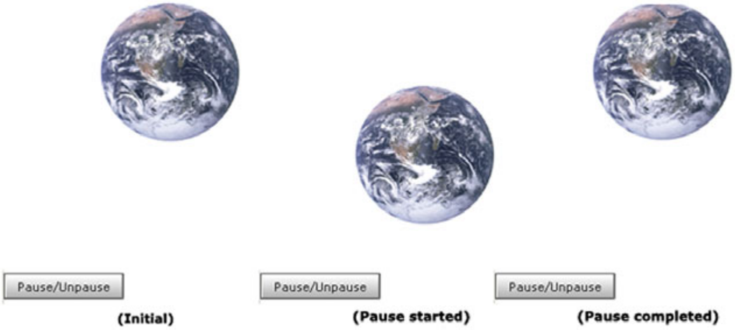


Figure 13.15 A reset operation moves an object back to a home position.

Reset is useful for restarting a feature without having to restart the entire application. It's also useful during development. For example, if an application is a game requiring some type of coordinated input, it's useful to be able to get back to a previous state. Figure 13.15 illustrates a reset for the falling ball.

While ease suggests that an object has mass by dealing with its inertia, an even more realistic technique is to imply inertia though a time delay.

13.5.3 Using delay for expressive purposes

Delay is used in many disciplines, such as music, to produce a warmer ambience. Suppose that a group of panes needs to be displayed in an application. The simplest way to display them is to just pop them into place. This is the way most user interfaces work. But it also results in a one-dimensional user experience. Listing 13.7 shows how this popping action could be produced.

Listing 13.7 Popping panes into place

```
<canvas bgcolor="0xAFAFAF">
   <simplelayout axis="y" spacing="40" />
   <view id="container" layout="axis:y; spacing:-15"
         width="100%" height="300">
      <attribute name="goal_x" value="300"/>

      <method name="pop" args="show">
         rr1.setVisible(show);
         rr2.setVisible(show);
         rr3.setVisible(show);
         rr4.setVisible(show);
      </method>
```

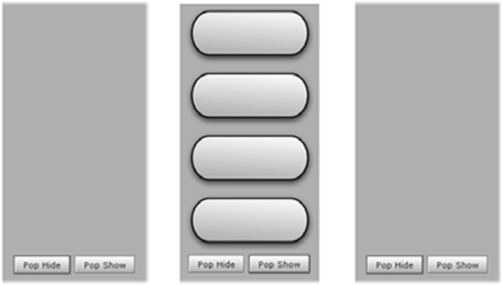
Toggles visibility

```
        <view id="rr1" resource="resources/roundrect.png"/>
        <view id="rr2" resource="resources/roundrect.png"/>
        <view id="rr3" resource="resources/roundrect.png"/>
        <view id="rr4" resource="resources/roundrect.png"/>
     </view>
     <view id="controls" x="15" layout="axis: x; spacing: 5">
        <button text="Pop Hide"
                onclick="container.pop(false)"/>
        <button text="Pop Show"
                onclick="container.pop(true)"/>
     </view>
  </canvas>
```

Handles visibility buttons

Figure 13.16 shows this group of panes popping into and out of view. The effect is so simple it doesn't require an animator; it's carried out simply by setting and resetting the `visible` attribute for each rounded rectangle.

The problem with this approach is that there is no transition, which produces a jarring effect. In the physical world, objects normally don't just pop into existence. A better approach is to animate the opacity of the four objects to produce a smooth transition. The resulting delay materializes the panes slowly and then gently fades them away to provide a calmer, more relaxed feeling. An analogous aural situation would be the difference between simply hitting the Stop button on a music player and first turning down the volume before stopping. Listing 13.8 shows the code for doing this.



(Initial) (Pop) (Hide)

Figure 13.16 Abrupt transitions should be avoided because they produce a jarring sensation.

Listing 13.8 Using opacity to smoothly transition objects into view

```
  <canvas bgcolor="0xAFAFAF">
     <simplelayout name="lay" axis="y" spacing="40"/>
     <view id="container"
           layout="axis:y; spacing:-15"
           width="100%" height="300">
        <view resource="resources/roundrect.png"/>
        <view resource="resources/roundrect.png"/>
        <view resource="resources/roundrect.png"/>
        <view resource="resources/roundrect.png"/>
     </view>
```

Contains layout for rectangles

```
      <view id="controls" x="20" layout="axis: x; spacing: 5">
         <button text="Fade In"
            onclick="container.                               Exposes panes
               animate('opacity', 1, 500);"/>
         <button text="Fade Out" >                        ❶ Fades panes
            <handler name="onclick">
               lay.lock();                                    Ensures buttons
               container.animate('opacity', 0, 500);          don't fade
            </handler>
         </button>
      </view>
   </canvas>
```

To make the panes fade away, we animated the `opacity` attribute to travel from a full opacity of 1.0 to invisibility at 0. To prevent the control buttons from moving up, the layout is locked ❶ before fading out the panes. Figure 13.17 shows this warmer transition.

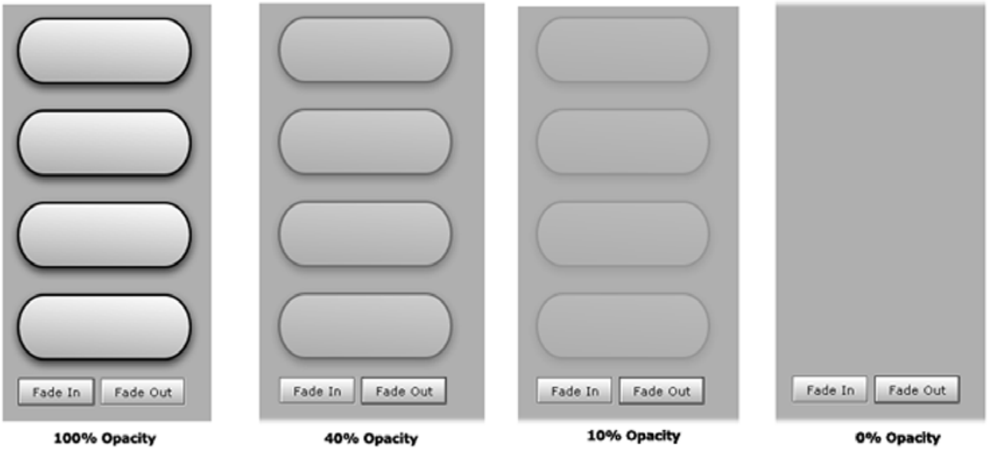


Figure 13.17 A smooth transition is accomplished by animating opacity.

Although animating an object's opacity is quite effective in producing a more relaxing sensation, we can take matters one step further by distributing the delaying effect among objects.

Using layouts to control delay

Up to this point, we have used a single delay effect against either a single object or group of objects. Now we'll use an offset to vary the delay for each member of a group. The easiest way to coordinate objects in a group is through the parent's subviews array. We'll apply an accumulative algorithm to each object in the array. Listing 13.9 contains a delay algorithm, known as the Simister Slide, written by Bret Simister of Laszlo Systems.

Listing 13.9 The Simister Slide delay algorithm

```
<canvas bgcolor="0xAFAFAF">
   <simplelayout axis="y" spacing="40"/>
   <view id="container"
         layout="axis:y; spacing:-15"
         width="100%" height="300">
      <attribute name="goal_x" value="100"/>

      <method name="slide">
         var farthest = goal_x;
         var duration = 500;
         var count = 0;
         for (var i in this.subviews) {
            var view = this.subviews[i];
            view.animate("x", farthest+(20*i), duration+(200*i));}
         goal_x = (goal_x == 100 ? -250 : 100);
      </method>
      <view resource="resources/roundrect.png" x="-250"/>
      <view resource="resources/roundrect.png" x="-250"/>
      <view resource="resources/roundrect.png" x="-250"/>
      <view resource="resources/roundrect.png" x="-250"/>
   </view>

   <view id="controls" x="15">
      <button text="Simister Slide" onclick="container.slide()"/>
   </view>
</canvas>
```

❶ Creates simplelayout for four panes

❷ Contains this simplelayout

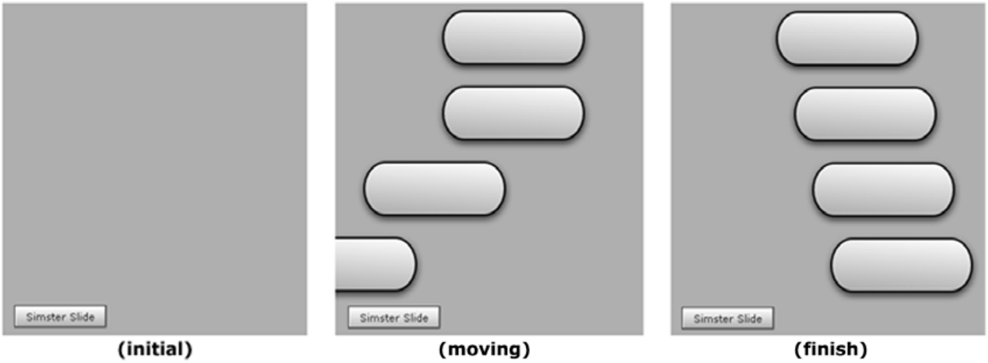
❸ Controls onscreen visibility

❹ Contains slide algorithm

A parent view ❷ called `container` contains an object group of four views (round rectangle images) that has a `simplelayout` ❶ applied to them to keep them consistently positioned. The container view has a `goal_x` attribute ❸ to indicate whether or not the objects are visible. The visibility is controlled by positioning the object group on-screen (100) or off-screen (–250).

Initially, the object group is located off-screen with the images stacked vertically. When moved to the right by the method `slide` ❹, each object in the subviews array below the top object moves a further distance over a longer duration than its predecessor. Since there is a fixed relationship between the increased distance and the duration, each travels at the same speed. The differences among the objects are manifested as a slight delay as each group member reaches its final position.

In earlier examples, we eased an object into and out of motion. That was a single or global sense of ease. But in this example, ease now exists among the objects. This further distribution provides a realistic impression that the objects accelerate from the start and glide to a halt. Finally, the `goal_x` attribute is flipped to ensure that the objects exit off-screen on a subsequent click of the button. The results of this application are shown in figure 13.18.



(initial) (moving) (finish)

Figure 13.18 The Simister Slide uses layouts to establish delay among the objects in a group, providing a realistic sense of acceleration before gliding to a halt.

Creating a subtle or fine-grained sense of delay provides the most realistic sensation that objects have mass. The art of creating compelling animation involves finding novel ways to emulate the physical laws of the real world.

We'll now turn to the Laszlo Market to provide an example of how animation can be used to provide a visual joke or pun to impart a sense of familiarity to an application's operations.

13.5.4 Animating the Market trashcan

We would like to make shopping at the Laszlo Market a more enjoyable experience. The best way to accomplish this is to make it more entertaining. We'd like to introduce a visual joke or pun. To ensure that this joke doesn't become tiresome, it needs a purpose. We'll leverage the operation of the physical world to provide more information to the user. When users are searching for something, they'd appreciate verbal hints like "you're getting warmer" or "you're getting colder." We can translate this verbal metaphor into a visual pun that suggests another feature of the physical world: temperature. It provides feedback to viewers helping them locate their mouse cursor directly over the trash to dispose of an item.

Listing 13.10 shows how this visual pun works. When the mouse cursor moves over the target, it starts an animated sequence that transitions from a light to a bright yellow to indicate higher temperature. When the cursor leaves the target, the animation transitions back through a light yellow color to a transparent state to indicate that the target is now "cold." When the dragged image is dropped, the background color abruptly changes to a reddish glow to indicate that the target was successfully hit. Afterward, it slowly transitions back to the transparent state to indicate that its contents were digested.

Listing 13.10 Animating the mouse cursor over the trashcan for user feedback

```
<canvas>
   …
   <view name="shoppingcart" … >
      …
      <view height="90" width="100%" y="${parent.height-90}">
         <view>
            <view resource="trashcan" clickable="true">
               <handler name="oninit">
                  LzTrack.register(this, "trash_target");
               </handler>
               <handler name="onmousetrackout">
                parent.anim.
                 setResource(null);
                parent.anim.                         Removes
                 dimming.doStart();                  glow
               </handler>
               <handler name="onmousetrackover">
                parent.anim.                                   Displays
                 setResource("lit_trashcan");                  glowing image
                parent.anim.                         Increases opacity
                 glowing.doStart();
               </handler>
               <handler name="onmousetrackup">
                parent.anim.                                   Displays
                 setResource("red_trashcan");                  red image
                parent.anim.                         Decreases opacity
                 dimming.doStart();
                main.shoppingcart.shopcart.deleteItem();
               </handler>
            </view>
            <view name="anim">
               <animator name="glowing" attribute="opacity"
                        from="0" to="1.0" duration="700"/>

               <animator name="dimming" attribute="opacity"
                         from="1.0" to="0" duration="1000"/>
            </view>
```

```
           </view>
           …
     </view>
     …
  </canvas>
```

We can easily add these animated sequences by replacing the changing background colors in the mouse event handlers with calls to the `anim` animator (indicated in bold). The animator changes the opacity of different resource images over 700 milliseconds to produce the gradual glow and fade of the trashcan. Figure 13.19 shows the trashcan images that correspond to the different mouse states.

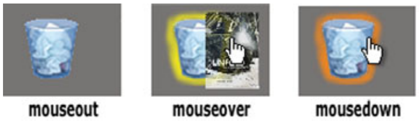


Figure 13.19 These figures illustrate the visual states involved with the disposal of a shopping cart item. First, the target is cold, then it is warm, and finally it lights up to dispose of the trash.

Although we previously provided feedback to indicate when the mouse cursor was over the target, that information was binary. Here we use the incremental nature of animation to add another dimension. With a binary "off" or "on," there is no difference between a close miss and a mile. There is no way to gauge the relative position of the mouse. Animation creates a convincing display of temperature, which is further twisted into a metaphor for positioning to allow the effect to work at several levels simultaneously.

Because novice store customers are already familiar with temperature, they have a sense of confidence that they are using the application correctly. A confident user is a happy user, one who feels that shopping at our store is an enjoyable experience (at least, so we hope).

13.6 Summary

This chapter introduced techniques to enhance an application's interface with visual design and animation elements that produce a more comfortable working environment. We want to avoid splashy effects that might initially enthrall users but that would quickly become tiresome. Instead we based the design of our interface elements on providing informative and functional services to users. We had already started this process in the earlier chapters, when animation was used to support multiple virtual screens in the Laszlo Market and to perform drag-and-drop operations. We have extended these principles to other interface elements, such as buttons and panes, to maintain a consistent appearance that corresponds to the actions of a user's mouse.

This chapter also introduced animation techniques that can be used to enhance presentation. These techniques are based on a set of principles that provide objects with physical properties governed by physical laws. The advantage of this technique is that it leverages existing user knowledge to provide an interface that appears intuitive. We also covered emotional issues that involve delay to produce a warmer ambience to our interface. Instead of providing abrupt state changes, the sensations of ease and delay are used to provide more relaxed interactions among objects. It is hoped that collectively these techniques result in an application interface that users find more enjoyable to use.

14 Branding an application

This chapter covers

- Handling cross-platform font issues
- Creating cross-platform source files
- Creating a simple customized component
- Creating a photo-realistic customized scrollbar

> *Your first appearance… is the gauge by which you will be measured; try to manage that you may go beyond yourself in after times, but beware of ever doing less.*
>
> —Jean-Jacques Rousseau,
> philosopher

Any product intended for general release requires a branded identity to distinguish it from competing products. Familiar brand identifications include McDonald's golden arches and Nike's swoosh. The previous chapter focused on providing a distinctive feel to the Laszlo Market; we now complement that with a tailored appearance. Not simply a sales-related tool, branding makes an application more accessible. And it's more than just an icon; the features that distinguish a Mac are more important than the Apple icon alone. Although we can't claim to produce anything as polished as an Apple product, we can still produce an identifiable personality for the Laszlo Market—one that's appropriate for our needs.

Achieving a unique identity requires outside creative design skills that developers generally don't possess. The skills of developers and designers are different enough to reliably produce divergent views. This generally results in more creative ideas and fresh approaches than can be produced by a single individual working alone.

All large-scale projects—those with budgets—are initiated in the same way: by gathering customer and focus group input to establish the application's tone. Establishing a correct tone is determined by the answers to questions such as these:

- Who is the target audience?
- Why do they need this application?
- What makes this application different from others?
- How, when, and where will this application be used?

Answering these questions is the engine propelling a design process. Our continually refined prototype methodology is intended to open the design process to input from a wide circle of outsiders. In many cases they have inside knowledge that is otherwise unobtainable, so their input is critical in avoiding the familiar trap of "a solution in search of a problem."

14.1 Creating an application-specific look

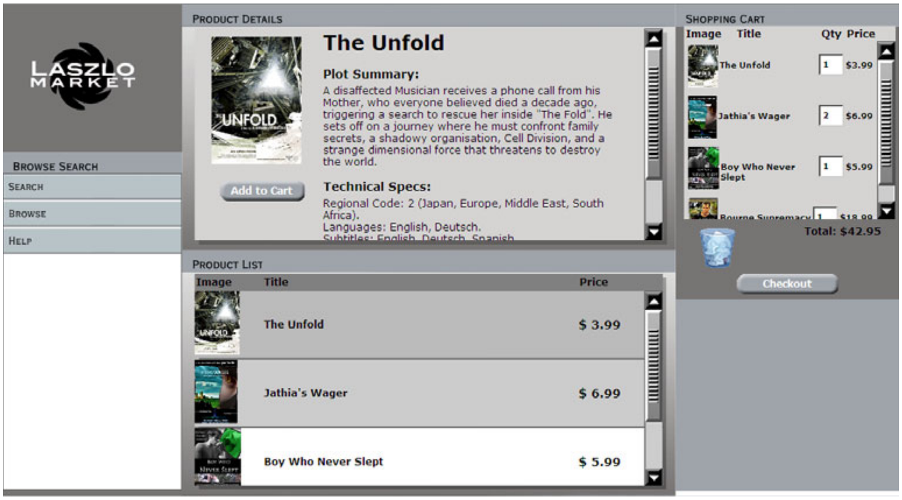
Since we're building a store to sell action-based videos, we need to identify the emotional qualities that customers associate with this genre. From sales research, we know our target demographic: males between the ages of 12 and 32. Ideally, our overall tone should so clearly identify this genre that we could remove all the store's products and the store would still be identifiable with action-based products. To achieve this, we choose the high-tech/comic book appearance shown in figure 14.1.

The tools that we'll use to create this branded appearance are font and color selection, logos, stylized components, and animation. We've left out one additional design element, video, which we'll discuss in chapter 15.

Up to this point, we've worked with core Laszlo features to avoid differences between the Laszlo Flash and DHTML implementations. One of our initial design goals for the Laszlo Market was to create a single set of source files that support common operations and appearance across both platforms. But now we begin to encounter some roadblocks with this approach. The problem is that Flash offers a wider range of graphical capabilities than DHTML. In particular, Flash supports these features not available in DHTML:

- Embedded fonts
- Vector images
- Audio and video media

We'll cover ways to simulate the first two features in DHTML using alternative methods. Although this simulation process won't provide the flexibility available to Flash, the final appearance will be indistinguishable. In the next chapter, we'll



The Unfold

Plot Summary:

A disaffected Musician receives a phone call from his Mother, who everyone believed died a decade ago, triggering a search to rescue her inside "The Fold". He sets off on a journey where he must confront family secrets, a shadowy organisation, Cell Division, and a strange dimensional force that threatens to destroy the world.

Technical Specs:

Regional Code: 2 (Japan, Europe, Middle East, South Africa).
Languages: English, Deutsch.
Subtitles: English, Deutsch, Spanish

Image	Title	Qty	Price
	The Unfold	1	$3.99
	Jathia's Wager	2	$6.99
	Boy Who Never Slept	1	$5.99
	Bourne Supremacy	1	$18.99

Total: $42.95

Image	Title	Price
	The Unfold	$ 3.99
	Jathia's Wager	$ 6.99
	Boy Who Never Slept	$ 5.99

Figure 14.1 The Laszlo Market is branded to project a high-tech/comic book theme.

examine DHTML and Flash features, including audio and video media, that require a hybrid application.

Let's start with the differences between the vector and bitmapped graphical capabilities of Flash and DHTML.

14.1.1 Vector and bitmapped graphics

Two major graphic systems, vector and bitmapped, are used to display images. Vector-based graphics use geometrical primitives such as points, lines, and curves to represent an image. Vector images are constructed mathematically so that an image can be numerically rotated, skewed, or stretched without causing distortion. Vector graphics are ideal for simple drawings such as line drawings and font character sets.

A bitmapped image, consisting of *pixel* building blocks, is characterized by its width and height in pixels and the number of bits (8, 16, or 24) per pixel. The number of bits per pixel determines the number of displayable colors, which, along with the total number of pixels—the *resolution*—determines the quality of a bitmapped image. Bitmapped images display more quickly and require less processing, but unlike vector images, they can't be resized without distortion, as shown in figure 14.2.

While Flash supports both vector and bitmapped graphics, DHTML only supports bitmapped graphics. In the next section, we'll explore the font-related ramifications of DHTML's inability to support vector images.

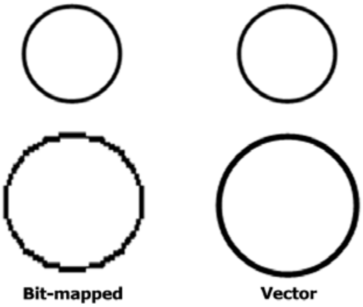


Figure 14.2 Vector images can be resized without the ragged distortion characteristic of resized bitmapped images.

14.1.2 Font differences

In typography, a *glyph* is the shape of a particular character in a typeface. Fonts can be represented as either vector-outlined or bitmapped. A bitmapped font stores each glyph as a *bitmap*. Since bitmapped images aren't resizable, bitmapped fonts require separate bitmaps for each size.

In contrast, vector-outlined fonts use Bézier curves and other mathematical tools to represent each glyph. These fonts are resized by applying a mathematical function to each point in a glyph, thus allowing a single vector-outlined font to display equally well across all sizes. Mathematical transformations can be applied to a glyph to achieve a wide range of special effects, such as rotation and varying opacity. While a vector font supports multiple sizes, it only supports a single style; additional styles such as bold or italic require another font package. The most popular font families—PostScript, TrueType, and OpenType—are based on vector-outlined fonts.

Browsers come prepackaged with a small collection of fonts from the underlying operating system, such as Verdana, Helvetica, and Courier, also known as *client* or *device fonts*. Because each font reflects the physical characteristics of its platform, there's no guarantee of identical display across different platforms.

Laszlo Flash can access both the browser's client fonts and any installed vector-outlined fonts. Since no two computers have the same set of installed vector fonts, we have to compile these fonts into our application and download them to the browser (hence the name *embedded font*). Any TrueType font file, identified by a .ttf suffix, can be used as an embedded font. TrueType fonts were originally introduced by Apple, and subsequently licensed by Microsoft; as a result, TrueType is a popular format containing both open source and commercial fonts.

Laszlo DHTML is subject to the limitations of DHTML, so its font selection is limited to a browser's set of client fonts. Another solution to expand the variety of available fonts in DHTML is simply to capture the font in an image. Capturing a font in a bitmapped image requires an image editor, such as Photoshop or GIMP. This approach significantly reduces flexibility since text can't be updated by changing a character string; instead, the graphical image must be updated. Despite its clumsiness, this approach allows a DHTML application to display a particular font. In the following section, we'll use this technique to display Copperplate fonts in the Laszlo Market's section headers.

14.1.3 Selecting a font

Although you may never *consciously* notice the fonts in an application, they do *subconsciously* affect your attitude toward it. Fonts can subtly establish a mood, ranging from formal to casual, modern to traditional, cool to warm. For our purposes, fonts are used either for decoration or for information. We simplify things by limiting decorative fonts to the main title. There are no rules governing the use of a decorative font; a designer has complete freedom to run wild with it.

Informative fonts are used for labels and other text that needs to be read and thus are subject to the cardinal rule of design: text must be legible. Although it's tempting to use distinctive fonts to stylize an application, we'll limit ourselves to fonts with a clean style, meaning they must be sans serif. Serifs are the decorative details at the end of letters, and sans means *without*. The result is a font that's physically less taxing to read, especially at small sizes.

To further increase legibility, be sure to limit the number of font styles in your application to one or two, and distinguish your sections by the font size instead. This approach results in a hierarchical appearance, with prominence indicated by a larger font. The haphazard use of different fonts generally produces a disorganized appearance. In the next section, you'll see how carefully selected fonts in combination can effectively complement one another.

Selecting a font for the Laszlo Market

For the Laszlo Market, we'll use a combination of sans serif Copperplate and Verdana for titles and text. This provides a classic appearance that works across a wide variety of genres. Copperplate complements Verdana by supporting built-in capitalization for titles; a word's initial letter is slightly enlarged and the remaining characters print in uppercase. This effect is subtle enough that it can't be accomplished by simply increasing the font size. This combination also uses an embedded and a client font to demonstrate their respective uses.

Verdana, a widely available client font, is installed on most computers. Copperplate is an open source embedded TrueType font freely downloadable from www.fonts.com and other websites. Copperplate comes in a variety of different styles, from which we select its Gothic style. Since DHMTL doesn't support embedded fonts, we'll develop classes to hide its implementation. But first, let's see how to use embedded fonts.

Declaring an embedded font with Laszlo Flash

Copperplate is added to Laszlo Flash as a font resource. This code can't be compiled for Laszlo DHTML; it would produce the error message "DHTMLWriter does not support importing fonts."

```
<font name="copperplate" style="bold"
   src="resources/Copperplate_Gothic_Bold.ttf"/>
```

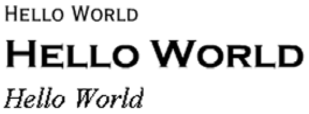
Once this font resource has been added, any view-based object can access it. Listing 14.1 uses the resource with text objects.

Listing 14.1 Creating and accessing a TrueType font resource

```
<canvas>
   <font name="copperplate" style="plain"
         src="resources/Copperplate_Gothic_Condensed_BT.ttf" />
   <font name="copperplate" style="bold"
         src="resources/Copperplate_Gothic_Bold.ttf" />
   <simplelayout axis="y"/>
   <text font="copperplate" fontsize="22">Hello World</text>
   <text font="copperplate" fontsize="32"
```

```
         fontstyle="bold" text="Hello World"/>
   <text font="copperplate" fontsize="22"
         fontstyle="italic" text="Hello World"/>
</canvas>
```

Figure 14.3 shows the results of the Gothic plain and bold font styles for the Copperplate font, along with a Copperplate italic font style that doesn't seem to match the others. The italic font displays so differently because Laszlo, upon encountering an unsupported font style, silently substitutes a default font, in this case Geneva. Laszlo does this because there's no telling which fonts a browser might possess for a particular platform.


HELLO WORLD

HELLO WORLD

Hello World

Figure 14.3 The top two lines are examples of the Copperplate font, plain and bold. The third line shows the result of attempting to display an unsupported Copperplate italic font.

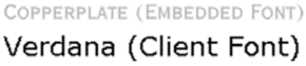
Because embedded vector fonts support special text effects, let's take a look at how to use them.

The advantages of embedded fonts

Embedded vector fonts support many text effects such as stretching, rotating, or varying opacity. In contrast, a client font can't be manipulated and can support only the opacity states of one (fully visible) or zero (invisible) with nonzero values displayed as fully visible. Listing 14.2 uses both font types to illustrate their differences.

Listing 14.2 Using special text effects with embedded vector fonts

```
<canvas>
   <font name="copperplate" style="plain"
         src="resources/Copperplate_Gothic_Condensed_BT.ttf"/>
   <simplelayout axis="y"/>
   <text font="copperplate" fontsize="22"
         opacity=".5" text="Hello World"/>
   <text font="Verdana" fontsize="22"
         opacity=".5" text="Hello World"/>
</canvas>
```


COPPERPLATE (EMBEDDED FONT)

Verdana (Client Font)

Figure 14.4 A client font cannot vary its opacity; it's either visible or invisible.

Figure 14.4 compares the results of setting a font's opacity to 0.5 for both an embedded and client font. An embedded font's ability to vary opacity allows text to be animated so it slowly fades away or becomes visible.

Embedded fonts do have one disadvantage, however. Since they are packaged with an application, download size and time increases. But generally the additional size isn't large enough to be critical.

Now that you've seen how to use embedded fonts with the Flash platform, we'll show you how to capture this font representation for the DHTML implementation of the Laszlo Market.

Replicating embedded fonts in DHTML

We'll organize the Laszlo Market screens into labeled sections composed of a title against a solid gray background with a black outline. Let's start with a Flash implementation using a `sectionheader` class to support multiple instances, which are differentiated with a `title` attribute. Listing 14.3 defines the title font, which is the embedded Copperplate font.

Listing 14.3 Laszlo Flash section header

```
<library>
   <font name="copperplate" style="plain"
         src="resources/Copperplate_Gothic_Condensed_BT.ttf"/>

   <class name="sectionheader">
      <attribute name="title" type="string"/>
      <view resource="section_header" width="${classroot.width}"
            stretches="width"/>
      <text x="10" y="5" font="copperplate" fontsize="18"
            text="${classroot.title}" resize="true"/>
   </class>
</library>
```

Our updated section header can easily be examined with a small test program:

```
<canvas>
   <simplelayout axis="y" spacing="4"/>
   <section_header title="Product Details" width="${canvas.width}"/>
   <section_header title="Product List" width="${canvas.width}"/>
</canvas>
```

which adds the titles to the stretched view to produce the consistent set of section headers shown in figure 14.5.

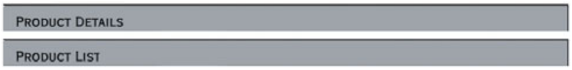


Figure 14.5 The Copperplate font permits capitalization with uppercase letters, providing a title style complementary to the Verdana font.

To replicate this with DHTML, we'll create a series of bitmapped images containing these Copperplate titles and store them in files with matching names. These titles are given a transparent background, using an image editor such as Photoshop or GIMP. Then, they can be applied like a decal to the stretched `section_header` image. Listing 14.4 shows the section header template that we'll use to produce our titles.

Listing 14.4 Laszlo DHTML section header

```
<library>
   <class name="sectionheader">
      <attribute name="title" type="string"/>
      <view x="10" y="5">
         <method name="init">
            this.setSource(classroot.title + '.png');
         </method>
      </view>
<view resource="section_header"
      width="${classroot.width}"
      stretches="width"/>
   </class>
</library>
```

Stretches section header image

Despite the inconvenience, this produces an appearance and interface identical to that of the `section_header` object. The two can be packaged into two libraries, flash_class.lzx and dhtml_classes.lzx. Then, it's only necessary to invoke the correct runtime library.

14.1.4 Choosing between DHTML and Flash implementations

Maintaining a single set of source files that encompasses both platforms requires a way to dynamically select between these implementations at runtime. The `lzr` request type parameter in the URL query string can be used to set the `runtime` attribute to a selected platform:

```
http://localhost:8080/lps/Test/main.lzx?lzr=dhtml
```

When this `runtime` attribute is set, it enables the `switch`, `when`, and `otherwise` tags to select platform-specific code. Listing 14.5 shows how to include the appropriate libraries for a selected implementation.

Listing 14.5 Including the correct runtime library based on the URL parameter

```
<canvas >
   <switch>
      <when runtime="dhtml">
         <include href="dhtml_classes.lzx/>
      </when>
      <when runtime="swf7">
         <include href="flash_classes.lzx"/>
      </when>
      <when runtime="swf8">
         <include href="flash_classes.lzx"/>
      </when>
      <otherwise>
         <include href="flash_classes.lzx"/>
      </otherwise>
   </switch>
</canvas>
```

Although Laszlo defaults to an `swf8` runtime, its matching `when` branch is triggered only when an `lzr` request type has been explicitly set. When the URL doesn't contain an `lzr` parameter, the `otherwise` tag is executed.

Although the DHTML approach for supporting vector fonts is less elegant than Flash's, it can still be used to duplicate the appearance of a Flash implementation. This allows us to maintain a single code base that provides an identical appearance across both platforms.

We'll next design a set of custom components that can be used by both the Flash and DHTML implementations.

14.2 Branding with custom components

Just as chrome trim on a car gives a distinctive appearance, custom components give an identity to an application. More important, they can draw the viewer's attention to important interface controls. Consequently, design time spent on custom components is more important than time spent on background images. Fortunately, Laszlo includes all component code and image files in a distribution, which simplifies the development of custom components.

For the Laszlo Market, we'll use the behavior of existing components, but customize their appearance. We'll demonstrate how to leverage the design of existing components to create a photo-realistic appearance.

All Laszlo components are written in LZX, with their source code stored in the $LPS_HOME/lps/components directory. A component's code is separated into a `base` directory, dealing with behavior, and an `lz` directory, dealing with appearance. Because we're just changing the appearance, we're concerned only with the `lz` directory.

The image resources supporting a component's appearance aren't available as attributes, so we can't override them with a subclass. Instead, it's necessary to create a new class for a custom component. Since bitmapped images are supported in both platforms, we'll choose the bitmapped PNG format for our resources.

We'll focus on custom `tabelement` and `scrollbar` components. A custom component needs a unique name to avoid conflicts with the stock Laszlo components. We'll add a `my` prefix to the class name, resulting in `mytabelement` and `myscrollbar` components. To house our new components, we'll need a local `components` directory. All component source files are neatly packaged for easy copying. Here are the steps to do this:

1. Create a local `components` directory.
2. Copy the `tabelement` and `scrollbar` source files from $LPS_HOME/lps/components into this directory and rename them by adding the `my` prefix.
3. Create a local `resources` directory.
4. Copy the `tabelement` and `scrollbar` resource files from $LPS_HOME/lps/components/resources into this directory. It isn't necessary to rename any of these files, since they now have a unique path.

Now we're ready to start constructing our custom components; let's begin with `mytabelement`, the simpler of the two.

14.2.1 Customizing the tabelement component

Before we start, let's review how a `tabslider` works. It's composed of two element types: a `tabslider` and multiple `tabelements`. The `tabslider`'s appearance is minimal since its purpose is to contain and control the `tabelement`. Consequently, only the `tabelement`'s appearance needs to be altered.

By default, a `tabelement` has a height of 22 pixels; its width is inherited from the `tabslider` and its text is left-aligned. These values and the font settings can be changed through its attributes. A `tabelement` consists of these three views, also shown in figure 14.6:

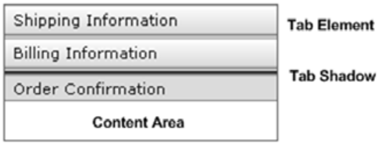


Figure 14.6 Three design elements in a tabelement are available for modification.

- The *tab element* contains a title against a background image.
- The *tab shadow* is positioned below the open tab.
- The *content area* is positioned at the bottom.

A tabelement has three visible mouse states: up, over, and down. These states correspond to frames within a resource. To create our custom component, we'll change the tabelement's resources to match the appearance shown in figure 14.7.

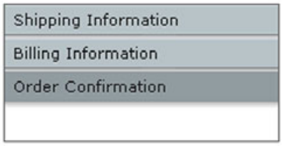


Figure 14.7 The updated tabelement appears more in tune with the overall look of the Laszlo Market.

Laszlo has a single address space and resources are globally defined at the top level so it's also necessary to update resource definitions to have a my prefix. Listing 14.6 indicates in bold all renamed resources along with their updated PNG image files.

Listing 14.6 Customizing the tabelement

```
<library>
   <include href="utils/layouts/resizelayout.lzx"/>
   <include href="../base/basebutton.lzx"/>
   <include href="../base/basetabelement.lzx"/>

   <!--- Tabelement button resource -->          Contains multiframe
   <resource name="mytabrsrc">               <--  image resource
      <frame src="resources/tabslider/tab_slider_up.png"/>
      <frame src="resources/tabslider/tab_slider_over.png"/>
      <frame src="resources/tabslider/tab_slider_down.png"/>
   </resource>

   <!--- Tabelement shadow resource -->
   <resource name="mytabshadow"
             src="resources/tabslider/                  Uses local stock
                  tab_element_shdw.png"/>           <--  shadow image

   <class name="mytabelement" extends="basetabelement"
          styleable="true">     <--  Defines
   ...                              mytabelement class
   <basebutton name="bkgnd"
               resource="mytabrsrc"                         Accesses new images
               styleable="true" … />                   <--  for tabelements
      …
      <view name="shdw" resource="mytabshadow" stretches="both" … />
      …
   </class>
</library>
```

We'll update our library.lzx file to make this new customized mytablement component available to our application. Listing 14.7 contains the updated `checkout` class that includes it in the Laszlo Market.

Listing 14.7 Including the custom mytabelement component

```
<class name="checkout">
   …
   <tabslider height="${parent.height}"
                    width="${immediateparent.width}"
                    spacing="2" slideduration="300">
      <mytabelement text="Shipping Information"
                 font="copperplate" fontsize="16">
         <shippinginfo name="shiptab"/>
      </mytabelement>
      <mytabelement text="Billing Information"
                 font="copperplate" fontsize="16">
         <billinginfo name="billtab"/>
      </mytabelement>
      <mytabelement text="Order Confirmation"
                 font="copperplate" fontsize="16"/>
         <orderconfirm/>
      </mytabelement>
   </tabslider>
</class>
```

Figure 14.8 shows the result of our custom `mytabelement`. Since we only changed its appearance, it still operates like a standard `tabelement` in a `tabslider`.

Titles appear in the Copperplate font, which means we still need to update the font resource for the DHTML implementation. Listing 14.8 shows how a DHTML implementation would access this title font.

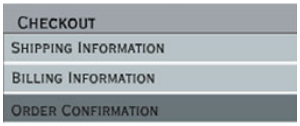


Figure 14.8 The Checkout window of the Laszlo Market is updated to include the custom tabelement.

Listing 14.8 Font resource substitution for DHTML

```
<when runtime="dhtml">
   …
   <mytabelement minheight="30">
      <handler name="oninit">
         this.top.header.setSource("Shipping Information" + ".png");
      </handler>
   </mytabelement>
   …
</when>
```

Now we're ready to tackle more complex components that use interacting image resources to produce dynamic lighting effects.

14.2.2 Creating a custom scrollbar

When creating a custom scrollbar, your natural inclination may be to hire a designer to create something ornate. Unfortunately, you'd probably be disappointed; an image that looks great on paper can appear lifeless in an application. The key step in creating realistic custom components is to ensure that its parts are illuminated by a common light source. Although the light source can come from any direction, Laszlo's components default to the upper-left corner. Laszlo designers have already done the hardest part for you—the logic interconnecting the scrollbar parts.

It's easiest to conform to the default lighting source. We only have to put light colors on the top and left and darker colors on the bottom and right. This turns out to be pretty easy, doesn't require artistic training, provides flexibility for changes, and produces very professional results.

A stock scrollbar consists of 15 images, illustrated in figure 14.9, contained in three groupings: eight arrow-button images supporting the top and bottom button mouse states for up, over, down, and disabled; three track images for the states of up, down, and disabled; and finally, three handle images for the top, middle, and bottom, along with the gripper.

Although a scrollbar can be vertical or horizontal, we'll work only with a vertical scrollbar to simplify things. These concepts can easily be applied to a horizontal scrollbar. Let's start by looking at each of the parts.

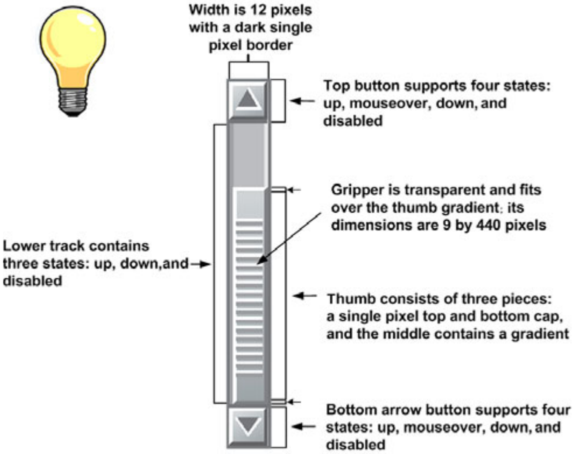

Width is 12 pixels with a dark single pixel border
Top button supports four states: up, mouseover, down, and disabled
Gripper is transparent and fits over the thumb gradient; its dimensions are 9 by 440 pixels
Lower track contains three states: up, down,and disabled
Thumb consists of three pieces: a single pixel top and bottom cap, and the middle contains a gradient
Bottom arrow button supports four states: up, mouseover, down, and disabled

Figure 14.9
Fifteen parts contribute to the construction of a scrollbar conforming to a common lighting source, represented by the lightbulb in the upper-left corner. The top and bottom buttons each contain four pieces, the track contains three, and the handle consists of four pieces—top, middle gradient, bottom, and a gripper to fit over the gradient.

It's necessary to understand how the different parts fit together. The width of a stock Laszlo scrollbar is 14 pixels minus a single-pixel black border, leaving 12 pixels for the lower track, which cradles the handle. To conform to our light source, the track image, shown in figure 14.10, has a dark pixel on the left side, representing a shadow, and a light pixel on the right side, representing a lighted region. The image file is stretched along its length to produce the track. The track's color pattern indicates that the lighting source is on the left.

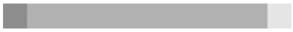


Figure 14.10 When stretched, the dark and light pixels are seen as strips of shadow and highlight the track bed for the handle.

Enlarging the Laszlo arrow-button images shows how buttons are aligned to a common light source. Figure 14.11 compares the vector art for the default set of buttons with the bitmapped art for our updated buttons. The arrows have a dark region under each button, and the angle of the triangle's white edges indicates that the lighting source is above. In addition, the images for the up and over states use an angled gradient, whereby the lightest region corresponds to the upper-left corner. This pattern is reversed for the down state to provide a sense of depth. The same lighting pattern is used for our bitmapped versions of these arrow-button images.

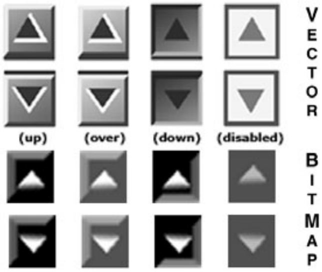


Figure 14.11 These images compare the vector and bitmap-based scrollbar buttons representing the mouse states up, over, down, and disabled.

Let's now move to the three pieces of the handle, shown on the left in figure 14.12: the top angled edge, the middle flat surface with angled side edges, and the bottom angled edge. The design of the handle is consistent with a single light source. The top and bottom edges have contrasting gray shades representing light and shadow, while the middle piece has a highlighted left edge and a darkened right edge bracketing a gradient. The top and bottom pieces are one pixel high and work as end caps, while the middle piece is stretched vertically. When put together, they create the slab shown on the right.

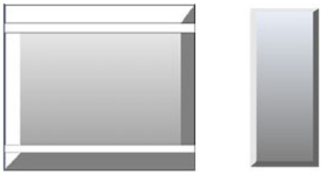


Figure 14.12 The three parts comprising the scrollbar's handle are shown enlarged and out of scale on the left and together as a unit in scale on the right. The top and bottom sections have a height of one pixel. The light lines mark vertical pixels.

All that remains is to attach the gripper. The gripper image is a consistent pattern of white and gray stripes on a transparent background displaying a background gradient. This provides each stripe with a subtle difference in shadowing, as shown in figure 14.13, while still conforming to the light source.

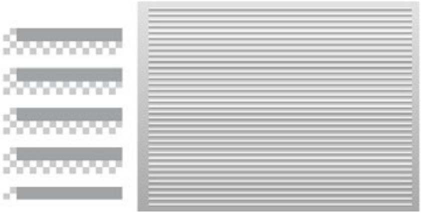


Figure 14.13 The gripper is an alternating sequence of white and gray bars on a transparent background, represented here with a checkerboard. The image on the right demonstrates the visual effect of a gradient against a pattern.

The length of the gripper image is fixed to prevent it from being stretched; otherwise, the symmetry of the alternating sequence would be lost and the strip would distort. Since a tall window requires a long gripper, its image is a long (440-pixel) PNG file clipped to the parent's height. The end result is a gripper that resizes to the window's height.

Now that we understand the relationship of the parts, we can use this architecture to build our custom scrollbar.

Customizing the scrollbar

The default Laszlo scrollbar has an art deco look that isn't appropriate for our target audience. We want to tweak the default scrollbar to create a fatter, jazzier scrollbar that works with both Flash and DHTML. For the fatter look, we'll increase the gripper width to 20 pixels. This is easily done by updating the gripper image using an image editor. We'll create a more elaborate grip with a small hook along the right side, as shown in figure 14.14. This hook shape continues to face toward the light source, but now contains a rich shadow for more depth.

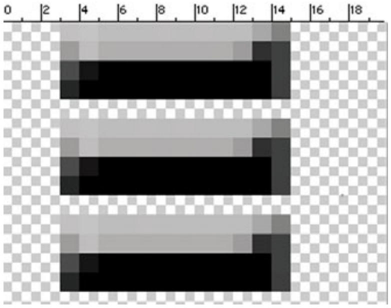


Figure 14.14 A bitmapped image enhances the gripper with a short, right-aligned hook and color gradations.

Since the gradient background is no longer a vector image that can be continuously stretched, we'll need to increase the size of the gradient pattern to match the 440-pixel length of the gripper. The top and bottom images are also enlarged to spread across the additional width.

The scrollbar has myscrollbar_xresources.lzx and myscrollbar_yresources.lzx files listing its resource image files. We just have a vertical scrollbar, so we only have to update the myscrollbar_yresources.lzx file. So altogether there are 15 PNG files to be updated with new images. Listing 14.9 shows the updated library code.

Listing 14.9 Updated vertical scrollbar resources in myscrollbar_yresources.lzx

```
<library>
   <resource name="myscrollbar_ythumbtop_rsc"
             src="resources/scrollbar/scrollthumb_y_top.png"/>
   <resource name="myscrollbar_ythumbmiddle_rsc"
             src="resources/scrollbar/scrollthumb_y_mid.png"/>
   <resource name="myscrollbar_ythumbbottom_rsc"
             src="resources/scrollbar/scrollthumb_y_bot.png"/>
   <resource name="myscrollbar_ythumbgripper_rsc"
             src="resources/scrollbar/thumb_y_gripper.png"/>

   <resource name="myscrollbar_ybuttontop_rsc">
      <frame src="resources/scrollbar/scrollbtn_y_top_up.png"/>
      <frame src="resources/scrollbar/scrollbtn_y_top_mo.png"/>
      <frame src="resources/scrollbar/scrollbtn_y_top_dn.png"/>
      <frame src="resources/scrollbar/scrollbtn_y_top_dsbl.png"/>
   </resource>
   <resource name="myscrollbar_ybuttonbottom_rsc">
      <frame src="resources/scrollbar/scrollbtn_y_bot_up.png"/>
      <frame src="resources/scrollbar/scrollbtn_y_bot_mo.png"/>
      <frame src="resources/scrollbar/scrollbtn_y_bot_dn.png"/>
      <frame src="resources/scrollbar/scrollbtn_y_bot_dsbl.png"/>
   </resource>
   <resource name="myscrollbar_ytrack_rsc">
      <frame src="resources/scrollbar/y_scrolltrack.png"/>
      <frame src="resources/scrollbar/y_scrolltrack_dn.png"/>
      <frame src="resources/scrollbar/y_scrolltrack_dsbl.png"/>
   </resource>
</library>
```

The next step is to update the `myscrollbar` class with these new resources. This class contains two main states corresponding to the x- and y-axes for horizontal and vertical scrolling. Because our scrollbar is vertical, we need only to change the y state. Stylizing a scrollbar doesn't require that you understand every detail of its operation; it's only necessary to know which resource and width settings—shown in bold in listing 14.10—must be updated.

Listing 14.10 Updating resources for vertical scrolling in myscrollbar.lzx

```
<class name="myscrollbar" extends="basescrollbar" bgcolor="0x595959">
   …
   <state apply="${parent.axis == 'y'}">
      <attribute name="width" value="20"/>

      <view name="toparrow">
         <basescrollarrow x="1" y="1"
            resource="myscrollbar_ybuttontop_rsc"
            direction="-1"/>
      </view>
      <view name="scrolltrack">
         <basescrolltrack name="top" x="1"
            resource="myscrollbar_ytrack_rsc" stretches="height"
            overResourceNumber="0" downResourceNumber="2"
            disabledResourceNumber="3" direction="-1">
            <attribute name="height" value="${parent.thumb.y}"/>
            <attribute name="width" value="${parent.width}"/>
         </basescrolltrack>
         <basescrollthumb name="thumb" x="1">
            <view resource="myscrollbar_ythumbtop_rsc"/>
            <view resource="myscrollbar_ythumbmiddle_rsc"
                  stretches="both"/>
            <view resource="myscrollbar_ythumbbottom_rsc"/>
            <stableborderlayout axis="y"/>
            <view resource="myscrollbar_ythumbgripper_rsc"
                  y="4" x="1" width="20"
                  height="${Math.min(200, parent.height-46)}"
                  clip="true" valign="middle"/>
         </basescrollthumb>
         <basescrolltrack name="bottom" x="1"
            resource="myscrollbar_ytrack_rsc" stretches="height"
            overResourceNumber="0" downResourceNumber="2"
            disabledResourceNumber="3">
            <attribute name="y"
                       value="${parent.thumb.y+parent.thumb.height}"/>
            <attribute name="height"
                       value="${parent.height - parent.thumb.y –
                       parent.thumb.height}"/>
            <attribute name="width" value="${parent.width}"/>
         </basescrolltrack>
      </view>
      <view height="20" name="bottomarrow">
         <basescrollarrow x="1"
            resource="myscrollbar_ybuttonbottom_rsc"/>
      </view>
      <stableborderlayout axis="y"/>
   </state>
</class>
```

Updates scrollbar width to 20 pixels

Displays up arrow

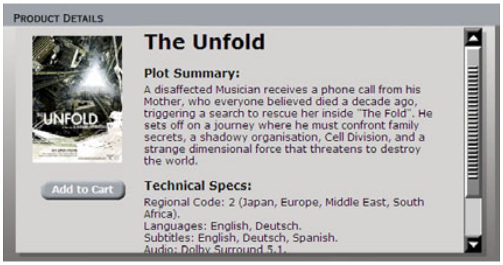
Displays track cradling handle

Displays top track segment

Displays handle with gripper

Displays bottom track segment

Displays down arrow



PRODUCT DETAILS

The Unfold

Plot Summary:

A disaffected Musician receives a phone call from his Mother, who everyone believed died a decade ago, triggering a search to rescue her inside "The Fold". He sets off on a journey where he must confront family secrets, a shadowy organisation, Cell Division, and a strange dimensional force that threatens to destroy the world.

Technical Specs:

Regional Code: 2 (Japan, Europe, Middle East, South Africa).
Languages: English, Deutsch.
Subtitles: English, Deutsch, Spanish.
Audio: Dolby Surround 5.1.

Figure 14.15 By conforming to a consistent light source, both the multistate button and the scrollbar provide a realistic sense of depth. This produces the illusion that these controls are protruding as foreground controls against a flat background.

Although this might seem like a lot of work, it's actually quite easy; our updated scrollbar, you can see in figure 14.15, shows that our efforts are well rewarded. We can easily test this custom scrollbar by adding it to the Laszlo Market.

Tweaking components is a great way to produce an individual look that differentiates your work from competitors. You can easily create a library of different custom components with a tailored appearance appropriate for different markets. Since you are leveraging the design logic in the existing assets, created by the professional designers at Laszlo, you'll generally have impressive results.

14.3 Summary

In this chapter we examined our application's purpose and identified its target audience. We used this information to create a branded appearance that hopefully resonates with that audience. We selected, based on our product offerings and our customer base, a high-tech/comic book theme. Our goal was to provide an identical operation and appearance with both Flash and DHTML.

To achieve this goal, we explored the properties of bitmapped and vector images to arrive at a suitable design compromise. We then examined how each platform can implement this design compromise to provide identical functionality and appearance.

Finally, we used these principles to create stylized components for tabelements and scrollbars. We showed how components are based on a number of moving parts and how the interaction of these parts provides a dynamic appearance. The operation of these parts ensures that a component's appearance is always consistent with a common lighting source, which is the secret to creating components with a photo-realistic appearance. The source code and image files provide a strong base to create tweaked versions of components, since we only need to maintain a common lighting source.

In the next chapter, we'll cover platform features that can't be simulated using alternative means. This requires the creation of a hybrid application so that one platform can access features of the other.

15 Integrating DHTML and Flash

This chapter covers

- Embedding Laszlo applications in HTML
- Intermixing Laszlo Flash and DHTML
- Building a search engine–accessible application
- Streaming with the Red5 server
- Accessing Flash video from DHTML

If scientific reasoning were limited to the logical processes of arithmetic, we should not get very far in our understanding of the physical world. One might as well attempt to grasp the game of poker entirely by the use of the mathematics of probability.

—Vannevar Bush (1890–1974),
academic, engineer, and visionary

So far, we've managed to preserve a level of compatibility between our Laszlo Flash and DHTML implementations and still maintain a single set of source code files. Where a platform is missing a feature, we improvised with workarounds. Although this requires some artistic compromises, with careful planning we were able to minimize any sacrifices in quality. In this chapter, we'll address features for which there are no workarounds. These features can only be supplied by melding Laszlo Flash and DHTML modules into a hybrid application.

15.1 Advantages of a hybrid approach

The Flash and DHTML platforms have relative strengths and weaknesses. Depending on your particular need, you might find that one is better suited for your application than the other. But rather than picking one or the other, we prefer packaging Laszlo Flash and DHTML modules into a single, hybrid application, since each implementation is missing critical features available in the other. Let's look at some of these critical issues and later we'll show you how they can be overcome.

A critical problem facing Flash-based applications is that its SWF file format is binary, while the Internet is composed of text-based HTML applications. The Web is so large that finding anything requires it to be prominently listed by a search engine such as Google. But the web crawlers that build these listings are only designed to work with text-based applications and can't index or rank the content of Flash applications. This situation isn't likely to change in the immediate future. We'll resolve this issue by packaging Laszlo Flash applications in an HTML wrapper to make them accessible to web crawlers. Later we'll demonstrate how the Laszlo Market can easily be made accessible to search engines.

A critical shortcoming facing DHTML applications is their limited multimedia capability. Today's users expect audio and video presentation tightly integrated within their web applications. But browsers don't possess audio or video capabilities, instead relying on plug-in software modules. We'll use a Laszlo Flash module to supply streaming video capabilities to a Laszlo DHTML implementation of the Laszlo Market.

Having seen the advantages of these implementations working together, the next step is to find a common environment for combining their features.

15.2 Using an HTML wrapper

All programming environments, from Unix to Java, employ the same model to provide a common environment for application packaging and deployment. They enclose or *wrap* their a.out or Java class executables within a Unix shell or Windows BAT file, an approach we'll refer to as *wrapping*. This is the model that we'll use as well.

Using a shell or Windows BAT wrapper to contain executables provides an outer meta-layer that allows you to control execution by setting environmental variables. It also provides an enclosure to package multiple executables in a single application body. We'll do something similar by wrapping our Laszlo DHTML and Flash executables in an HTML file. In a shell or Windows BAT wrapper, users and other programs can set or alter environmental variables to control an application's processing. In our HTML wrapper, we'll use browser JavaScript variables to function as web-based environmental variables.

A shell or Windows BAT wrapper allows applications to execute a new shell wrapper to initiate a recursive application sequence. Similarly, Laszlo applications can execute a new HTML file to initiate this recursive sequence. Later, we'll demonstrate each of these topics.

But to understand the mechanics of this approach, let's start by examining how a browser executes a Laszlo application.

15.2.1 Embedding Laszlo applications in HTML

Browsers are designed to render HTML documents. Although they can render common data types such as text, GIF, and JPEG images, and fonts, they require an external application, or plug-in, for unsupported data types such as audio, video, applets, or Flash SWF files. HTML supports several tags to work with these unsupported data objects; the `embed` tag supports external plug-in programs. HTML 4.0.1 introduced the `object` tag to support any generic data object. They may use different tags, but all browsers require these unsupported data type objects to be identified and transferred to the appropriate plug-in to be executed.

A Flash-enabled browser contains separate DHTML and Flash rendering engines that process JavaScript or ActionScript (a JavaScript derivative). As figure 15.1 shows, all Laszlo applications consist of an HTML file containing either

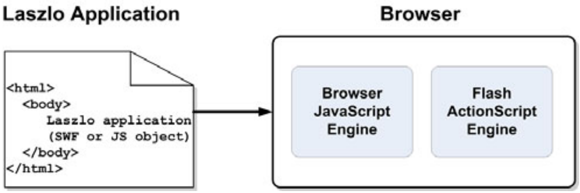


Figure 15.1
Laszlo applications consist of an HTML file containing an embedded SWF data object or DHTML object. The browser processes the HTML and delivers the Laszlo application to either the DHTML or Flash rendering engine.

an embedded Flash SWF or DHTML JavaScript object that executes in one of the browser's rendering engines.

A Laszlo DHTML application acts like a natural extension of its surrounding HTML page and is mapped directly to the web page's Document Object Model (DOM). In contrast, a Flash executable acts more like a foreign body encased within an HTML membrane. Despite these differences, Laszlo provides a common set of interface methods to support both application types.

To demonstrate how all Laszlo applications are contained within HTML, let's start by examining how a browser handles a SOLO Flash application contained in an SWF file. You'll need to display a sample SOLO executable like the OpenLaszlo `clockblox` in your browser:

```
http://www.openlaszlo.org/apps/clockblox.lzx.swf
```

This should display a clock. While all browsers provide a View Source window, this facility is of limited use as it only shows the static HTML source for a URL. When a browser displays a data object, it needs to dynamically generate HTML for the object. To view all HTML statements, you must use a special browser extension program such as Firebug, the DOM Inspector, or one of several others. These tools are necessary for creating Laszlo hybrid applications. You can download and install them from either of the following:

```
https://addons.mozilla.org/en-US/firefox/addon/1843
http://www.mozilla.org/projects/inspector/
```

Since these are browser plug-ins, the browser will automatically ask to install them when they are downloaded. Each of these plug-ins contains extensive documentation. Although they provide a wide assortment of features, we only need them to display a complete HTML source listing.

We'll walk through the steps to activate and use Firebug to display the HTML for the `clockblox` URL. After Firebug has been downloaded and installed, and the `clockblox` URL is accessed, you only have to press F12 to toggle Firebug on and off. The complete HTML source listing for this `clockblox` SWF object, viewable under Firebug's HTML menu, will look like this:

```
<html>
   <body marginwidth="0" marginheight="0">
      <embed width="100%" height="100%" name="plugin"
              src="http://www.openlaszlo.org/apps/clockblox.lzx.swf"
             type="application/x-shockwave-flash"/>
   </body>
</html>
```

The wrapped object is encased in an `embed` tag that identifies it as an `x-shockwave-flash` data object to prepare it for execution by the Flash engine. Microsoft Internet Explorer would also dynamically generate HTML, but it would instead encase the object within an `object` tag. The upcoming sections will demonstrate the advantages of using Laszlo's application-embedding features to manually enclose these executables in an HTML wrapper.

15.2.2 Examining HTML files created by Laszlo

Let's look at a Laszlo application delivered from an OpenLaszlo server. An OpenLaszlo server always delivers a Flash or DHTML application enclosed in an HTML file, consisting of two sections: `head` and `body`. The `head` section is used for declaring libraries and setting configuration values, while the `body` section makes the calls to generate the presentation of the web page.

The content of our `head` section is shown in listing 15.1. Because this section can be reused by many applications, we'll put it into an included header file. There are numerous ways to include the header, but we'll use two solutions: server-side includes (SSIs)—to include a header.ini file for SOLO applications; and JSP includes—to include a header.jsp file for OpenLaszlo server-delivered applications.

The header.ini file can be included in an HTML page like this:

```
<!-- include virtual="header.ini"-->
```

while the header.jsp file can be included like this:

```
<%@ include file="header.jsp"/>
```

For now, we have to hard-code the OpenLaszlo Server version of 4.1 into the call. This value should reflect the OpenLaszlo Server version that you are using. In a later section, we'll show how this can be dynamically set.

Listing 15.1 Laszlo DHTML head section (header.jsp or header.ini)

```
<head>
   <noscript>
      Please enable JavaScript in order to use this application.
   </noscript>
   <script type="text/javascript"
```

```
      src="lps-4.1/lps/includes/
           embed-compressed.js">
   </script>
   <script type="text/javascript">
      lzOptions={"lps-4.1/lps/resources"};
   </script>
   <script type="text/javascript">
      Lz.dhtmlEmbedLFC("lps-4.1/lps/" +
            "includes/lfc/LFCdhtml.js");
   </script>
   <style type="text/css">
      html, body {
        height: 100%; width: 100%;
        margin: 0; padding: 0;
        border: 0 none; overflow: hidden;
      }
   </style>
</head>
```

Includes Lz library

Specifies root location

Includes LFC for DHTML

Sizes application to browser

The first step is to use the `noscript` tag to check that JavaScript hasn't been turned off in the browser. If it is off, an appropriate message is displayed. Next we include the `Lz` library contained in the embed-compressed.js file. The `Lz` API, described in table 15.1, provides JavaScript methods to embed Flash or DHTML applications and support additional methods that allow the output to be manipulated.

Table 15.1 `Lz` calls to embed a Laszlo DHTML or Flash application

Name	Target	Description
`dhtmlEmbed`	DHTML	Loads a Laszlo DHTML application
`dhtmlEmbedLFC`	DHTML	Loads the Laszlo LFC library; only necessary for DHTML
`swfEmbed`	Flash	Loads a Laszlo SWF application

The next step is to use the `dhtmlEmbedLFC` method to load the Laszlo `LFC` library for DHTML applications. The DHTML implementation requires two steps for embedding since it has a separate `LFC` library. Laszlo created a separate `LFC` library that it needs to load only once; after that, it is cached in the browser so it doesn't have to be reloaded for subsequent applications. This decreases the download time when multiple applications are being loaded.

Next, we set the `lzOptions` variable to provide specific values required by either implementation for certain operations. In this case, DHTML needs the root path set to support display components.

Finally, we use Cascading Style Sheets (CSS) to set the browser's display characteristics. This ensures that applications are sized to the browser's maximum screen space, removes any padding or margins, and prevents the browser from scrolling.

Now that we've set everything up in the head section, we'll demonstrate the application type combinations by embedding DHTML or Flash applications for SOLO and server implementations. We've created a set of simple applications, dhtml.lzx and flash.lzx, consisting of a labeled shaded box, to illustrate how to embed an application. Since our example applications differ only in background color, we'll only display the listing for the DHTML version:

```
<canvas>
   <attribute name="message" value="no label" type="string"/>
    <view id="main" height="100" width="100" bgcolor="0xBBBBBB">
       <method name="setMsg" args="str">
          msg.setText(str);
       </method>
       <text name="msg" text="${canvas.message}" resize="true"
             align="center" valign="middle"
             fontsize="14" fontstyle="bold"/>
   </view>
</canvas>
```

The first step is to get the OpenLaszlo server to compile the example applications, dhtml.lzx and flash.lzx, into corresponding SOLO files for each platform. To perform this compilation for the DHTML platform, we'll enter the OpenLaszlo server's URL along with the file path, application (dhtml.lzx), and the proxied=false and lzr=dhtml URL parameters:

```
http://localhost:8080/lps-4.1/book/dhmtl.lzx?proxied=false&lzr=dhtml
```

The second step is compiling for the Flash platform by entering the OpenLaszlo server's URL along with the file path, application name (flash.lzx), and the proxied=false and lzr=swf8 URL parameters:

```
http://localhost:8080/lps-4.1/book/flash.lzx?proxied=false&lzr=swf8
```

Now we have SOLO applications, dhtml.lzx.js and flash.lzx.swf?lzr=swf8, for the Flash and DHTML platforms.

15.2.3 Embedding Laszlo applications in HTML

In this section, we'll demonstrate how to embed every type of Laszlo application, Flash SOLO, DHTML SOLO, and those delivered directly from an OpenLaszlo server, in an HTML file. This is the first step in eventually combining multiple executables in an HTML file.

We can't execute DHTML SOLO applications directly through a browser URL, as we can SWF files, because their LFC library has been removed. Instead, a DHTML SOLO application must be embedded in an HTML file. We'll assume that these SOLO applications are being served from a web server, so we can use SSIs (JSP includes for those served by an OpenLaszlo server) to include the header:

```
<html>
   <!-- include virtual="header.ini"-->
   <body>
      <script type="text/javascript">
         Lz.dhtmlEmbed({url: 'dhtml.lzx.js?lzr=dhtml',
                        width: '100%', height: '100%', id: 'dhtml'});
      </script>
   </body>
</html>
```

This HTML output contains the results of our Laszlo application, a simple gray box with our default label of "no label," as shown in figure 15.2.

Figure 15.2 Both the DHTML and Flash embedded applications display a gray square of varying shades.

Embedding a Flash SOLO application, flash.lzx.swf, requires that swfEmbed be called with the url parameter set to the Flash SOLO application name:

```
<html>
   <!-- include virtual="header.ini"-->
   <body>
      <script type="text/javascript">
         Lz.swfEmbed({url: 'flash.lzx.swf?lzr=swf8',
                      width: '100%', height: '100%', id: 'flash'});
      </script>
   </body>
</html>
```

Using dhtmlEmbed and updating the url parameter causes it to refer to an embedded DHTML application delivered from an OpenLaszlo server:

```
<html>
   <%@ include file="header.jsp"/>
   <body>
      <script type="text/javascript">
         Lz.dhtmlEmbed({url: 'dhtml.lzx?lzr=dhtml',
                      width: '100%', height: '100%', id: 'dhtml'});
      </script>
   </body>
</html>
```

Finally, embedding a Flash application delivered from an OpenLaszlo server is just as simple. We just change to swfEmbed and update the URL parameter for

Flash. But since there are multiple Flash executables (`swf7` and `swf8`), we'll instead update the `lzt` request type parameter to `swf` to cover them all:

```
<html>
   <%@ include file="header.jsp"/>
   <body>
      <script type="text/javascript">
         Lz.swfEmbed({url: 'flash.lzx?lzt=swf',
                   width: '100%', height: '100%', id: 'flash'});
      </script>
   <body>
</html>
```

Now that we can embed any type of Laszlo application within an HTML file, let's see how to apply these techniques to make Flash applications behave more like HTML.

15.2.4 Creating default web pages

It's common web practice to specify a URL ending with a directory path instead of naming an explicit HTML file. The HTTP server defaults to displaying either an index.html or an index.jsp file. By embedding the main.lzx file, containing the `canvas` tag, in either of these default HTML files, we can invoke our Laszlo Market application through the following URL:

```
http://localhost:8080/lps-4.1/book
```

This approach results in a cleaner URL that hides internal implementation information such as SOLO or server implementation, request type, and platform settings.

An additional benefit of serving applications from an OpenLaszlo server is that the HTML page can be updated with JSP variables. This provides an additional level of flexibility. The HTTP request can be accessed using JSP to find the value of the `contextPath` parameter. This parameter contains the context or relative file path of our request URL—in our case, this translates into the `lps-4.1` string. We don't have to hard-code the file path for our JavaScript libraries; their starting location is derived from the request URL. Now when it's necessary to switch to a different release of Laszlo, we won't have to update our header file. Listing 15.2 shows how to update the `head` section to access the `contextPath` parameter.

Listing 15.2 Using the `contextPath` parameter to configure flexibility (header.jsp)

```
<head>
   <script type="text/javascript"
      src="<%=request.getContextPath()%>" +
        "/lps/includes/embed-compressed.js">
   </script>
```

Sets the resources path

```
   <script type="text/javascript">
      lzOptions = { "<%=request.getContextPath()%>" +
                    "/lps/resources" };
   </script>
   <script type="text/javascript">
      Lz.dhtmlEmbedLFC("<%=request." +
          "getContextPath()%> lps/includes/
lfc/LFCdhtml.js");
   </script>
   <style type="text/css">
      html, body {
        height: 100%; width: 100%; margin: 0; padding: 0;
        border: 0 none; overflow: hidden;
      }
   </style>
</head>
```

Now that we can embed every application type, we can intermix any combination of Laszlo DHTML and Flash applications in a single HTML file.

15.3 Intermixing DHTML and Flash applications

Now that we have multiple Laszlo applications embedded within HTML output, they have to stake out their separate display areas. By default, embedded Laszlo applications are displayed in sequential order in HTML output and identified by an application id. When each Laszlo application is embedded, its Lz call returns an object containing the name of its application id. If an application id isn't supplied, the default name "lzapp" is used. It's considered good programming practice to always provide application ids for your embedded applications, since multiple applications using a default name will overwrite one another.

In listing 15.3, we specify an id name for each application. The height of the first application is updated to 200 pixels. HTML displays the Laszlo applications sequentially, a height of 200 pixels for the DHTML application immediately followed by the Flash application with a height of 100 pixels.

Listing 15.3 HTML that combines Laszlo DHTML and Flash applications

```
<html>
   <%@ include file="header.jsp"/>
   <body>
      <script type="text/javascript">
         Lz.dhtmlEmbed({url: 'dhtml.lzx?lzr=dhtml',
                       width: '100%', height: '200', id : 'dhtml'});
      </script>
```

```
        <script type="text/javascript">
           Lz.swfEmbed({url: 'flash.lzx?lzt=swf',
                            width: '100%', height: '100', id : 'flash'});
        </script>
     </body>
  </html>
```

The returned object is named after its application id and is accessible through Lz. It can be used in slightly different ways depending on its target platform. For Flash and DHTML applications, it provides access to the calls listed in table 15.2. In upcoming examples, we'll use these calls to access any node in an application.

Table 15.2 Lz calls to communicate with embedded applications

Name	Target	Description
callMethod	Flash	Calls a method to an embedded SWF application and returns the result
getCanvasAttribute	Both	Reads a value from a top-level attribute of the canvas element
setCanvasAttribute	Both	Writes a value to a top-level attribute of the canvas element

The setCanvasAttribute and getCanvasAttribute calls provide access to the top-level attributes in an application. When a Laszlo attribute is changed using the setCanvasAttribute call, it results in an on+attribute event being sent to registered event handlers.

callMethod, which can only be used by embedded Flash applications, navigates through the levels of Laszlo's node structure to call a method from any object. DHTML applications have direct access to the browser's DOM, with their returned object serving as the root node. As a result, a Laszlo application is an extension of the DOM tree, so we can navigate through this tree to access any node's attributes and methods.

Embedded applications have a lifecycle separate from the loading and execution of the HTML page, so we have to ensure that an application has completed its initialization before any calls are made to it. The object's onload event handler signals that an application has completed its initialization and is ready for use. In that event handler, we'll insert a function to contain the code to execute the application. Listing 15.4 shows an example of using the onload event to update the message attribute and display the correct label for each implementation.

Listing 15.4 The `onload` event, triggered when its application has completed initialization

```
<html>
    <%@ include file="header.jsp"/>
   <body>
      <script type="text/javascript">
         Lz.swfEmbed({url: 'flash.lzx?lzt=swf',
                      width: '100%', height: '100',
                      id : 'flash'});
         Lz.flashapp.onload = function() {
            Lz.flashapp.setCanvasAttribute('message',
                                 'Flash');
         }
      </script>
      <script type="text/javascript">
         Lz.dhtmlEmbed({url: 'dhtml.lzx?lzr=dhtml',
                        width: '100%', height: '100',
                        id : 'dhtml'});
         Lz.dhtmlapp.onload = function() {
            Lz.dhtmlapp.setCanvasAttribute('message',
                                 'DHTML');
         }
      </script>
   </body>
</html>
```

Sets label to Flash

Sets label to DHTML

The result, shown in figure 15.3, demonstrates that setting the `message` attribute results in an event being sent to trigger the constraint in the text element that produces an updated label.

We can also use `callMethod` for Flash and specify the path for DHTML to update these text labels. Instead of being limited to accessing only top-level variables, these calls can navigate the node hierarchy to access methods at any level. Listing 15.5 shows the different ways in which the Flash and DHTML implementations call a method.

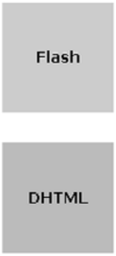


Figure 15.3 The HTML web page contains two embedded Laszlo applications; each displays a label to indicate their output area.

Listing 15.5 Equivalent ways to call a method for Flash and DHTML

```
<html>
   <%@ include file="header.jsp"/>
   <body>
      <script type="text/javascript">
         Lz.swfEmbed({url: 'flash.lzx?lzt=swf',
                        width: '100%', height: '200', id : 'flash'});
         Lz.flashapp.onload = function() {
            Lz.flashapp.callMethod
               ("main.setMessage('Flash')");
         }
      </script>
      <script type="text/javascript">
         Lz.dhtmlEmbed({url: 'dhtml.lzx?lzr=dhtml',
                        width: '100%', height: '200', id : 'dhtml'});
         Lz.dhtmlapp.onload = function() {
            Lz.dhtmlapp.canvas.setMessage
               ('DHTML');
         }
      </script>
   </body>
</html>
```

Updates Flash label

Updates DHTML label

In the next section, you'll learn how to use HTML statements to control the placement of the Laszlo applications in the HTML output.

15.3.1 Controlling Laszlo output placement in HTML

We can use `div` tags to control the placement of embedded Laszlo applications in the HTML output. A byproduct of using the `Lz` embed calls is that a `div` tag is dynamically created, unless a `div` with a matching name already exists, whose name is the `id` field plus a `Container` suffix. This `div` tag serves as a container for other tags and provides an area to display Laszlo output. Since these `div` tags are dynamically generated, they won't appear in the View Source output and can only be seen using an add-on tool such as Firebug or the DOM Inspector.

Listing 15.6. contains Laszlo DHTML and Flash applications whose output is intermixed with the output from HTML text tags.

Listing 15.6 Intermixing HTML with Laszlo output

```
<html>
   <%@ include file="header.jsp"/>
   <body>
      <h2>Flash Container</h2>
      <div id="flashContainer"></div>
      <h2>Flash Container</h2>
      <div id="dhtmlContainer"></div>
      <script type="text/javascript">
         Lz.swfEmbed({url: 'flash.lzx?lzt=swf',
                      width: '100%', height: '100', id : 'flash'});
      </script>
      <script type="text/javascript">
         Lz.dhtmlEmbed({url: 'dhtml.lzx?lzr=dhtml',
```

Contains Laszlo Flash output

Contains Laszlo DHTML output

```
                    width: '100%', height: '100', id : 'dhtml'});
      </script>
   </body>
</html>
```

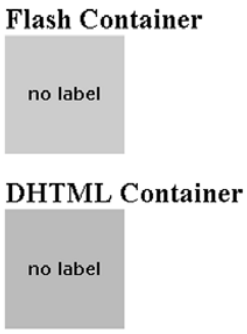


Figure 5.4 HTML text is used to label Laszlo output.

We can control the placement of our Laszlo output in the HTML page by defining our own div tags. As figure 15.4 shows, this allows the Laszlo output to be placed immediately following each of the HTML labels.

There are also situations where we want our application output to be redirected to a different div tag. We can do this by adding an appenddivid argument containing the target div name to the input string. Listing 15.7 shows an appenddivid argument that directs the application output to the designated div area. This produces the same output as seen in figure 15.4, but these div tags now have our specified names.

Listing 15.7 Redirecting Laszlo to specific HTML divs

```
<html>
   <%@ include file="header.jsp"/>
   <body>
      <div id="flashapp">
         <h2>Flash Container</h2>
      </div>
      <div id="dhtmlapp">
         <h2>DHTML Container</h2>
      </div>
      <script type="text/javascript">
         Lz.swfEmbed({
            url: 'flash.lzx?lzt=swf',
            width: '100%', height: '200',
            id: 'flash', appenddivid: 'flashapp'});
      </script>
      <script type="text/javascript">
         Lz.dhtmlEmbed({
            url: 'dhtml.lzx.js',
            width: '100%', height: '200',
            id: 'dhtm', appenddivid: 'dhtmlapp'});
      </script>
    </body>
</html>
```

Directs output to first div

Directs output to second div

You'll now learn how to use `div` tags to make applications accessible to search engines like Google.

15.3.2 Building a search engine–accessible application

The key to bridging the mismatch between search engines, oriented to indexing HTML pages, and the continuous user experience of RIAs is to supplement an RIA with a cluster of supporting HTML pages. Search engines require a different page for each product, so for the Laszlo Market we'll provide a special "home page" of the most popular products for them. Because web crawlers don't understand JavaScript, Flash, or complex DHTML tags, we'll create a simplified HTML version.

Let's start by removing the `noscript` tag from our header.jsp file to allow a web crawler to scan the file. This page will also include an embedded application that displays a specific product. We'll use `LzBrowser.getInitArg` to pass in a product ID and set the application's initial state to display that product. We'll simultaneously support a default HTML display of the selected product information for JavaScript-impaired users and web crawlers, as well as the Laszlo Market, which displays the selected product for JavaScript-enabled users.

To accomplish both these displays, we'll store all product information in a `div` whose name matches the `appendivid id` for our embedded application. Now, when JavaScript is enabled, our embedded application will be executed and the HTML will display the Laszlo application's output. For users whose JavaScript is turned off, only the default HTML in the `div` tag is displayed. Listing 15.8 shows the HTML content for a sample product. We've used the ellipsis to indicate continuing description and specs information.

Listing 15.8 Creating the HTML/Laszlo web page for web crawlers

```
<html>
   <%@ include file="header.jsp"/>
   <body>
      <div id="productcontent">
         <h1>Spider Man 2</h1>
         <div id="price">$19.99</div>
         <div id="description">Peter Parker is having a rough…</div>
         <div id="specs"><p>Regional Code: 2 (Japan,…</p></div>
      </div>
      <script type="text/javascript">
         Lz.swfEmbed({ url: 'main.lzx?lzt=swf&productid=SKU-001',
                       width: '100%', height: '100%',
         appenddivid: 'productcontent'});
      </script>
   </body>
</html>
```

Provides HTML for web crawlers

Replaces HTML content with Laszlo Market

Supporting this new feature in the Laszlo Market is easy. We just need to check for the existence of a URL productid parameter in the init method for the canvas. If it exists, then we'll update the state controller attribute to the Splash to Display Product state transition. To simplify the presentation, we'll ignore security issues relating to login, and as listing 15.9 shows, we'll just immediately move to that state.

Listing 15.9 Updated the state controller (gController.lzx)

```
<canvas>
   <attribute name="productid" type="string"/>
   <handler name="onproductid">
      gController.setAttribute("appstate", "Login to Main");
      logwin.close();
   </handler>
   ...
   <method name="init">
      var sku = LzBrowser.getInitArg('productid');
      if (sku != null)
         canvas.setAttribute(
            'productid', sku);
                           ...
   </method>
   ...
</canvas>
```

Updates state controller

The Splash to Display Product state transition displays the main screen with the appropriate product information. We use the showProduct method in listing 15.10 to find the corresponding product from the dsProducts dataset and then update the productdp data pointer to update the Product Display window with this product information.

Listing 15.10 Update state controller to display products (gController.lzx)

```
<node name="gController">
   <handler name="onappstate" args="state">
      switch (state) {
         case "Display Product ID":
            this.showProduct(canvas.productid);
            this.setAttribute("currstate", "Main");
         break;
      ...
   </handler>

   <method name="showProduct" args="sku">
      if (sku == null) return;
```

```
         var plist =
            dsProducts.getPointer().xpathQuery
            ("dsProducts:/products/
            product[@sku='" + sku + "']");
         if (plist) productdp.setPointer(plist);
      </method>
      ...
   </node>
```

Finds product in dataset

Updates Product Details window

The techniques outlined in this section allow the contents of Flash-based applications to be searched by web crawlers and listed by search engines.

We now move on to complete the browser/embedded application relationship.

15.4 Calling browser JavaScript from Laszlo

In the previous section, we covered communication from the browser to embedded applications. Now it's time to complete this relationship with the reverse, communicating from Laszlo to the browser. Laszlo provides the `LzBrowser` service to access a browser or Flash environment. Table 15.3 provides the complete list of `LzBrowser` methods.

Table 15.3 `LzBrowser` methods

Name	Target	Description
`callJS`	Browser	Runs a JavaScript method in the browser; returns a value
`getInitArg`	Browser	Returns a key value from the initiating request string
`getLoadURL`	Browser	Returns the URL from which the application was loaded
`getVersion`	Flash	Gets the version number of the Flash player
`loadJS`	Browser	Runs a JavaScript method in the browser; doesn't return a value
`loadURL`	Browser	Loads a URL in the browser
`setClipboard`	Flash	Sets the system clipboard to the specified string
`urlEscape`	Browser	Escapes a string using URL encoding
`urlUnescape`	Browser	Unescapes a string using URL encoding
`xmlEscape`	Browser	Escapes XML special characters such as `&` and `<`

Laszlo provides two methods, `loadJS` and `callJS`, to call browser JavaScript from within an application. The `loadJS` method easily makes calls to JavaScript functions, but has the limitation of not returning a value. Here's an example of `loadJS` invoking the JavaScript alert function:

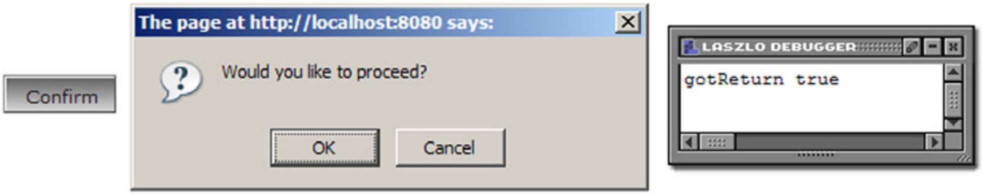
```
<canvas>
   <button>Alert
      <handler name="onclick">
         LzBrowser.loadJS("alert('Hello!')");
      </handler>
   </button>
</canvas>
```

Although the `callJS` method requires more setup, it is more capable since it can return a value. This method takes two or more arguments: the name of the browser JavaScript function to call, a reference to the LZX method to receive the return value, and any arguments that need to be passed to the JavaScript function.

We can use `callJS` to invoke the `confirm` browser function, but it can also be used to call user-defined functions. This browser function displays a dialog box that contains our argument string and, after executing it, returns a boolean indicating the user's selection:

```
<canvas debug="true">
   <button>Confirm
      <handler name="onclick">
         LzBrowser.callJS('confirm', this.gotReturn,
                          'Would you like to proceed?');
      </handler>
      <method name="gotReturn" args="r">
         Debug.write('gotReturn', r);
      </method>
   </button>
</canvas>
```

Figure 15.5 shows the familiar browser dialog box that appears when a Laszlo application calls a browser JavaScript function.



The page at http://localhost:8080 says:

Would you like to proceed?

```
gotReturn true
```

Figure 15.5 In this example, invoking a browser-level JavaScript dialog box that returns a value, we have shown the returned debug value after the dialog window's OK button has been clicked.

Although a Laszlo DHTML application can directly call browser JavaScript functions and directly access variables, this practice is strongly discouraged since the result of doing so is an implementation-specific application. In the next section, we'll look at situations where implementation-specific features are needed and show you how to access them.

15.5 Calling Flash from Laszlo

Just as Laszlo can execute browser JavaScript functions, it can also execute Flash functionality. This allows a Laszlo application to be augmented with Flash-specific functionality, such as accessing the system clipboard or calling any primitive Flash ActionScript object.

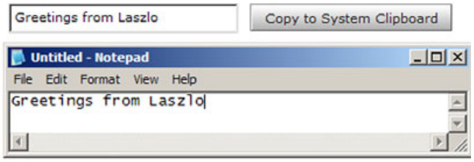
15.5.1 Using Flash to set the system clipboard

Flash can store and retrieve character strings to the system clipboard, using the `setClipboard` method, for all major operating systems. Text saved to the clipboard is thereby accessible to desktop applications. There is no comparable way to do this using DHTML:

```
<canvas>
   <edittext name="input" width="200"/>
   <button x="210">Copy to System Clipboard
      <handler name="onclick">
         LzBrowser.setClipboard(parent.input.getValue());
      </handler>
   </button>
</canvas>
```

Figure 15.6 shows how the user's input value is retrieved from an input field, passed to the system clipboard, and finally accessed by the Notepad application.

For those situations where Laszlo hasn't already created a method or tag for a desired Flash capability, Laszlo allows the Flash ActionScript object to be called directly.


Greetings from Laszlo

Greetings from Laszlo

Figure 15.6 Laszlo Flash supports the `setClipboard` method, thus allowing images or text to be copied.

15.5.2 Accessing Flash ActionScript objects

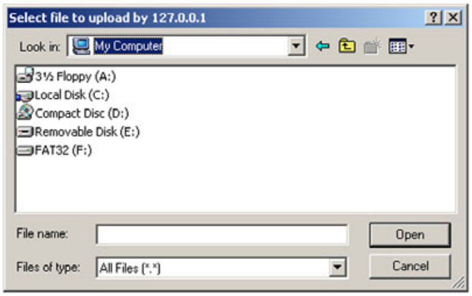
Laszlo's tags and methods are designed to limit the need to access implementation-specific libraries. But for those cases when a Flash-specific feature is necessary, Flash

ActionScript objects supported by a particular release (Flash 7 or Flash 8) can be called directly. The following example demonstrates how a Flash Action object can be instantiated in Laszlo to display a Select File to Upload menu:

```
<canvas>
   <script>
      var fr;
      function getReference() {
         if (global['fr'] == undefined) {
             fr = new flash.net.FileReference();
         }
         return fr;
      }
   </script>
   <button text="Select File" onclick="(getReference()).browse()"/>
</canvas>
```

We invoke the browse function from the Flash ActionScript `flash.net.FileReference` object to produce the Select File to Upload dialog box shown in figure 15.7.

Although additional work is required on both the server and client side to get file uploading to work, we've provided a client-side window for selecting this file.



Look in: My Computer

Files of type: All Files (*.*)

Figure 15.7 This client-side file-selection window is produced by a native Flash call.

15.6 Embedding HTML in Laszlo

The final step to complete the functionality provided by a shell wrapper is to have a Laszlo application complete the circuit by embedding an HTML web page. Laszlo's view-based `html` object provides HTML-based content to a parent view, which can be used in both Laszlo Flash and DHTML applications. While it's intended to be used in a nested visible object, its operation is the reverse of other view-based objects. Instead of causing its parent to size itself to its physical dimensions, the `html` object adopts its parent's dimensions. This new usage requires some new attributes, listed in table 15.4, to describe this relationship.

Table 15.4 `html` object attributes

Name	Data Type	Attribute	Description
`heightoffset`	`number`	Setter	Trims an HTML image's height to fit within a view
`iframe`	`string`	Read-only	Name of the containing `iframe`

Table 15.4 `html` object attributes *(continued)*

Name	Data Type	Attribute	Description
`loading`	`boolean`	Read-only	States whether the HTML application is still loading
`src`	`string`	Setter	Specifies the URL to be loaded
`target`	`LzNode`	Setter	Name of the parent view
`visible`	`boolean`	Setter	Controls the visibility of the HTML display
`widthoffset`	`number`	Setter	Trims an HTML image's height to fit within a view
`xoffset`	`number`	Setter	Sets the horizontal offset of the HTML image
`yoffset`	`number`	Setter	Sets the vertical offset of the HTML image

The reason for this unusual relationship is that, unlike regular nested views, where the child initializes first, the `html` object needs its parent to initialize first. The parent must be completely initialized before it can set its `src` attribute. This attribute setting is the initial URL for the `html` object and it can't complete its initialization without it.

We can display a Google web page in a Laszlo application. It will be contained in a window component that features automatic scrolling both horizontally and vertically. We use the `xoffset` and `yoffset` attributes to maneuver the HTML image directly over the window, and the `widthoffset` and `heightoffset` attributes to trim the HTML image in the window. When the HTML image is correctly sized using these settings, the window's scrollbar operates correctly:

```
<canvas>
   <window width="400" height="400"
           oninit="this.win.setSrc('http://www.google.com/')">
      <html xoffset="8" yoffset="25" widthoffset="-20"
            heightoffset="-45"              name="win"
            oninit="this.bringToFront()"/>
   </window>
</canvas>
```

Another consequence of not being nested in the normal way is the fact that the `bringToFront` method must be used to place the HTML image in front of the window. Table 15.5 contains the complete set of `html` methods.

Table 15.5 `html` object methods

Name	Description
`bringToFront`	Brings the HTML view to the front
`sendToBack`	Sends the HTML view to the back

Table 15.5 `html` object methods *(continued)*

Name	Description
`setSrc`	Sets the URL source
`setTarget`	Sets the parent node
`setVisible`	Changes the visibility of the HTML view

While a DHTML application can use the default HTML wrapper, a Flash application requires some additional customization. We need to specify its `LzOptions` setting to set its `wmode` parameter to transparent. The `wmode` parameter is an internal Flash setting that allows the Flash content to be layered with DHTML. We create another header file especially for Flash applications using the `html` object:

```
<head>
   <script type="text/javascript">
           lzOptions = { "wmode: transparent" };
   </script>
   ...
</head>
```

Now our embedded Flash application produces the "live" web page seen in figure 15.8. This isn't simply a static display but is a fully operational web page. If you click any of its links, a new web page appears in its place.

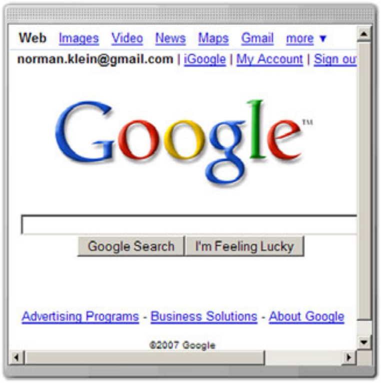

Web Images Video News Maps Gmail more ▾
norman.klein@gmail.com | iGoogle | My Account | Sign ou
Google™
Google Search
I'm Feeling Lucky
Advertising Programs - Business Solutions - About Google
©2007 Google

Figure 15.8 A live HTML web page is displayed in a window.

Of course, there is nothing to prevent you from displaying web pages that contain embedded Laszlo applications; you just need to be careful not to set up an endless loop!

Now that we've replicated the functionality of a shell wrapper in Laszlo, it's time to put these tools to use by embellishing DHTML applications with the audio and video capabilities found only in Flash. This provides developers with the best of both worlds: taking advantage of the open standard compliance of DHTML, while customizing with the hottest features from Flash. We'll demonstrate these techniques by completing the Media Player window for the Laszlo Market and supporting it under different combinations of embedded application types.

But first we'll explore how Laszlo works with video.

15.7 Working with video

Laszlo currently supports Flash FLV (Flash Video), a binary file format delivering bit-mapped video to a Flash player. This allows Flash to execute SWF files to display animated vector graphics and FLV files for bit-mapped video. When an FLV file is displayed, these two file formats are complementary—the SWF file controls the display of FLV files. Although an FLV file is limited to a single video, several FLV connections can be simultaneously opened. Flash provides multiple ways to deliver video:

- Embedded video (requires SWF)
- Progressive download (requires SWF and FLV)
- Streaming video (requires SWF and FLV)

SWF and FLV files can be used to deliver both video and audio (MP3). The calling methodology is the same; the only difference is that the content is audio instead of video.

15.7.1 Using streaming media

With streaming media, audio and video data is delivered in real time to support "live" broadcasting. But streaming offers much more than just broadcasting services. Unlike HTML, it provides a persistent connection to support bidirectional service, transferring both media content and control messages. Users can transmit audio and video data as well as receive it and interact as equal participants, instead of simply being passive viewers.

The player can send control messages to the streaming server to have its data rate adjusted to account for changing network conditions. If a user's network connection becomes congested, a streamed webcast can downgrade the quality to still allow a continuous viewing experience. Control messages can also be sent to dynamically access any frame within an FLV file.

Streaming is a property of the delivery system rather than of the media. A persistent connection is required to support streaming, so the HTTP protocol can't be used. A persistent connection also relieves the need for a server to maintain a session. Since these requirements are beyond the capabilities of HTTP, another protocol is needed along with special server software. Various companies support their own proprietary streaming media file formats: Adobe Flash, Apple QuickTime, and Microsoft's Window Media, among others. Of course, none of these file formats or protocols are compatible with one another, and they all require a player plug-in to be installed. Each of these companies also sells its own streaming media server. Since Laszlo currently only supports Flash, we'll default to using FLV.

Flash uses the Real-Time Messaging Protocol (RTMP) with the Flash Communication Server. Although RTMP is proprietary, Adobe has slowly begun to make its core Flash platform protocols and file formats available to open-source groups to encourage a larger audience. One of these is the Red5 project, which provides an open source version of Adobe's Flash server. We'll use the Red5 streaming server to provide streaming video for our Laszlo Market application.

15.7.2 Using the Red5 server

Red5 is an open source project providing an RTMP server to support real-time streaming data transfer services for Flash players. Red5 eventually will support all the major features available from the Adobe Flash Communication Server. You can download Red5 at www.osflash.org/red5. While it is still a few releases away from a 1.0 version, it already provides the fundamental services.

Since Red5 is a Java-based server-side application, you can install it as easily as you would a web server. Red5 listens for TCP connections on port 1935, which means it can reside on the same hardware as a web server. Before it can download videos, a Laszlo application first needs to establish a persistent RTMP connection with the Red5 server. Once this connection has been established, a Laszlo application can begin using Red5's downloading services.

Accessing Red5 server through RTMP

A Red5 server request is made through an RTMP URL. This has the familiar form of an HTTP URL, except that the RTMP protocol is used:

```
rtmp://host:port/app
```

Although oflaDemo is referred to as an application, it specifies this directory:

```
$RED5_HOME/webapps/oflaDemo/streams
```

which contains the collection of video files.

In this request, app specifies the application; Red5's default port setting is 1935. Now we'll access some demo FLV videos supplied with Red5 that are contained in the oflaDemo application through this RTMP URL:

```
rtmp://localhost:1935/oflaDemo/Spiderman3.flv
```

We can't simply enter this URL into the browser window to display the Spiderman 3 movie trailer; instead, we need a Flash player to initiate the connection. In the next section, we'll construct our Laszlo application that displays this movie trailer.

15.7.3 Interfacing Laszlo to a Red5 server

Laszlo has created a number of classes to simplify access to streaming media. These classes are contained in the $LASZLO_HOME/lps/components/extensions/av directory and supply the fundamental interfaces to the Red5 streaming media server. Before you can use any of the other classes, you must first establish a connection using the `rtmpconnection` class. The attributes for this class are listed in table 15.6.

Table 15.6 `rtmpconnection` attributes

Name	Data Type	Attribute	Description
`src`	`string`	Setter	Application URL.
`autoconnect`	`boolean`	Setter	Connect automatically during initialization. If false, you need to explicitly call `connect()`. The default is true.
`status`	`string`	Read-only	String that indicates connection status.
`stage`	`number`	Read-only	Number that indicates the current connection stage—`0`: disconnected, `1`: connecting, `2`: connected.

We only need to set the `src` and `autoconnect` attributes to display a video. We'll also add the `rtmpstatus` component to provide status information telling us whether a successful connection was established with the Red5 server. This component displays a "status light" showing red, yellow, or green to indicate the connection state:

```
<canvas>
   <rtmpconnection src="rtmp://localhost:1935/oflaDemo"
                   autoconnect="true"/>
   <rtmpstatus/>
</canvas>
```

When the `rtmpstatus` component displays a green light (indicating it has successfully connected to the Red5 server), we're ready to access our demo video.

The `videoview` can be used to show a video:

```
<canvas>
   <rtmpconnection src="rtmp://localhost:1935/oflaDemo"
                   autoconnect="true"/>
   <videoview url="Spiderman3.flv" type="rtmp" autoplay="true"/>
</canvas>
```

Although this creates a simple video display, useful for many applications, most people expect a minimum video interface that provides stop, rewind, fast-forward, and volume controls. Laszlo's `videoplayer` class supplies these functions:

```
<canvas>
   <include href="av"/>
   <rtmpconnection src="rtmp://localhost:1935/oflaDemo"
                   autoconnect="true"/>
   <videoplayer height="300" width="400"
                url="Spiderman3_trailer_300.flv" type="rtmp"
                autoplay="true"/>
</canvas>
```

Since the autoplay attribute is set to true, the videoplayer immediately begins playing the Spiderman 3 trailer. We can also use the url attribute to trigger the playing of a video. Figure 15.9 shows the video player with its control interface. There is a Pause button to momentarily stop playback and a knob that supports fast-forward and rewind. Additionally, the user can control the volume through the audio icon in the right corner.

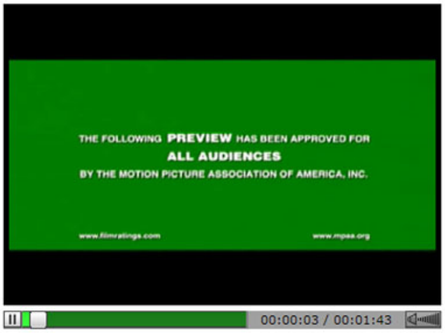

THE FOLLOWING PREVIEW HAS BEEN APPROVED FOR
ALL AUDIENCES
BY THE MOTION PICTURE ASSOCIATION OF AMERICA, INC.
www.filmratings.com
www.mpaa.org
00:00:03 / 00:01:43

Figure 15.9 A video player is shown with its control interface.

Now that you know how to display Flash video in Laszlo, let's add these video capabilities to our Laszlo Market application so that the video is accessible from both Laszlo DHTML- and Flash-based implementations.

15.7.4 Adding video to the Laszlo Market

At first glance, making Flash-based services accessible to a DHTML-based application would appear to require highly specialized surgical procedures. But by building on the techniques described earlier in this chapter, adding Flash-based services—in this case, video—turns out to be remarkably easy.

The Laszlo Market requires a mediaplayer class to display a customer's video when it is dragged and dropped into the Media Player window. For a Flash implementation, we can build this functionality directly into the application. However, a DHTML implementation requires a separate Laszlo Flash application, which we'll call mediaplay.lzx, to contain this video functionality. We'll use an HTML wrapper to bundle the DHTML and Flash applications together into a seamless whole. We can add Flash video to the Laszlo Market by including the mediaplayer:

```
<canvas>
    <include href="lzmodules/mediaplayer.lzx"/>
    <mediaplayer width="100%" height="100%"/>
</canvas>
```

The mediaplayer is divided into Flash and DHTML sections. Because video functionality is only available in Flash, the Flash segment implements the video player. A DHTML application embeds the output from the Flash-based video player using the html object. Let's start by looking at the Flash portion, shown in listing 15.11.

The mediaplayer must be able to operate in two different modes: when used in a Laszlo Flash application and also when embedded in Laszlo DHTML. The Laszlo Flash application only implements mouse-tracking functionality to drop videos onto the player. When it is embedded in Laszlo DHTML, the DHTML handles all mouse tracking. It registers itself in the media_target tracking group and only needs to handle the onmousetrackup event to load the video url.

As listing 15.11 shows, the videoplayer listens for the canvas.playurl attribute, which specifies the name of the requested video.

Listing 15.11 Laszlo Flash section containing video player

```
<library>
    <switch>
        <when runtime="dhtml">
            <class name="mediaplayer">
                ...
            </class>
        </when>
       </when>
       <otherwise>
            <include href="av"/>
            <attribute name="playurl" value="" type="text"/>
            <class name="mediaplayer">
                <videoplayer id="vid"
                        width="$once{classroot.width-10}"
                        height="$once{classroot.height}">
                    <method name="playurl" args="url">
                        if (url == null) return;
                        this.setAttribute('url', url);
                    </method>
                    <handler name="oninit">
                        LzTrack.register(this, "media_target");
                    </handler>
                    <handler name="onmousetrackup" args="t">
                        var d = dragger.data;
                        var url = d.attributes.video;
                        canvas.setAttribute(
                           'playurl', url);
                    </handler>
                    <handler name="onplayurl" args="l" reference="canvas">
                        this.playurl(l);
                    </handler>
                </videoplayer>
```

Displays videoplayer

Sets video title

Loads video

```
                </class>
            </otherwise>
        </switch>
    </library>
```

Now we'll take a look at the DHTML side in listing 15.12, which needs to perform all the mouse tracking. Since there are multiple applications, we'll also need to control the z-axis placement of these applications; this determines which application receives keyboard and mouse input. The Laszlo DHTML application will generally be the frontmost application, but the Laszlo Flash application should move to the front when a user needs access to the control buttons. This requires that both local and tracking-group mouse events be handled. Now the control buttons are accessible when an onmouseover event occurs and no tracking is occurring.

Listing 15.12 Laszlo DHTML section embedding a video player

```
<library>
    <switch>
        <when runtime="dhtml">
            <class name="mediaplayer">
                <view id="vid" width="100%" height="100%">
                    <attribute name="url" type="text" value=""/>
                    <handler name="onurl" args="u">
                        this.playurl(u);
                    </handler>
                    <method name="playurl" args="url">
                        if (url == null) return;
                        Lz.mediaplayer.
                         setCanvasAttribute
                           ('playurl', url);      <- Sets video for videoplayer
                    </method>
                    <handler name="oninit">
                        LzTrack.register(this, "media_target");
                        parent.videoplayer.sendToBack();
                    </handler>
                    <handler name="onmousetrackover">
                        this.tracking = true;
                        parent.videoplayer.sendToBack();
                    </handler>
                    <handler name="onmousetrackout">
                        this.tracking = false;
                    </handler>
                    <handler name="onmousetrackup">
                        this.tracking = false;
                        var d = dragger.data;
                        var video_url = d.attributes.video;
                        vid.playVideo(video_url);
                    </handler>
```

Handles dragging operations in DHTML

```
                    <method event="onmouseover">
                        if (this.tracking != true)
                            parent.videoplayer.
                             bringToFront();
                    </method>
                    <method event="onmouseover"
                            reference="LzGlobalMouse" args="e">
                        if (e == this) return;
                        parent.videoplayer.sendToBack();
                    </method>
                </view>
                <html name="videoplayer"
                      visible="${gController.currstate == 'Main'}"
                      src="mediaplayer.jsp"/>
            </class>
       </when>
       <otherwise>
          ...
       </otherwise>
    </switch>
</library>
```

Brings videoplayer to front to access playback controls

Displays video output

The display of the video player is provided by the `html` tag. Its display is produced from the HTML output returned by mediaplayer.jsp. This file also communicates the video name from DHTML to Flash. Because the video player is a complete Laszlo application, we can set attributes on its canvas to pass the video name from DHTML to Flash. Use the following code to configure the embedded Laszlo Flash application so it can be easily referenced within the DHTML application:

```
<html>
   <%@ include file="header.jsp"/>
   <body>
      <script type="text/javascript">
         Lz.swfEmbed({url: 'mediaplayer.lzx?lzt=swf',
                      width: '100%',
                      height: '100%', id: 'mediaplayer',
                      wmode: 'transparent'});
         window.parent.Lz.mediaplayer = Lz.mediaplayer;
      </script>
   </body>
</html>
```

Because we are working in DHTML, we can directly access the browser's DOM. This allows us to store a reference to the embedded `mediaplayer` in the parent frame. Now the embedded Flash video player can be easily accessed by the Laszlo DHTML application through `Lz.mediaplayer`. We can then use `setCanvasAttribute` to pass the video name from DHTML to the embedded video player.

Congratulations, we've now added video-playing capabilities to both our Laszlo Flash and DHTML applications. We achieved this so seamlessly that it is now difficult to distinguish one application from the other. This frees developers to develop and deploy applications on their preferred target platform.

15.8 Summary

You don't have to choose between the DHTML and Flash platforms—Laszlo lets you develop hybrid applications that combine the best features from both platforms. Developers who prefer the open source environment provided by DHTML can now supplement its functionality with proprietary Flash features on an as-needed basis. This allows them to keep the majority of their application in DHTML and isolate these features until future open source solutions are available. Developers who prefer working with Flash benefit since their applications can now be listed in Google and other popular search engines to attract a larger audience.

We've now concluded all client-side development that is possible without server-side support. The remaining chapters will address server-related issues. At this point, we've created a full-featured prototype application that allows all client-side functionality to be exercised. The advantage of this approach is that client-side development is completely isolated from server development. In this way, developers can proceed independently without having to deal with coordination issues between various development groups. This dramatically simplifies project management, since issues in one group don't impact the other.

Part 5

Server and optimization issues

The previous parts of this book promoted postponing server-side development by relying on resident datasets for early testing. This approach severs dependencies between the front-end and server-side development groups. Consequently, each group can work independently toward a common API, which serves as the bridge between their efforts. Although some development groups might find it more productive to work jointly in early development, that choice should be an option and not imposed by the working environment.

Our next objective is to make the transition to an integrated development effort as seamless as possible. Chapter 16 provides a methodical approach to update resident datasets into HTTP-supported datasets. Once we have access to large server-supported databases, we'll need to worry about performance optimization issues. The Laszlo Market will be updated to support sessioned data—that is, to maintain shopping cart contents between sessions. Chapter 17 demonstrates methods for managing large datasets in Laszlo to ensure that resources aren't wasted. In particular, the Market will be updated to display large amounts of data using paged datasets. Chapter 18 deals with optimizing an application's startup time with dynamic libraries and redistribution of initialization costs over time.

16 Networked data sources

This chapter covers

- Converting resident to HTTP datasets
- Building data services
- Building a sessioned application
- Maintaining server domains

Be silent as to services you have rendered, but speak of favors you have received.

—Seneca, Roman philosopher

We've arrived at a point where the major Laszlo design issues have been resolved and further development requires access to a networked data source. Up to this point, we've used resident datasets to supply all of our data-related needs. Although networked data could have been introduced earlier, we've purposely postponed it. Decoupling client and back-end development allows each to proceed independently, an approach that has a number of advantages. We enjoyed unlimited freedom to experiment with interface development without support from a server development group. Our client development front end was also isolated from any possible back-end problems, which simplified and sped up development. Finally, although we didn't use this capability, this approach allows extensive unit testing early in development, which can be reused for the server implementation.

Although local datasets initially sped up our development, they are constrained by capacity and nonpermanent persistence. To remove these constraints, it's necessary to work with networked resources. Unfortunately, this transition complicates processing. Since our data resources are no longer compiled into the application, we can't be assured that our data is always available. We must now deal with the fallibility issues of networked data, including server failure and congestion issues that result in error and timeout conditions.

This transition must be as effortless and seamless as possible. In chapters 8 and 12, we used local datasets to establish the XML data structure for the `dsProducts` and `dsCart` datasets. This XML data structure was used as a model for creating our data path's XPath expressions. Now, we must ensure that data returned by the server has an identical XML structure, as any differences will cause these expressions to return incorrect matches. We want to avoid the pain of debugging broken XPath expressions. The best approach is to add unit testing to ensure compliance.

In this chapter, we'll demonstrate how to create supporting classes that allow an easy transition from local to networked data. These classes can be added to our existing application with minimal code changes. While it isn't quite as easy as flipping a switch, this transition is easily applied to most applications with minimal pain and heartache.

16.1 Interfacing to web servers

To interface to a web server, Laszlo requires only an XML-over-HTTP service, which returns an XML document for a resident dataset. XML-over-HTTP is supported by most servers.

The equivalence between resident- and HTTP-server-supplied data makes comparison easy. If an HTTP-supplied dataset matches a resident datset that was used in early development, then the data path XPath statements in the application remain valid. But before we can start the transition from resident to HTTP datasets, we need to review XML-over-HTTP and dataset operation.

The HTTP standard specifies that all HTTP servers return a response whose content type can be set to `text/xml`. This allows Laszlo to interface to any HTTP web server—ranging from basic HTTP web servers such as Apache, Jetty, or Microsoft's Internet Information Services (IIS) to the various web frameworks such as Struts, Tapestry, Ruby on Rails, and Microsoft .NET—that work with servlet containers to provide enterprise-class services.

There are two approaches to providing XML-over-HTTP web services: they can be either activity or resource oriented. An *activity-oriented* approach views the Internet in terms of available services. A popular example of this approach is Simple Object Access Protocol (SOAP). The *resource-oriented* approach, also known as Representational State Transfer (REST), takes the opposite approach and views the Internet in terms of resources. These REST resources can be manipulated with standard HTTP commands: POST, GET, PUT, and DELETE. We've decided to use the REST approach in this book, since it's an extension of existing best practices and provides the simplest way to interface Laszlo to an HTTP server.

A good example of a REST-based web service is Really Simple **S**yndication (RSS). RSS newsfeeds are a familiar feature on the Web as they aggregate syndicated web content such as news reports into a list of headlines. These aggregated headlines provide coverage to almost any topic of interest. RSS output is a dialect of XML, which can be retrieved through a standard URL. For example, to access the top stories from Yahoo!, we'd type in the following URL:

```
http://rss.news.yahoo.com/rss/topstories
```

It will return an XML document whose data composition is defined by the RSS 2.0 specification:

```
<?xml version="1.0" encoding="iso-8859-1" ?>
<rss version="2.0" xmlns:media="http://search.yahoo.com/mrss/">
   <channel>
      <title>Yahoo! News: Top Stories</title>
```

```
        <item>
            <title>Populations of 20 common birds
                   declining (AP)</title>
            <link>
  http://us.rd.yahoo.com/dailynews/rss/topstories
            </link>
            ...
        </item>
        <item>
          ...
        </item>
  </rss>
```

This ensures that the XML document delivered by any RSS 2.0–compliant server has a compliant composition.

We'll start by demonstrating how a dataset can be used to directly retrieve data from a short list of RSS websites and then examine the limits of this approach. Next, we'll see the benefits of using buffered datasets. Finally, we'll implement a *framework* that can be used across different datasets to easily generate and send HTTP requests that automatically invoke its response handler methods. By the end of this chapter, we'll have a general approach for handling HTTP data that is easily extended to work with any request.

16.1.1 Using datasets with HTTP

Before you can make the transition from static resident to dynamic HTTP datasets, you need a good grasp of dataset properties, particularly those concerned with HTTP. These properties are needed to deal with the dynamic timing involved in loading data from an HTTP server. With that understanding, we'll start with the simplest type of HTTP connection, interfacing to an RSS server.

A dataset can store local data compiled into an application or HTTP data returned by a web server. From Laszlo's viewpoint, data is data; its origin is immaterial.

An application can have any number of datasets. While datasets support every configuration feature in the HTTP standard, we'll cover only their most important features.

A dataset is an unusual Laszlo object since it is descended from multiple parents. It's derived from the LzNode object, which means it can be used as a declarative tag, as well as from the LzDataElement object (to gain access to its data-access methods). A dataset is flexible enough to supply a simple interface for accessing local data while also supporting all the optional settings defined by the HTTP standard. A dataset's `src` attribute, which determines the location of the data, can have these values:

- `none`: Data is contained in the dataset.
- `pathname`: A local XML file is compiled into the application.
- `url`: An absolute or relative URL points to HTTP-based data.

A dataset's operating mode is based on its `src` attribute. If the attribute is omitted or set to a local file, then it's a local dataset; when set to a URL, it interfaces to HTTP-based data. This allows an application's development cycle to easily transition from local test data to networking with an HTTP server.

By default, Laszlo sends HTTP GET requests, but a dataset's `setQueryType` method can be used to change this. Requests with a large number of query parameters should be sent with HTTP POST to ensure that data is sent as part of the request body, since some servers have limits on the size of a GET query string. Tables 16.1 and 16.2 list some of the most commonly used dataset attributes and methods. To see the complete listing of attributes and methods for a dataset, view the latest reference manual at the Laszlo website at http://www.Laszlo.org/lps4/docs/reference.

Table 16.1 Commonly used dataset attributes

Name	Data Type	Attribute	Default	Description
`querystring`	`string`	Read-only		A string appended to a dataset request.
`request`	`boolean`	Setter		When true, the dataset makes a request when it begins the init stage.
`secure`	`boolean`	Setter	`false`	Specifies whether or not the app-LPS connection is secure.
`secureport`	`number`	Setter	`443`	The port number to use to connect to the LPS for a secure connection.
`Src`	`string`	Setter		The source for requests made by this dataset.
`timeout`	`number`	Setter	`30000`	The timeout period in milliseconds for load requests.
`type`	`string`	Setter		If set to `http`, the dataset interprets its `src` attribute as a URL from which to load its content, rather than a static XML file to inline.

Table 16.2 Commonly used dataset methods

Name	Description
`abort()`	Stops loading the dataset's current request.
`doRequest()`	Issues a request immediately using the current values; if `autorequest` is true, this method is called automatically when values change.
`getResponseHeader(name)*`	Returns the value for the specified response header, or false if there is no header with that name; if `name` is omitted, all response headers are returned as an object of name/value pairs.
`getSrc()`	Returns the `src` attribute of the dataset.
`setHeader(key, val)*`	Sets a header for the next request.
`setQueryParam(key, val)`	Sets a named query parameter to the given value.
`setQueryParams(assoc array)`	Sets multiple query parameters using the keys in the argument as keys and the values of those keys as values.
`setQueryString(string)`	Sets the `querystring` parameter of the dataset to the given string.
`setQueryType(reqtype)`	Sets the query type for the parent data source to either POST or GET by calling the method of the same name on this dataset's data source.
`setRequest(boolean)`	Specifies whether or not the dataset makes its request on initialization.
`setSrc(src)`	Sets the `src` attribute of the dataset's parent data source.

*Subject to platform capabilities (not available on Flash unless in proxy mode)

To ensure that an application is initially populated with data, the `request` attribute for its datasets should be set to true. This sends a series of initial HTTP requests to populate each dataset when application initialization completes. An HTTP dataset can be manually populated by calling its `doRequest` method.

We'll now use the attributes and methods listed in tables 16.1 and 16.2 to demonstrate how easy it is to implement an RSS newsfeed with Laszlo.

Implementing an RSS newsfeed

Data paths and datasets make accessing RSS information simple. Listing 16.1 sets the `request` attribute for the `newsfeed` dataset, so it's initialized with data to create a headline listing display. Since the data contained in these feeds is RSS-compliant, their data composition can be expressed with a single set of data path expressions. We can change the src attribute of our dataset to point to any other RSS 2.0–compliant newsfeed with the assurance that our data path expressions still work correctly. In our example, we'll toggle between the CNN and Yahoo! newsfeeds by clicking the Change button. This example could easily be expanded to include additional selections.

Listing 16.1 Accessing RSS newsfeeds

```
<canvas>
   <dataset name="newsfeed" request="true"
            src="http://rss.news.yahoo.com/rss/topstories"/>
   <button y="50" text="Change">
      <attribute name="cnt" value="1" type="number"/>
      <handler name="onclick">
         if (this.cnt % 2) {
newsfeed.setSrc(
"http://rss.cnn.com/rss/cnn_topstories.rss"
);}
         else
newsfeed.setSrc("http://rss.news.yahoo.com/rss/topstories");
         newsfeed.doRequest();
         this.setAttribute("cnt", this.cnt+1);
      </handler>
   </button>
   <window title="RSS Reader" x="80" height="150" width="350">
      <view>
         <view datapath="newsfeed:/rss/channel/item">
            <text name="txt" fontsize="9" resize="true"
                  text="$path{'title/text()'}" fgcolor="blue"/>
         </view>
         <simplelayout axis="y"/>
      </view>
      <scrollbar/>
   </window>
</canvas>
```

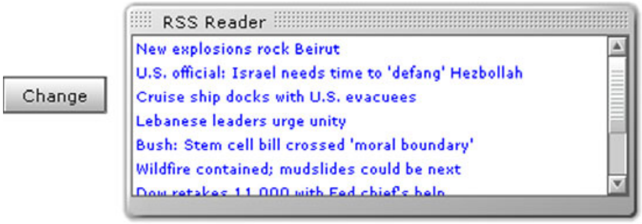
❶ **Generates initial display**

❷ **Changes news source**

❸ **Sends request to server**

The dataset's request attribute ❶ generates the initial display of the Yahoo! RSS news headlines, as seen in figure 16.1. Clicking the Change button updates the src attribute ❷ to display the URL for CNN's RSS feed. After this update, the doRequest method ❸ sends the request to the server.

Although a single dataset works adequately in this trivial example to demonstrate the basic dataset features, the shortcoming of this approach becomes apparent when we deal with real-world Internet problems such as HTTP servers timing out or



New explosions rock Beirut
U.S. official: Israel needs time to 'defang' Hezbollah
Cruise ship docks with U.S. evacuees
Lebanese leaders urge unity
Bush: Stem cell bill crossed 'moral boundary'
Wildfire contained; mudslides could be next
Dow retakes 11,000 with Fed chief's help

Figure 16.1 Titles from the Yahoo! RSS newsfeed are displayed initially. A click of the Change button switches the RSS newsfeed to CNN.

being down. In these cases, our RSS reader would lose its current display listing, and only an HTTP server exception code such as HTTP 404 or HTTP 500 would be returned. Providing a seamless viewing experience requires that buffered datasets be used to handle HTTP transfers.

16.1.2 Buffered HTTP datasets

Before an application discards its current display, it must have an acceptable next state. So, when an error or timeout blocks meaningful data, a supplemental error message should appear. But the current data state should be maintained. This allows a degraded but still usable application.

Maintaining this visual continuity requires multiple datasets: a *buffering dataset* to handle data transfers and a *destination dataset* for interfacing and binding to visual objects. Only when a data transmission successfully completes is the destination dataset updated with the buffered contents. This approach centralizes error processing and simplifies processing since the destination dataset is guaranteed to always receive valid data.

Here's a typical set of declarations for a pair of buffering and destination datasets. The buffering dataset provides handlers for the three possible outcomes of a request-response sequence: data, error, or timeout. In this case, valid data is handled in a `handleData` method, errors in a `handleErrors` method, and timeout situations in a `handleTimeout` method:

```
<dataset name="dsBuffer" type="http"
         ondata="this.handleData(this);"
         onerror="this.handleError(this);"
         ontimeout="this.handleTimeout(this);">
</dataset>
<dataset name="dsProducts"/>
```

Specifies the buffering dataset

Specifies the destination dataset

The HTTP standard specifies that any response status containing a numeric value between 200 and 299 is considered to be valid, resulting in a call to the `ondata` handler. A response in the 400 to 599 range is an error, resulting in a call to the onerror handler. An example of a typical error response is the 404 error generated when an HTML page can't be found. The delay invoking ontimeout is configurable in the timeout attribute, defaulting to 30 seconds.

Although this provides a workable system for communicating with an HTTP server, it requires a buffering dataset for every destination dataset. With a large number of datasets, this is a significant overhead. A better solution is to pool the buffering datasets among the display datasets. This is the rationale behind the LzHttpDatasetPool service.

16.1.3 Pooling buffering datasets

The LzHttpDatasetPool service maintains a pool of buffering datasets for sharing among any number of destination datasets. Datasets are not preallocated to the pool; an additional dataset is created whenever required. Consequently, the number of pooled datasets increases to handle the highest level of traffic. Released datasets are put back into the pool for the next HTTP connection.

The LzHttpDatasetPool object has two methods: get retrieves a pooled dataset and recycle releases this dataset back into the pool. This call obtains a buffering dataset:

```
var ds = LzHttpDatasetPool.get(this.dataDel, this.errorDel,
                               this.timeoutDel);
```

where

- dataDel is a delegate for the ondata event.
- errorDel is a delegate for the onerror event.
- timeoutDel is a delegate for the ontimeout event.

A delegate associates a method with each event. The once qualifier ensures that the dsLoad, dsError, and dsTimeout methods are set up only once:

```
<attribute name="dataDel"
    value="$once{new LzDelegate(this, 'dsLoad')}"/>
<attribute name="errorDel"
    value="$once{new LzDelegate(this, 'dsError')}"/>
<attribute name="timeoutDel"
    value="$once{new LzDelegate(this, 'dsTimeout')}"/>
```

The LzHttpDatasetPool service ensures that all buffered datasets are always clean, empty, and ready for use to communicate with an HTTP server.

16.1.4 Building a data service

A *data service* is a global object providing an HTTP-based service that is easily accessible by other Laszlo objects. For example, a useful place for a data service is the use of getProducts to populate the Product List window in the Laszlo Market. A data service isolates lower-level HTTP-based communications within a convenient interface.

HTTP server communications are built on requests and responses. We'll want to keep the server implementation isolated in a class, for easy revising to work with other networking technologies such as SOAP or XML-RPC. Instead of directly invoking the methods of a dataset, we'll create a data service object to encapsulate the HTTP communication details for the dataset.

A data service interface consists of a matching pair of request and response methods corresponding to each server-related service required by a dataset. The *request method* retrieves data from the HTTP server and the *response method* handles the server's response. A further level of encapsulation behind the interface contains the common data transfer operations for buffered datasets.

Let's suppose that we have a dsProducts dataset contained in a productDataService object whose request method is called getProducts. The productDataService's getProducts method, used to populate the dsProducts dataset, looks like this:

```
productDataService.getProducts("New");
```

The advantage of data service objects is that they can easily be integrated into an application. Suppose we are controlling an application with a state controller. We could control the populating of a product listing for a particular screen by adding a call to the productDataService's getProducts method (see listing 16.2).

Listing 16.2 Sample application state controller

```
<node id="gController">
   <handler name="onappstate" args="state">
      var title = "";
      switch (state) {
         …
         case "Login to Main":
              this.setAttribute("currstate", "Main");
              productDataService.getProducts("new");
              break;
         … }
   </handler>
   …
</node>
```

Listing 16.3 shows a supporting gDataService library to handle the buffering dataset issues required to handle HTTP requests and responses. The dataset's pathurl attribute provides a base URL. We'll construct a request from the base URL and an associate array to hold the URL parameters. An error attribute is used to alert other view objects about network problems; this allows any visual object to easily provide an error indicator.

Listing 16.3 productDataService.lzx handles server-related interaction for the dsProducts dataset

```
<library>
   <dataset name="dsProducts"/>
   <node name="productDataService">
      <attribute name="error" type="string" value=""/>
      <attribute name="pathurl" type="string"
                 value="http://www.laszlomarket.com/store/"/>
      …
   </node>
   …
</library>
```

Listing 16.4 shows a getProductParams method for packaging the URL parameters into an associative array. Although Laszlo provides an LzParams utility for handling HTTP parameters, a JavaScript associative array works just as well.

Listing 16.4 productDataService.lzx: building a query string

```
<method name="getProductParams">
   var params = {};
   params.action   = "list";
   params.category = "new";
   return params;
</method>
```

Listing 16.5 shows the data service object's request and response methods to retrieve all products from the HTTP server for the dsProducts dataset. Although any name can be used for the response, we adopt a convention of using the request name and appending Result.

Listing 16.5 productDataService.lzx: getting a product listing

```
<method name="getProducts">
   var requesturl = pathurl + "products.do";
   var params = getProductParams();
```

```
      gDataService.sendRequest(this,
                   requesturl, params,
                   "getProductsResult");
   </method>
   <method name="getProductsResult" args="status, data">
      if (status == true)
         this.appendChild(
              data.getFirstChild());
      else
         Debug.write("getProductsResult Failed: " + data);
   </method>
```

Establishes request-response link

Updates dataset

All that remains is to encapsulate the common data transfer operations for using buffering datasets, and to connect the request and response methods.

Implementing data transfer operations

For a clean interface to our data services, we package the common underlying methods into a globally accessible node-based object called `gDataService`, which

- Obtains and disposes of a buffering dataset
- Transfers data from the buffering to the destination dataset
- Provides an asynchronous response method
- Provides standard error handling

To make it globally accessible, we define the `gDataService` object as a top-level variable in a library in listing 16.6. It uses the `LzHttpDatasetPool` service with a set of delegate attributes to attach methods for the data, error, and timeout events. The `sender` argument of `sendRequest` corresponds to the object, in this case to the `dsProducts` dataset that invokes the request. The buffering dataset stores the names of both the sending object and the response method to set up the automatic invocation of the response method.

Listing 16.6 gDataService.lzx: encapsulating common HTTP functionality

```
<library>
   <node name="gDataService">
      <attribute name="loadDel"
         value="$once{new LzDelegate(this,
               'dsLoad')}"/>
      <attribute name="errorDel"
         value="$once{new LzDelegate(this,
                'dsError')}"/>
      <attribute name="timeoutDel"
         value="$once{new LzDelegate(this,
               'dsTimeout')}"/>
```

Sets up data, error, timeout delegates

```
    <method name="sendRequest" args="sender, url,
                    params, response">
      var ds = LzHttpDatasetPool.get(this.loadDel, this.errorDel,
                                     this.timeoutDel);
      ds.setAttribute("sender",sender);                Saves sender name,
      ds.setAttribute("response",response);            response method
      ds.setSrc(url);
      ds.setQueryType('POST');            Updates
      ds.setQueryParams(params);          HTTP settings
      ds.doRequest();     <-- Sends
    </method>                 HTTP request
    …
  </node>
  …
</library>
```

When a valid HTTP response arrives, the dataset receives an ondata event, which is handled by the dsLoad method with the buffering dataset as an argument. Although the response is valid, it must still be checked for a status message indicating server-side processing errors. When the HTTP server returns an error, the returned XML response consists of a status element with error and message attributes containing an error description:

```
<response>
   status error="true" message="Can't find any products"/>
</response>
```

This approach, shown in listing 16.7, centralizes error processing and simplifies the response method since it is now guaranteed to receive only valid data.

Listing 16.7 gDataService.lzx (cont): associating the response with the request

```
<method name="dsLoad" args="ds">
   var ebyt = ds.getFirstChild().                      ❶ Finds status
              getElementsByTagName("status");       <--   code
   if (ebyt.length){
      if (ebyt.getAttr("error") == true) {       <-- ❷ Checks error attribute
         var msg = ebyt.getAttr("message");
         this.setAttribute("error",                    ❸ Displays error
                "Request Failure: " + msg);        <--   message
         LzHttpDatasetPool.recycle(ds);
         return; } }
   ds.callback[ds.response]                      ❹ Invokes response
            (ds.getFirstChild());            <--   handler
   LzHttpDatasetPool.recycle(ds);
   return;
</method>
```

The method dsLoad first checks the status code ❶ by searching through the first child node for a status node. If a status node ❷ is found, its error attribute is checked and its message attribute ❸ is displayed in an error window. Remember that getAttr retrieves XML attributes, while setAttribute sets dataset attributes. Next, we invoke the response handler ❹. This line of code requires an extra bit of explanation.

An object in JavaScript is represented by a name-value associative array. Since JavaScript treats functions as data—they are used anywhere a data value can be used—a value in this array can be a function. This is important here because the response string can be an identifier to access the response method. Because this is a method, it can naturally take an argument, which is an LzDataElement object containing the data returned by the server, minus its wrapping node:

```
ds.sender[ds.response](ds.getFirstChild());
```

When a valid HTTP response arrives, the productDataService's getProductsDataResult method is automatically invoked with an argument containing the returned XML data. This allows the appropriate response handler to be automatically called.

Finally, standardized methods for handling the error and timeout conditions are required. When one of these conditions occurs, the gDataService's error attribute is set with the failure reason and the buffered dataset is released back into the pool (see listing 16.8).

Listing 16.8 gDataService.lzx: handling error and timeout conditions

```
<method name="dsError" args="ds">
   this.setAttribute('error', 'The request failed ' + ds.src);
   LzHttpDatasetPool.recycle(ds);
</method>
<method name="dsTimeout" args="ds">
   this.setAttribute('error', 'The request timed out. ' + ds.src);
   LzHttpDatasetPool.recycle(ds);
</method>
```

Any visual object can use the error attribute as a constraint to ensure that error messages are automatically displayed. This provides a wide latitude for how these messages are displayed. Now that you've seen how to communicate with an HTTP server, let's use it to save our application state.

16.2 Accessing sessioned data

Although a Laszlo application can operate independently by maintaining state in its datasets, this data is released and lost when the application terminates. Longer persistence requires an outside server to store its state. Since HTTP is a stateless protocol, a server uses a *session* object to maintain state for each client. A session generally has a time limit, 30 days for example, before it needs reinitializing.

Associating a server's session with a particular browser requires a way to associate it with a particular client browser. Each session contains a session `id` that allows it to be easily located. Including this session `id` with each request from a client browser identifies its corresponding session state. Several techniques have been developed to transfer a session `id`: cookies, URL encoding, and SSL sessions. Cookies are the most popular method, so we'll look at them. A web server initially inserts a cookie containing a unique session identifier into a response header and sends it to the browser. Assuming that the browser supports cookies, it processes the cookie header and stores it for later use. On subsequent requests, the browser includes this cookie. When the server processes a request, it checks the cookie for the session `id` to identify the browser.

Laszlo uses this same mechanism for both its Flash and DHTML applications. Because the browser automatically performs the processing, Laszlo doesn't have to perform any special processing. Everything works just as with regular HTML applications. Although Flash has its own session mechanisms such as Flash cookies, there's no reason not to use the cookie facilities available in the browser.

16.2.1 Building a sessioned shopping cart

Whenever a shopping cart's contents are updated, we call the shopping cart data service to update the server's session to reflect the changes. This data service contains the CRUD (create, replace, update, and delete) methods to update a session's contents to reflect these actions:

- Populating a shopping cart from a session
- Updating a sessioned shopping
- Deleting from the shopping cart

The following sections examine each of these operations.

Populating a shopping cart from a session

At application startup, the shopping cart needs to be initialized with any previously saved contents. These contents are obtained through the `cartDataService`'s getShopCart method. In listing 16.9, this method is added to the state controller's Login to Main state to ensure the shopping cart window is updated before the main screen is first displayed.

Listing 16.9 controller.lzx: retrieving the initial shopping cart contents at startup

```
<library>
   <node id="gController">
      …
      <handler name="onappstate" args="state">
         …
         case "Login to Main":
            title = "Login to Main";
            productDataService.getProducts();
            cartDataService.getShopCart();
            break;
      </handler>
   </node>
</library>
```

Listing 16.10 shows that when valid data is received by the response method, it is copied to the dsCart dataset. Since this data conforms to the shopping cart object's data path XPath expressions, the shopping cart's display is automatically updated to reflect the contents. Afterward, the total amount for the shopping cart items can be calculated by the shopping cart object's updateTotals method.

Listing 16.10 cartDataService.lzx: retrieving initial contents

```
<library>
   <dataset name="dsCart"/>

   <node name="cartDataService">
      <attribute name="pathurl"
                 value="$once{canvas.apiurl + '/store/'}"
                 type="string"/>
      <method name="getShopCart">     <|—— Builds request for cart contents
         var params = null;
         var requesturl = pathurl + "xcart";
         gDataservice.doRequest(this, requesturl, params,
                                "getShopCartResult");
      </method>

      <method name="getShopCartResult"           Receives cart
              args="status, data">     <|——————   contents
         if (status == true) {
            dsCart.setChildNodes([data.getFirstChild()]);
            main.shoppingcart.shopcart.
```

```
                            updateTotals();
            Debug.write("ShopCartData returned: ", data);
         } else {
            Debug.write("getShopCartDataFailed: "+data);
         }
      </method>
      ...
    </node>
</library>
```

Calculates shopping cart totals

Now that we know how to populate a shopping cart, let's look at the other CRUD-related methods supported by the cart data service to manage a sessioned shopping cart.

Updating a sessioned shopping cart

In chapter 12, we added a set of scoreboarding methods to enable all input sources to easily update the shopping cart. One of these methods, `updateShopcart`, now needs to be updated from operating with a local dataset to communicating its changes to an HTTP server. Because adding and updating a shopping cart item both occur in this method, we'll handle them together. In either case, the server must update its stored session. Listing 16.11 shows `updateShopcart` updated with additional persistence-related methods.

Listing 16.11 shoppingcart.lzx: telling the server to add or update an item to a session

```
<class name="shoppingcart" … >
   …
   <method name="updateShopcart" args="dp">
      <![CDATA[
      var curr = dp.xpathQuery("@sku");
      var exist = dptr.xpathQuery("item/@sku");
      if (exist != null) {
         if (typeof exist != "object") {
            var items = new Array();
            items[0] = exist;
            exist = items; }
         for (i = 0; i < exist.length; i++) {
            if (exist[i] == curr) {
               dptr.setXPath("dsCart:/items/item[@sku='"
                     + curr + "']");
               var qty = dptr.getNodeAttribute("qty");
               var sku = dptr.getNodeAttribute("sku");
               dptr.setNodeAttribute("qty", ++qty);
               cartDataService.
                updateShopCartItem(sku,qty);
               main.shoppingcart.shopcart.update_totals();
               return; }}}
```

❶ Updates quantity of existing item

```
        var ele = new LzDataElement("item");
        var sku =
            dp.getNodeAttribute("sku");
        var title =
            dp.getNodeAttribute("title");
        var image =
            dp.getNodeAttribute("image");
        var price =
            dp.getNodeAttribute("price");
        ele.setAttr("sku", sku);
        dptr.p.appendChild(ele);
        cartDataService.addShopCartItem(sku);
        main.shoppingcart.shopcart.update_totals();
        return;
        ]]>
    </method>
</class>
```

❷ **Creates new item for cart**

❸ **Adds new item to cart**

When a matching SKU isn't found in the shopping cart, a new item is created ❷ with a default quantity of 1. The addShopCartItem method updates the session ❸ with this new item. When a matching SKU is found, only its quantity ❶ is updated. The SKU is used to find the matching item in the stored session.

To support persistence in the shopping cart, we only need to add calls to the cart data service's updateShopCartItem and addShopCartItem methods. Listing 16.12 shows the implementation of these methods.

Listing 16.12 cartDataService.lzx: adding and updating items

```
<method name="addShopCartItem" args="sku">
    <![CDATA[
    var requesturl=pathurl + "add_to_cart/";
    var params=this.getShopCartParams(sku,1);
    gDataservice.doRequest(this, requesturl,
                  params, "getStatusResult");
    ]]>
</method>
<method name="updateShopCartItem" args="sku, qty">
    <![CDATA[
    var requesturl = pathurl + "update_cart/";
    var params =
        this.getShopCartParams(sku, quantity);
    gDataservice.doRequest(this, requesturl,
                  params, "getStatusResult");
    ]]>
</method>
<method name="getShopCartParams"
        args="sku, qty">
    var params = {};
```

Builds request for new item

Builds request to update item

Builds parameter array

```
      params.sku = sku;
      params.qty = qty;
      return params;
   </method>
   <method name="getStatusResult"
           args="status">
      if (status == false)
          Debug.write("getShopCartDataFailed: "+data);
   </method>
```

Contains common status response

addShopCartItem and updateShopCartItem only return a status response, so we'll create a single response method called getStatusResult to handle them. Also, the supporting getShopCartParams method is used to convert arguments into URL parameters to be included in an HTTP request to the server.

Since a shopping cart's quantity field can be directly updated through its input field, this is another spot where the server's saved shopping cart must be updated with the updateShopCartItem method (see listing 16.13).

Listing 16.13 shoppingcart.lzx: updating the number of items

```
<edittext name="qty" valign="middle" width="30" fontstyle="bold"
          doesenter="true" fontsize="10" datapath="qty/text()">
   <method name="doEnterDown">
      var qty = this.datapath.setNodeText(this.getText());
      var sku = this.datapath.getNodeAttribute("sku");
      cartDataService.updateShopCartItem(sku, qty);
   </method>
</edittext>
```

In a complete implementation, we'd check for negative or non-numeric values, and enforce an upper limit on the item number. We omit these details here to focus on the central HTTP issues.

16.2.2 Deleting from the shopping cart

We've shown how an item can be deleted from the shopping cart by dragging and dropping it into the trash. Listing 16.14 updates this with a call to the cart data service's deleteProduct method to instruct the server to delete this item from its session.

Listing 16.14 main.lzx: deleting a shopping cart item when it's dropped into the trashcan

```
<handler name="onmousetrackup">
   ...
   var sku = dragger.datapath.getNodeAttribute("sku");
```

```
    cartDataService.deleteProduct(sku);
</handler>
```

Listing 16.15 shows the `deleteProduct` and its supporting parameters method that sends a request to the server to delete this product from its session.

Listing 16.15 cartDataService.lzx: deleting a product

```
<method name="deleteProduct" args="sku">
   <![CDATA[
   var requesturl =
                pathurl + "delete_product/";
   var params =
           this.getDeleteProductParams(sku);
   gDataservice.doRequest(this,
     requesturl, params, "getStatusResult");
   ]]>
</method>
<method name="getDeleteProductParams"
        args="sku">
   var params = {};
   params.sku = sku;
   return params;
</method>
```

Builds HTTP request to delete item from cart

Builds parameter list

This completes the CRUD-related data services for managing the server's sessioned shopping cart. Whenever the contents of a shopping cart are modified, one of the `cartDataService` methods is called to build and send an HTTP request to the server. Although other data services, such as login and order completion, are required in a real application, we've omitted them here since they're just another set of data services.

As Figure 16.2 shows, determining the API marks the end of user-centered design. To understand how a back-end server implementation, such as Struts or Ruby on Rails, supports the Laszlo Market with XML-over-HTTP services, please take a look at appendix A or B online.

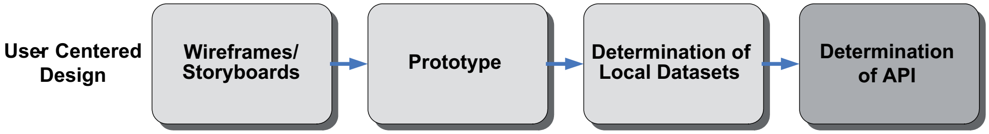


Figure 16.2 In our top-down development framework, we have advanced to the final stage of determining the API.

16.3 Maintaining server domains

Real-world application development requires an application to be deployed in different domains during its development lifecycle. An application typically moves from its initial development platform to a *staging environment* for testing and QA. After successfully completing its QA phase, an application is ready for production deployment. A *server domain* is an HTTP server designed specifically to support a particular phase of application development. For the Laszlo Market, we'll move through three phases: initial development, staging, and deployment. An application needs to be able to easily switch domains. To do this, we'll create a config.xml file to supplement the main controller.lzx file. An `url_env` attribute is added to the `gController` object to control the definition of an `apiurl` attribute containing the current deployment environment: `dev`, `staging`, or `www` (for production):

```
<library>
   <attribute name="apiurl"
       value="http://localhost:8082" type="text"/>

   <state apply="${gController.url_env == 'dev'}">
      <attribute name="apiurl" value="http://localhost:8082"
                               type="text"/>
   </state>
   <state apply="${gController.url_env == 'staging'}">
      <attribute name="apiurl" value="http://localhost:8080"
                               type="text"/>
   </state>
   <state apply="${gController.url_env == 'www'}">
      <attribute name="apiurl" value="http://www.laszloinaction.com"
                               type="text"/>
   </state>
</library>
```

The current deployment environment is easily changed by updating the value of the `gController`'s `url_env` attribute. Instead of a static value, a browser's URL query string can be used in a constraint to dynamically set the HTTP server domain. Now this value is propagated throughout the application, updating all the URLs.

In this chapter, we've shown how a Laszlo application can easily transition from working with local datasets to network-supplied data. This approach delays integration issues to the end of application development to minimize their impact. Using the supporting classes shown in this chapter provides an easy path for this integration. This approach can also be supplemented with server domain settings, providing a migration path for an application from development to final deployment.

16.4 Summary

This chapter demonstrated the ease with which a local dataset implementation can be converted to interface to an HTTP server. Our initial code base was transitioned to a networked environment by adding a handful of data service calls that handle all HTTP-related communication with the server. Since these data services are methods in a dataset object, they update that dataset. This isolation of a Laszlo application from implementation details made the transition relatively easy.

Because we ensure that the composition of an XML document returned by the server is compliant with the reference composition established by the local dataset, none of the data path XPath statements for the view-based objects need updating. Thus, the application's operation is identical for local and networked datasets. This development strategy relaxes the coupling between the client and server development groups, allowing each to work independently. It also encourages experimentation with different data compositions and APIs without impacting the other side's efforts.

Using XML-over-HTTP as the communication medium allows Laszlo to work with any HTTP-based web server since, by definition, all HTTP servers must support this capability. In this chapter, we only explored Laszlo's role in this exchange. Appendix A and appendix B, online, feature two server-specific implementations, Java-based Struts and Ruby on Rails, creating a complete client-to-server implementation.

Using an HTTP server with database access provides a volume of data that can't be replicated using local datasets. While this provides new capabilities, it also introduces optimization issues that are dealt with in the upcoming chapters. In the next chapter, we examine data-related optimization issues. The final chapter deals with application-level optimization issues.

17 Managing large datasets

This chapter covers

- Building sorting filters with setNodes
- Building mapping and merging filters with setNodes
- Optimizing with lazy replication
- Creating expandable displays with resize replication
- Creating paged datasets

One of the nicest things about being big is the luxury of thinking little.

—Marshall McLuhan,
communications theorist

In Laszlo's context, a large dataset is one with a large number of matching data node names. Up to now, we have only dealt with small datasets. This provided us with some luxurious working habits, such as allocating view resources to every displayed dataset element. This simplified our presentation so we could focus on fundamentals. But this freedom causes problems in dealing with web servers connected to databases delivering large datasets. Now, we need to learn how optimization replication managers can be used to rein in our resource consumption.

Laszlo's vertical communication system is built on the `datapath`. With its default setting, a `datapath` creates a replication manager when its XPath returns multiple data nodes. As a result, Laszlo's comprehensive set of optimization replication managers are all invoked through different `datapath` attribute settings. Their general approach to optimization is to trade performance for size in order to retain flexibility. Laszlo doesn't employ a single approach; rather, *lazy replication*, *pooling*, and *paged datasets* handle different aspects of the problem. We'll demonstrate how easily the Laszlo Market can be converted to use all these optimization techniques.

We'll start by taking another look at the operations of the replication manager to observe how the data path's `setNodes` method provides a backdoor into its operations. This allows alternative processing filters to be inserted into the data binding system. These filters can provide common filtering capabilities such as sorting, mapping, and merging of datasets.

17.1 Processing with alternative filters

The `setNodes` method is such a useful tool that it probably deserves its own tag. Instead, its powerful capabilities are buried as a method within the `datapath`. As a result, many developers aren't familiar with how to use it to build *alternative processing filters*. These filters supplement standard dataset processing with specialized capabilities such as sorting and merging. These capabilities can be transparently added without modifying the relationship between a view and its bound dataset. For example, if a sorting filter is added, an object's existing `datapath` expression still works but now returns sorted matching data nodes. A merging filter could be added on top of this to return a larger set of sorted data nodes from more than one source.

We'll start with an overview of the data path/replication manager system and `setNodes`' role within it.

17.1.1 The setNodes backdoor

Building processing filters doesn't modify a dataset's data nodes. Instead, a static copy of these data nodes is maintained and all modifications occur to this copy. This approach allows any number of alternative processing filters to be added or even stacked. Figure 17.1 illustrates how the `setNodes` method is the missing step to register this static copy with the replication manager that maintains the data path binding.

Once the updated data nodes are registered back with the data replication manager, the data path mechanism works normally to bind these data elements to visual objects. The `setNodes` method is a clever way to supplement the normal operation of the data path replication manager system. It provides the best of both worlds: the flexibility of JavaScript and compatibility with Laszlo's data-binding communication system.

In this next section, we'll demonstrate some of `setNodes`' capabilities with a multikey sorting example.

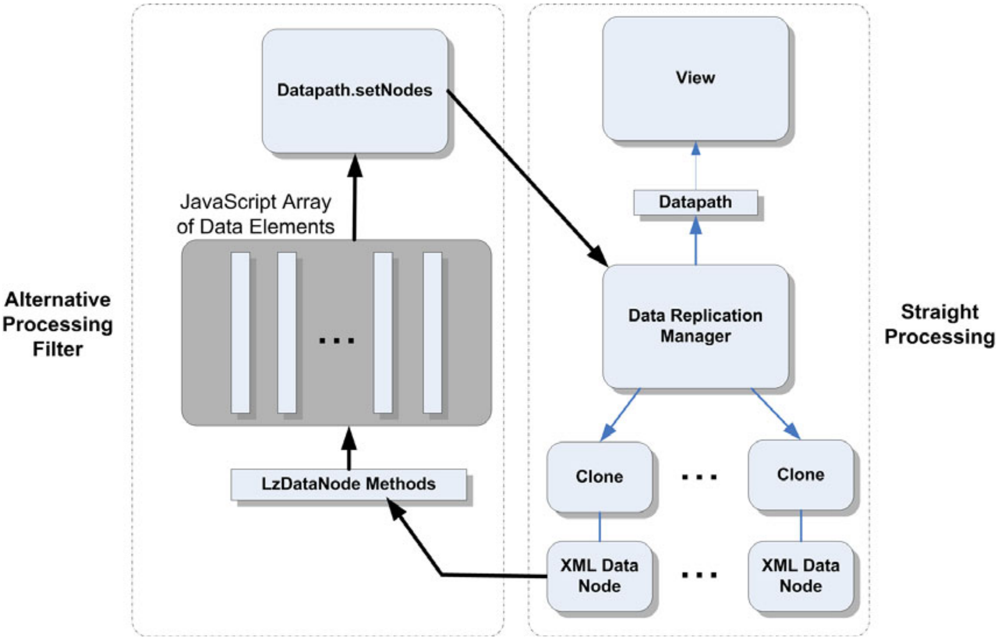


Figure 17.1 The `setNodes` method is a convenient backdoor mechanism for manipulating data nodes with JavaScript and re-registering them with a data replication manager. This introduces alternative processing into the data path replication system.

17.1.2 Multikey sorting with setNodes

One shortcoming of previously discussed sorting methods is that fact that they are applied only at the display level. Although displayed results are sorted, the XML data is unchanged. Since a static copy of the data nodes isn't maintained, adding a new item results in a re-sort of the dataset. If a static copy were available, the additional item could simply be inserted into its correct place. We'll now demonstrate how to create a static copy of the data nodes, sort its contents, and use `setNodes` to register these contents with a replication manager. In addition, this technique supports more complex sorts such as multikey sorting.

Listing 17.1 demonstrates a multikey sort where matching titles are further sorted by price.

Listing 17.1 Multikey sorting with `setNodes`

```
<canvas>
   <script>
      <![CDATA[
      function sortByTitlePrice( a, b ){
         var title_a = a.getAttr('title');
         var title_b = b.getAttr('title');
         if (title_a == title_b) {
            if (a.getAttr('price') ==
               b.getAttr('price')) return 0;
            return a.getAttr('price') <
               b.getAttr('price') ?  -1 : 1;}
         else if (title_a > title_b) {
            return 1; }
         return -1; }
      ]]>
   </script>
   <dataset name="dsProducts">
      <products>
         <product sku="SKU-001" title="Die Hard" price="29.95"/>
         <product sku="SKU-002" title="Speed" price="34.95"/>
         <product sku="SKU-003" title="Speed" price="31.95"/>
         <product sku="SKU-004" title="Die Hard" price="33.99"/>
         <product sku="SKU-005" title="Wizard of Oz" price="19.95"/>
      </products>
   </dataset>
   <method event="oninit">
      list = dsProducts.
                getFirstChild().childNodes;
      list.sort(sortByTitlePrice);
      repMgr.datapath.setNodes(list);
   </method>
   <window title="Product Listing" width="200">
```

❶ Sorts by title and price

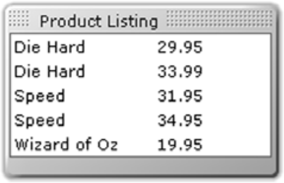
❷ Gets all product entries

❸ Calls customized sort

❹ Contains results

```
      <view id="repMgr" datapath="dsProducts:/products/product">
         <text width="100" datapath="@title"/>
         <text width="100" datapath="@price"/>
         <simplelayout axis="x"/>
      </view>
      <simplelayout/>
   </window>
</canvas>
```

A JavaScript array named list is assigned ❷ all the product tags from the dataset. This is the static copy of the dataset's data node that will be sorted. Next, the JavaScript library function `sort` is used ❸ to sort the contents of the list array. The JavaScript `sort` library can't call Laszlo methods, so the JavaScript function `sortByTitlePrice` ❶ is called. It first sorts by title and then by price. Once it completes, `setNodes` ❹ registers the contents of the sorted list back with the replication manager. Now all of the replicated clones are updated with these sorted data nodes. Figure 17.2 shows the multikey sorted output.



```
Product Listing
Die Hard       29.95
Die Hard       33.99
Speed          31.95
Speed          34.95
Wizard of Oz   19.95
```

Figure 17.2 In this output from the two-column sort in listing 17.1, the results are sorted first on title and second on price.

But this is only one type of processing filter that can be inserted into a data-binding stream. We'll now look at other filters to merge and map multiple datasets into a single virtual dataset.

17.1.3 Merging and mapping datasets

Suppose the Laszlo Market makes a 20 percent margin on its products, and has an agreement with another store to supplement its stock with their products—perhaps they specialize in foreign videos—at a 10 percent margin. However, this other store's XML data is structured differently from ours. In fact, suppose there are many such stores, each with potentially different XML data. This is a common situation as smaller stores use the Web to collectively compete against larger stores. These differing data streams need to be merged into a single stream.

We can solve this problem in one of two ways: either we introduce an intermediate server to merge the streams or, even better, we use mapping filters. Mapping filters provides a more flexible and inexpensive solution to this problem. Listing 17.2 demonstrates how a mapping filter can be used. The composition of product data from our store, contained in a file called our_store.xml, is expressed using node attributes:

```
<products>
   <product image="dvd/spiderman_II.jpg" title="Spiderman II"
            price="11.99" sku="SKU-001"/>
   <product image="dvd/speed.jpg" title="Speed"
            price="10.99" sku="SKU-002"/>
</products>
```

while the composition of product data arriving from another store, contained in a file called other_store.xml, is expressed with child nodes:

```
<products>
   <product sku="SKU-003">
      <title>Bourne Supremacy</title>
      <price>9.99</price>
      <image>dvd/bourne_supremacy.jpg</image>
   </product>
   <product sku="SKU-004">
      <title>The Terminator</title>
      <price>9.99</price>
      <image>dvd/the_terminator.jpg.jpg</image>
   </product>
</products>
```

Since we prefer our data composition, because it can be processed slightly faster, the other store's data is mapped and merged to match that of our store.

Listing 17.2 A mapping filter for the Laszlo Market

```
<canvas>
   <dataset name="dsOurStore"    src="our_store.xml"/>
   <dataset name="dsOtherStore"  src="other_store.xml"/>

   <method name="init">
      var list = dsOtherStore.getPointer().
        xpathQuery(
          "dsOtherStore:/products/product");
      list = fixNodes(list);
      list = appendDatasets(list);
      repMgr.datapath.setNodes(list);
   </method>

   <datapointer name="dp"/>
   <method name="fixNodes" args="list">
      <![CDATA[
      for (var i = 0; i < list.length; i++) {

         dp.setPointer(list[i]);
         var title = dp.xpathQuery("title/text()");
         var price = dp.xpathQuery("price/text()");
         var image = dp.xpathQuery("image/text()");
         var sku = dp.xpathQuery("@sku");
```

❶ Gets other store's data

❷ Rearranges other store's data to match ours

❸ Appends store datasets together

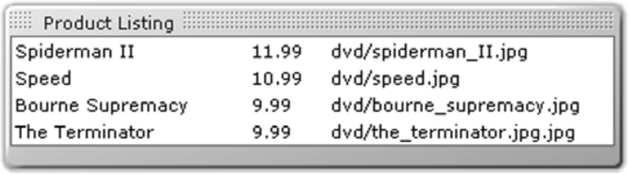
❹ Registers JavaScript array with replication manager

```
         list[i] = new LzDataElement("product",
            {sku: sku, title: title, price: price, image: image},
            [new LzDataElement("desc")]); }
      return list;
      ]]>
   </method>
   <method name="appendDatasets" args="list">
      <![CDATA[
      var map = dsOurStore.getPointer().
                xpathQuery("dsOurStore:/products/product");
      var last = map.length+1;
      for (var i = 0; i < list.length; i++) {
         map[last+i] = list[i]; }
      return map;
      ]]>
   </method>

   <window title="Product Listing" width="400">
      <view id="repMgr"
         datapath=
            "dsOurStore:/products/product">
         <text width="150" datapath="@title"/>
         <text width="50" datapath="@price"/>
         <text width="200" datapath="@image"/>
         <simplelayout axis="x"/>
         </view>
      <simplelayout/>
   </window>
</canvas>
```

❺ **Creates data path to access merged data**

First, a copy of the other store's product data nodes ❶ is obtained. This data is rearranged ❷ to match our data layout; its individual data elements are collected and a new `DataElement` with the correct layout is stored as an element in a JavaScript array. The two datasets with identical layouts are merged ❸ into a single JavaScript array. The `setNodes` method registers ❹ this JavaScript array with the replication manager. The contents of the array are now accessible by the data path ❺ as a single dataset.

Figure 17.3 displays the combined contents of these dissimilar datasets.



```
Spiderman II        11.99   dvd/spiderman_II.jpg
Speed               10.99   dvd/speed.jpg
Bourne Supremacy    9.99    dvd/bourne_supremacy.jpg
The Terminator      9.99    dvd/the_terminator.jpg.jpg
```

Figure 17.3
A mapping filter can be used in the Laszlo Market to merge output from disparate datasets.

Mapping filters greatly expand the capabilities of Laszlo, allowing any web server that delivers XML data to be interwoven with other XML data sources to form a combined data stream.

We have only scratched the surface of applying processing filters. Filtering takes mashups to a whole new level. A *mashup* is a website or web application that mixes and matches content from many sources to create a completely new service. Normally, a *mashup* occurs at the view level, but the same concepts can be applied at a data level to build composite data streams.

Now that we can build large composite datasets, we'll investigate ways to optimize their display.

17.2 Optimizing data display

Once we connect to an HTTP-supplied dataset, we lose the ability to control its size. Since we don't know how large a dataset may become, we must plan for a worst-case scenario. Instead of a single approach, Laszlo uses a combination of techniques to handle different aspects of this problem. The replication manager is extended to support *lazy replication* managers that optimize the allocation of resources for the current display. Lazy replication limits the replicated clones to those that are currently visible. *Lazy resize replication* extends this to support dissimilarly sized listings. *Pooling* recycles clone resources when adding or removing entries.

But this solves only half the problem of displaying large datasets. The other half is solved with paged datasets. The contents of a paged dataset are incrementally retrieved rather than downloaded all at once. This relieves users of having to wait for an entire dataset to be downloaded before being able to use it.

We'll illustrate each of these techniques with the Laszlo Market.

17.2.1 Lazy replication

Lazy replication is used to display large datasets in a view or window with a scrollbar. It's called lazy because only visible clone objects have view resources allocated for them. The lazy replication manager allocates resources for all its matching data node objects, because they are relatively lightweight. However, its cloned objects have large resource requirements, so their resource allocation is controlled. A pool of clone objects, corresponding to the number of displayable objects, is created. When the scrollbar moves, resources are recycled by transferring them from the scrolled off to the newly visible clones. Since lazy replication is so simple to use, it should always be used in a scrolled window displaying dynamic content.

A lazy replication manager object, LzLazyReplicationManager, is created when the data path's replication attribute is set to lazy and its XPath query returns multiple matching data nodes. There are a couple of restrictions on the LzLazyReplicationManager:

- The physical size of each replicated view must be identical.
- Selections are made through the dataSelectionManager rather than the selectionManager.

The dataselectionmanager must be used, because it operates on the data nodes, rather than on a cloned object, which would no longer exist if it should scroll out of view.

Although lazy replication is activated with a single attribute setting, it requires other modifications to its surrounding declarative statements. Let's examine what's involved in converting from normal to lazy replication.

Converting from normal to lazy replication

Converting an application from normal to lazy replication requires some adjustments to its declarative tag layout. With normal replication, a data path is normally attached to a view-based object as an attribute. Lazy replication requires that the data path be converted into a child node of the view-based object. This provides access to the data path's attributes so they can be set for different replication types. Layout for normal replication is generally handled by a simplelayout tag through its spacing attribute. When lazy replication is used, the axis and spacing attributes are used instead, as shown in table 17.1, to control its layout. Also the layout tag used for normal replication must be removed.

Table 17.1 LzLazyReplicationManager attributes

Name	Data Type	Attribute	Description
axis	x or y	Settable	The axis for layout of replicated views
spacing	number	Settable	The spacing between replicated views

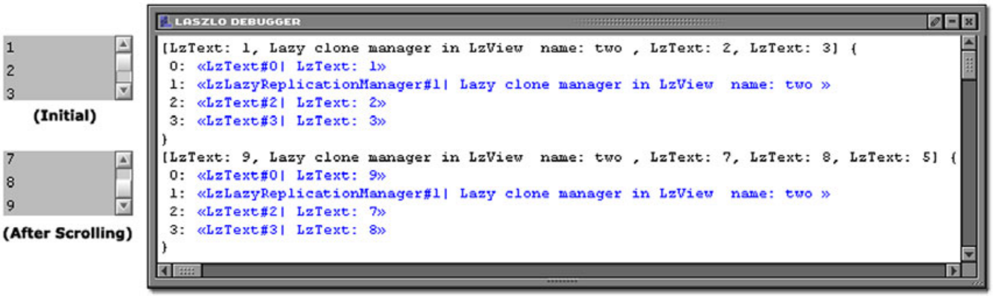
Listing 17.3 shows how the number of allocated clone resources is controlled by the display size.

Listing 17.3 Effects of lazy replication

```
<canvas>
   <dataset name="numbers">
      <num>1</num><num>2</num><num>3</num><num>4</num>
      <num>5</num><num>6</num><num>7</num><num>8</num><num>9</num>
   </dataset>
   <view name="one" height="50" width="100" clip="true">
      <view name="two" bgcolor="0xBBBBBB">
         <text name="three">
            <datapath name="four"
               xpath="numbers:/num/text()"
               replication="lazy" spacing="1"/>
            <handler name="onclick">
               one.setHeight("105");
               Debug.inspect(parent.subnodes);
            </handler>
         </text>
         <handler name="oninit">
            Debug.inspect(subnodes);
         </handler>
      </view>
      <scrollbar/>
   </view>
</canvas>
```

Replaces simplelayout with spacing attribute

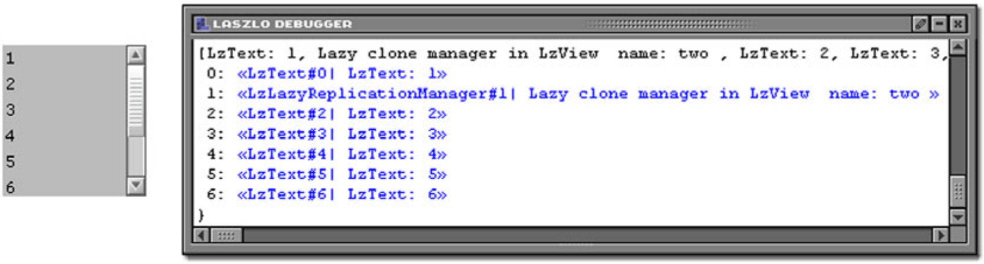
Figure 17.4 shows that setting a data path's `replication` attribute to `lazy` turns on the lazy replication manager. While nine replicated node objects are created, only the three view objects necessary to populate the display are created. A normal replication manager would create nine replicated node and view objects.



```
[LzText: 1, Lazy clone manager in LzView  name: two , LzText: 2, LzText: 3] {
 0: «LzText#0| LzText: 1»
 1: «LzLazyReplicationManager#1| Lazy clone manager in LzView  name: two »
 2: «LzText#2| LzText: 2»
 3: «LzText#3| LzText: 3»
}
[LzText: 9, Lazy clone manager in LzView  name: two , LzText: 7, LzText: 8, LzText: 5] {
 0: «LzText#0| LzText: 9»
 1: «LzLazyReplicationManager#1| Lazy clone manager in LzView  name: two »
 2: «LzText#2| LzText: 7»
 3: «LzText#3| LzText: 8»
}
```

Figure 17.4 The recycling of views in the pool of available views can be observed after moving the scrollbar.

Clicking the displayed numbers causes the containing view's height to expand to 100 pixels to accommodate three additional view objects. Figure 17.5 shows how this causes the `LzLazyReplicationManager` to increase the size of its view object pool to six.



```
[LzText: 1, Lazy clone manager in LzView  name: two , LzText: 2, LzText: 3,
 0: «LzText#0| LzText: 1»
 1: «LzLazyReplicationManager#1| Lazy clone manager in LzView  name: two »
 2: «LzText#2| LzText: 2»
 3: «LzText#3| LzText: 3»
 4: «LzText#4| LzText: 4»
 5: «LzText#5| LzText: 5»
 6: «LzText#6| LzText: 6»
}
```

Figure 17.5 Changing the height of the clipped view requires additional clones to display the text views.

One limitation of lazy replication is that displayed items must have identical heights. Although this isn't a problem for our current prototype of the Market, we would like to add built-in outline display for a selected product; this requires an expanded product selection to accommodate this outline. To support this feature, we need to use the `LzResizeReplicationManager`.

17.2.2 Handling expansible listings

`LzResizeReplicationManager` extends `LzLazyReplicationManager` with the capability to handle variable-sized replicated elements. All that is necessary to switch managers is to set the data path's `replication` attribute to `resize`. Listing 17.4 demonstrates using the resize replication manager to create expandable listings.

Listing 17.4 Expandable listing with `LzLazyReplicationManager`

```
<canvas>
   <dataset name="numbers">
      <num>1</num><num>2</num><num>3</num><num>4</num>
      <num>5</num><num>6</num><num>7</num><num>8</num><num>9</num>
   </dataset>
   <view height="100" width="100" clip="true">
      <view bgcolor="0xBBBBBB">
         <dataselectionmanager name="selector" toggle="true"/>
         <text datapath="numbers:/num/text()"
               onclick="parent.selector.select(this)">
            <datapath replication="resize"
                      spacing="1"/>
            <method name="setSelected" args="selected">
               if (selected) {
                  this.setAttribute("bgcolor", 0xCCCCCC);
                  this.setHeight(40);
                  this.setAttribute("fontsize", "16"); }
               else {
                  this.setAttribute("bgcolor", 0xBBBBBB);
```

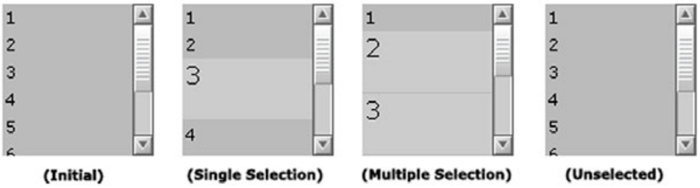
Creates a resize replication manager

Expands height for selected entry

```
                        this.setHeight(15);
                        this.setAttribute("fontsize", "11"); }
                  </method>
               </text>
            </view>
            <scrollbar/>
         </view>
      </canvas>
```

Collapses height for unselected entry

Figure 17.6 shows how selected views are resized, with a large font size, to make them even more distinctive.



(Initial)

(Single Selection)

(Multiple Selection)

(Unselected)

Figure 17.6 `LzResizeReplicationManager` **allows lazy-replicated list entries to have different sizes.**

In this example, selected views simply change size abruptly. This size change can be made more relaxing by delaying its presentation. We'll add this feature as we update the Laszlo Market with expandable displays.

17.2.3 Expandable displays in the Laszlo Market

An expandable display is an elegant way to give users selection information. When a short outline is embedded in the selection, a user can quickly assess a product and go to the Product Details window for more information. Rather than having the display jump from one size to another, we have it gently expand and collapse. Listing 17.5 shows the steps for updating the `productwin` class (defined in chapter 12) to contain an expandable Product List display.

Listing 17.5 Creating an expandable product listing

```
<class name="productwin">
   …
   <view name="container" width="100%"
            height="${immediateparent.height}"
                           clip="true" focusable="true">
      <view name="scroll" width="100%">
         <dataselectionmanager name="selector"
                              toggle="true"/>
```

❶ Replaces selectionmanager with dataselectionmanager

```
          <productrow name="products" width="100%"
                      onclick="parent.selector.select(this)">
            <datapath xpath=
              "dsProducts:/products/product"
                   replication="resize"
                   spacing="2"/>
          </productrow>
        </view>
        <myscrollbar focusview="$once{parent}"/>
      </view>
    </class>
```

❷ **Converts datapath from attribute to stand-alone object**

First, the `selectionmanager` tag is replaced ❶ with a `dataselectionmanager` tag; then the `datapath` is converted from a `productrow` attribute into a stand-alone object ❷ so the data path's `replication` attribute can be set to `resize`. The existing `simplelayout` tag is no longer needed, as the layout is now specified through the data path's `spacing` attribute.

When clone resources are recycled, the `dataselectionmanager` saves the selection and reapplies it when the original clone reappears. Although this works fine for ensuring that selections remain selected, it causes a problem for the master-detail relationship between the Product List and Product Details windows. Whenever a product selection scrolls out of view, it loses selection, along with its resources—resulting in an `ondata` event causing the Product Details window to go blank. Listing 17.6 shows how to animate the expandable product list and fix this selection problem.

Listing 17.6 Animating the expandable product listing

```
<class name="productrow" fontstyle="bold" fontsize="12">
   <attribute name="selected" value="false" type="boolean"/>
   <attribute name="title" valign="middle" datapath="@title"
                  multiline="true" resize="true"/>
   ...
   <handler name="onmouseover">
      this.setAttribute("selected", true);
   </handler>
   <handler name="onmouseout">
      this.setAttribute("selected", false);
   </handler>
   <method name="setSelected" args="isselected">
      if (isselected) {
         if (select)
            productdp.setPointer(this.data);
```

❶ **Tracks mouse position**

❷ **Updates productdp when mouse over selection**

```
            this.setAttribute("lastcolor", this.bgcolor);
            setAttribute("bgcolor", 0xBBBBBB);
            this.title.setAttribute("datapath",
               "outline/text()");
            this.title.setAttribute("fontsize",
                                    "10");
            this.animate("height", 100, 1000);
            this.image.setAttribute("valign",
                                  "middle"); }
         else {
            if (select) productdp.setPointer(null);
            setAttribute("bgcolor",
                         this.lastcolor);
            this.title.setAttribute("datapath","@title");
            this.title.setAttribute("fontsize","12");
            this.image.setAttribue("valign","top");
            this.animate("height", 80, 1000); }
      </method>
      ...
   </class>
```

❸ Displays outline data node

❹ Expands height to accommodate outline

❺ Adjusts image to middle of available space

❻ Restores original display

We'll use a pair of onmouseover and onmouseout event handlers ❶ that manage a selected attribute that reflects the current mouse position. We distinguish between user- and dataselectionmanger-initiated selections by requiring that the mouse be over the product row to update the productdp data pointer ❷. Since this can't occur when a row scrolls onto and off the scrolling area, it prevents updating of the Product Details window.

Figure 17.7 shows how selecting a product causes it to expand and switch to displaying an outline, instead of a title. The title text field's data path ❸ is updated to point to the outline data node. To fit the contents into the available

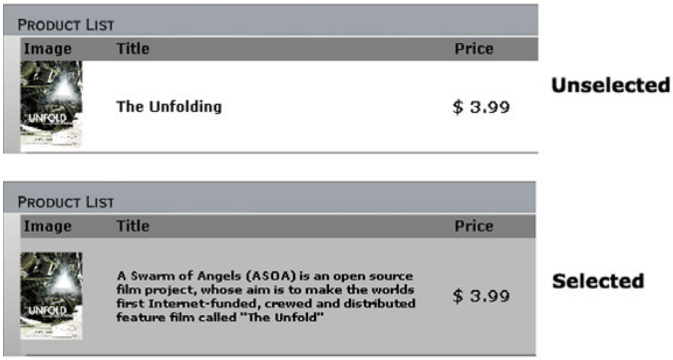

Product List
Image
Title
Price
The Unfolding
$ 3.99
Unselected
Product List
Image
Title
Price
A Swarm of Angels (ASOA) is an open source film project, whose aim is to make the worlds first Internet-funded, crewed and distributed feature film called "The Unfold"
$ 3.99
Selected

Figure 17.7 The top image shows a product listing in an unselected state. The height of the listing is 80 pixels. The bottom image shows it in a selected state. The height attribute is slowly animated from 80 to 100 pixels. This provides extra space to display a short outline.

space, its font size ❹ is decreased to 10 pt. Instead of simply changing the height of the entry, it is animated to slowly expand to 100 pixels. Also, the position of the image is moved ❺ to fill the middle of the frame. When an item is deselected, the attributes return to their original values ❻ to reset the product.

Now that you are familiar with optimizing listings, let's look at how to optimize the insertion and deletion of list entries to minimize screen flicker.

17.2.4 Pooling

If the data nodes bound to the cloned objects change at runtime, the replication manager normally destroys and re-creates the clones. This can cause a noticeable flicker, as the new cloned objects are instantiated. To address this problem, the data path's `pooling` attribute can be turned on to recycle the existing cloned objects. Since the replication manager only needs to remap the updated data nodes to the existing clones, data updates are faster, with no flicker. The `pooling` attribute can be added to normal replication and is automatically set with lazy replication.

Listing 17.7 shows a view that switches from one bound dataset to another when clicked on its displayed contents. One dataset contains numbers and the other letters. Under normal replication, when the switch occurs the displayed views are destroyed and a new set of view objects is instantiated. With pooling, these views are recycled for reuse. A button labeled Pool toggles the data path's `pooling` attribute to allow us to observe the effects.

Listing 17.7 Comparing normal and pooled replication

```
<canvas>
   <dataset name="letters">
      <item>A</item>
         …
      <item>Z</item>
   </dataset>
   <dataset name="numbers">
      <item>1</item>
         …
      <item>26</item>
   </dataset>

   <method name="toggle" args="list">
      var xpath = list.datapath.xpath;
      if (xpath == "letters:/")
         list.setDatapath("numbers:/");
      else
         list.setDatapath("letters:/");
   </method>
```

Switches datasets

```
      <simplelayout axis="x" spacing="5"/>
      <button text="Pool">
         <attribute name="pool" value="false" type="boolean"/>
         <handler name="onclick">
            this.pool = !this.pool;
            one.two.three.datapath.setAttribute("pooling", this.pool);
            if (this.pool) parent.msg.setText("pooling");
            else parent.msg.setText("Not Pooled");
         </handler>
      </button>
      <text name="msg" text="Not pooled"/>
      <view name="one" datapath="letters:/">
         <view name="two" bgcolor="0xBBBBBB">
            <text name="three" text="$path{'text()'}">
               <datapath xpath="/item"/>
               <method event="onclick">
                  canvas.toggle(one);
               </method>
            </text>
            <simplelayout axis="y" spacing="3"/>
         </view>
      </view>
   </canvas>
```

Toggles pooling

Dynamically switches pooled and unpooled datapaths

With the pooling attribute turned on, the fluidity of the dataset switch improves. This effect is even more noticeable when the clones contain memory-intensive resources with graphics. In the next section, we'll add both lazy replication and pooling to the Laszlo Market's shopping cart to help remove some of its choppiness.

Pooling in the shopping cart

Inserting or deleting from the shopping cart produces a momentary pause as the replication manager processes the updated data nodes and instantiates or releases resources for the clone. We can fix this by setting the replication attribute to lazy, since this also sets pooling:

```
<view name="container" width="100%"
      height="${parent.height - 20}" clip="true">
   <view width="100%" name="scroll">
      <shop_row datapath="item" width="100%">
         <datapath xpath="item"
             replication="lazy"
             spacing="30"/>
      </shop_row>
   </view>
   <scrollbar/>
</view>
```

Defaults pooling to on

Although the difference is too subtle to be displayed in this book, it is still observable to the eye. Although this is a small detail, attention to small details adds up to a superior overall viewing experience.

These different replication managers have all focused on optimizations designed to handle the page case, where a page represents enough displayable items to populate several screens or less. But how do you handle larger datasets containing significantly larger amounts of data? For that, you need paged datasets.

17.3 Paging datasets for long listings

Paged datasets complement the recycling techniques of lazy replication by incrementally loading large datasets. The returned results are divided into fixed-size pages, each page holding a fixed number of records and each record containing a product row. The response time for loading a large dataset is reduced by initially loading only the first page. Additional pages are loaded only on an "as-needed" basis, when they are referenced. If a page is never referenced, it's never downloaded. This strategy is also known as *lazy loading*.

Data display is controlled through a scrollbar's arrow buttons to move stepwise, or via a gripper to move in larger steps. In either case, moving through the displayed results eventually references nonresident data, which results in a request to a back-end server to retrieve a data page. Although a page can be of any size, it typically holds several screens' worth of rows. This utilizes *locality*, where it's assumed that a user will generally access information within a local area. This use of localization helps reduce the number of nonresident page hits. Pages are accumulated, so the dataset's complete contents are accumulated over time, which also causes page requests to decrease over time.

We'll start to design this paging system by examining the relationship between the scrollbar and its display. Figure 17.8 shows that when the scrollbar is moved, through either its gripper or its arrows, it generates an `ony` event as a scrolling child view is moved against its parent view. The scrolling child view starts at zero and generates progressively larger negative y values as we scroll forward.

As the scrollbar moves down, the scrolled view's `y` attribute grows negatively, causing it to move upward, and expose a new row in the display. What is now needed is an algorithm to retrieve a new page just when an otherwise blank row would appear at the bottom of the display.

Since scrolling occurs both forward and backward, it's necessary to check for nonresident page rows at both the top and bottom of the displayed area. When a nonresident page is detected, a request is generated to retrieve it. Generally only

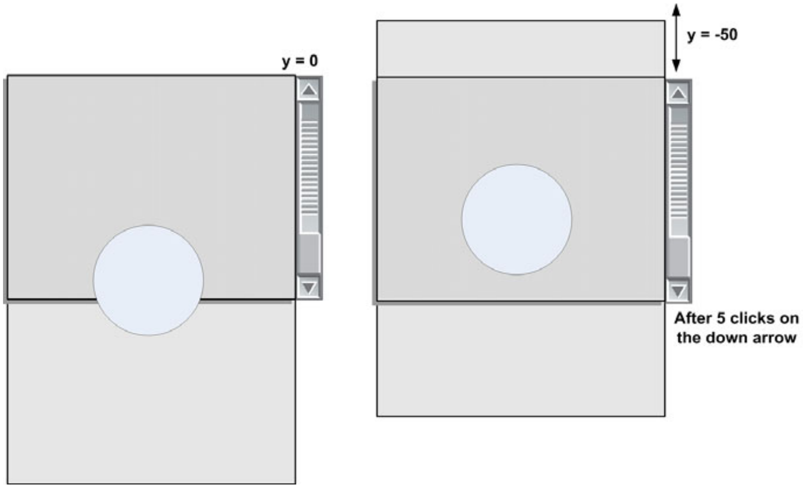


Figure 17.8 Initially, as the parent clipping view displays only the top portion of the circle, the `y` attribute for the scrolled view is 0. Pressing the scrollbar's down arrow five times causes the scrolled view's `y` attribute to move stepwise by a default value of 10 pixels for each click. This moves the scrolled view upward by 50 pixels, exposing the entire circle.

a single page needs to be retrieved, but when the scrollbar is positioned so that a page boundary is displayed, both pages must be checked and possibly retrieved.

The first step in this check is to convert the backing view's `y` displacement to an absolute row number in the dataset, using zero-origin indexing:

```
top_row = Math.floor(-y / row_height);
```

Because the `y` value is always negative, it's negated to make it positive. It's then divided by the row height and rounded down to an integer. Next we find the window size in rows:

```
window_rows = Math.ceil(window_height / row_height);
```

The value is rounded up to give the minimum number of rows displayed at one time. Also, note that if the window size is not an integral number of rows, one row is always partly obscured. The maximum number of rows displayable is one more than `window_rows`, in which case two rows are partly obscured.

The page numbers corresponding to these top and bottom rows are found by dividing their row numbers by the page size; these values are rounded down to give a page number using zero-origin indexing:

```
top = Math.floor(top_row / page_size);
bottom = Math.floor((top_row + window_rows) / page_size);
```

Both top and bottom now contain page numbers, in most cases the same page. We also maintain a loaded array that stores whether a page is resident or needs to be loaded. It is still necessary to check both the top and bottom values to determine if these pages are resident, because occasionally a page boundary appears. Whenever a page isn't found in the loaded array, the product data service's loadpage method needs to be called to load it:

Let's illustrate with a simple example. Suppose there are 10 five-row pages, where each row is 20 pixels high, and rows and pages start at zero. So each page contains 100 pixels. If the scrollbar is moved to a value of –218, then, as shown in figure 17.9 on the left, rows 12 through 15 are displayed (they appear within the dark outline). Since all these rows reside within the third page, only page 3 needs to be retrieved. In the figure on the right, rows 28 through 31 are displayed. Since these rows cover two pages or, put another way, a page boundary is displayed, it requires that two pages (pages 6 and 7) must be retrieved. When a page needs to be retrieved, loadPage is called to load it. So in this case, it will be called twice with each page number.

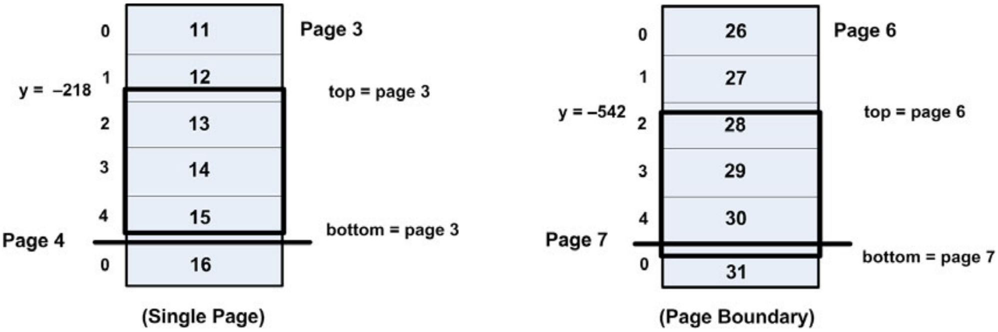


Figure 17.9 The ceil and floor methods are used to include top and bottom rows that only partially appear within the display area, indicated by the heavy black square. In the left example, the display area is filled with the rows of a single page, resulting in a single page retrieval. In the right example, the display area straddles two pages, resulting in retrieval of both pages.

Now that we have all the pieces, let's put them together in the Laszlo Market to handle paging for the product rows.

17.3.1 Adding paged datasets to the Market

Let's consider the steps to update the Market's Product List window with paged datasets. The productwin class already contains the replicated product rows and accompanying scrollbar. The container view defines the physical parameters for the display area, while the scroll view serves as the backing view populated with paged data. The scroll view also handles the ony events, generated by moving the scrollbar, to initiate the retrieval of new pages. Listing 17.8 shows the modifications to add paged datasets to the productwin class.

Listing 17.8 Adding paged datasets to the Laszlo Market

```
<class name="productwin">
   …
   <view name="container" width="100%"
         height="${immediateparent.height}"
         clip="true" focusable="true">
      <view name="scroll" width="100%">
         <dataselectionmanager name="selector" toggle="true">
            <handler name="oninit">
               this.sel = "setSelected";
            </handler>
         </dataselectionmanager>
         <productrow name="products" width="100%" height="79"
                     onclick="parent.selector.select(this)">
            <datapath xpath="dsProducts:/products/product"
                      axis="20" spacing="20" replication="lazy">
               <handler name="ondata"
                  if(main.contents.rowheight
                     == 0) return;
                  productDataService.
                    rowheight =
                      this.clones[0].height;
               </handler>
            </datapath>
         </productrow>
         <handler name="ony">
            if (typeof this.del ==
               "undefined" )
             this.loadDel = new LzDelegate(
                this, 'loadPage');
             LzTimer.addTimer(loadDel, 500);
          </handler>
      </view>
      <myscrollbar focusview="$once{parent}"/>
   </view>
   ...
</class>
```

❶ Finds row height from data

❷ Prevents interim page requests

To determine the row offset for the top row, we need to find a product's row height. This information is best obtained ❶ from the product itself, so that it is automatically updated to handle future design changes.

We can't allow every change in y to generate a page request; as this would generate spurious page requests as the scrollbar moves through interim positions. Instead, we require the scrollbar to remain stationary in one position for a fixed amount of time, say half a second, to indicate a selected position. A delegate is added ❷ with a delay timer to ensure that a page isn't called for half a second. If any subsequent ony events occur during this period, then the timer is reset.

Let's now look at implementing the paging logic. Implementing paged datasets is a collaborative effort between a Laszlo client and an HTTP server. The server must support paged datasets and be able to respond to a request for a page. Appendix B online contains a Ruby on Rails implementation that supports paging. Additionally, the server must supply information that describes the contents of each returned page. This information consists of the current page number, the total number of pages, and a count of the total number of records. In the example page response, this information is contained in attributes of the parent node:

```
<response>
   <products page="0" totalpages="10" recordcount="98">
      <product id="1" sku="SKU-001" ... />
      ...
   </products>
</response>
```

In this example page response, each page contains ten product data nodes or records, with the exception of the last page, which contains eight. All paging-related parameters are stored as attributes in the dsProducts dataset to make them easily accessible:

```
<library>
   <dataset name="dsProducts"/>
   <node name="productdataservice">
      <attribute name="url" type="string"
                 value="http://localhost:3000/store/list"/>
      <attribute name="loading" value="false" type="boolean"/>
      <attribute name="row_height" value="0" type="number"/>
      <attribute name="loaded" value="[]"/>
      <attribute name="total_pages" type="number"/>
      <attribute name="page" value="0" type="number"/>
      <attribute name="record_count" value="0" type="number"/>
      ...
   </node>
</library>
```

Now the productwin class, shown in listing 17.9, has the necessary information to implement this paging algorithm. Whenever a page needs to be checked for residency, the product window's loadPage method is called. If it determines that a new page needs to be loaded, then it calls the product's `loadPagedData` data service to perform the low-level paging operations.

Listing 17.9 Implementing the paging algorithm

```
<class name="productwin">
   …
   <method name="loadPage">
      <![CDATA[
      if (productDataService.loading)
        return;
      if (this.row_height == 0) {
        this.row_height = this.products.height;
          productDataService.loadPagedData(0);
        return; }
      var top = Math.floor(-products.y /
          productDataService.row_height);
      var display_area =
          Math.ceil(main.height /
          productDataService.row_height);
      bottom = Math.floor(top +
          display_area);
      bottom = Math.floor(bottom /
          productDataService.total_pages);
      top = Math.floor(top /
          productDataService.total_pages);
      if (bottom >
            productDataService.total_pages)
         bottom =
            productDataService.total_pages;
      if (productdataservice.loaded[top] &&
           productDataService.loaded[bottom])
        return;
      if (!productDataService.loaded[top])
           productDataService.loadPagedData(top);
      if (!productDataService.loaded[bottom])
           productDataService.loadPagedData(bottom);
      ]]>
   </method>
</class>
```

❶ Avoids multiple outstanding requests

❷ Checks for initial request

❸ Implements paging algorithm

❹ Checks top and bottom for residency

To avoid ❶ multiple outstanding page requests, the `productDataService` has a `loading` attribute that is turned on when the request is sent and turned off when its response is received. This permits two outstanding page requests for `top` and `bottom`. If no pages are loaded, then we make an initial page request ❷ to collect page-related information such as the height of each row, the number of records in a page, and the maximum number of pages. This information is used ❸ to check the top and bottom rows for nonresident pages ❹. If they haven't been loaded, the `loadPagedData` data service is called to load them.

While the `loaded` array maintains state among the pages, the `productDataService` maintains state among the data records. The dataset is initially populated to contain `record_count` number of empty `product` data nodes:

```
<dataset name="dsProducts">
   <products>
      <product/>      <!-- 1 -->
         …
      <product/>      <!-- 98 -->
   </products>
</dataset>
```

When a page of data is loaded, a page of empty product data nodes is updated with this newly loaded data. Over time the entire contents of the dataset will be loaded.

Listing 17.10 shows the implementation of the `loadPagedData` data service. It creates the request that is sent to the back-end server and updates the `dsProducts` dataset with the returned data nodes contained in the response.

Listing 17.10 Implementing the request and response for the `loadPagedData` data service

```
<library>
   <dataset name="dsProducts">
   <node name="productDataService">
      ...
      <method name="loadPagedData" args="page">
         <![CDATA[
         this.setAttribute('loading', true);
         var src = url +"?page=" + page";
         gDataservice.sendRequest                  Creates page
             (this, src, null,                     request
              "loadPagedDataResults");
         ]]>
      </method>

      <method name="loadPagedDataResults" args="status, data">
         <![CDATA[
         if (status == true) {
            data = data.getFirstChild();
            this.page = data.getAttr("page");          Discards wrapper,
            this.total_pages =                         gets attributes
               data.getAttr("total_pages");

            loaded[page] = true;
            if (this.record_count == 0) {
               record_count = data.getAttr(
                   "record_count");
               var empty_nodes =
                LzDataElement.makeNodeList(
                   record_count, "product");
               dsProducts.data.setChildNodes(
                   empty_nodes); }

            var per_page = Math.floor(
               record_count / total_pages);
            var cn =
               dsProducts.data.childNodes;
            var start = page  * per_page;
            for (var i = 0; i <
               per_page; i++) {
                  cn[start + i] =
                     data.childNodes[i]; }
            dsProducts.data.
                setChildNodes(cn);
            this.setAttribute('loading',
                               false); }
         else {
            Debug.write("getPagedDataFailed: "+data); }
         ]]>
      </method>
   </node>
</library>
```

Contains count of empty product nodes

Updates one page using array

The dsProducts dataset is large enough that it's faster to perform a bulk update than to individually update each XML data node. The entire contents of the dataset are first copied into a JavaScript array. The page number is multiplied by the calculated number of records per page to determine where the page of data should be loaded within this array. Next, the entire updated array is moved back into the dataset.

The loading attribute is used to inform users that data is in the process of being loaded. It's used in a constraint to control the parent view's background color:

```
<view name="container" width="100%"
      bgcolor="${productDataService.loading ?  0xcccccc : 0xffffff}" … >
```

The scrolled display has a gray background while the application is loading data; then, it returns to its normal background color.

Paged datasets greatly facilitate the viewing of large collections of data by splitting a dataset into smaller sections. In many cases, the entire dataset is large enough that it isn't practical to download all of it. In these cases, since a user is normally only interested in accessing a small portion of the dataset, only those pages of interest need to be downloaded.

17.4 Summary

This chapter capped our discussion of data paths. These last features addressed sorting, optimization of large datasets, and alternative processing filters. Each of these built on the underlying features of the data path replication manager architecture described in earlier chapters.

No system is complete without sorting. The data path uses its XPath features to define keys and other attributes for specifying sorting criteria. Custom sorting can be specified to provide nonalphabetical sorts or sorts on multiple keys.

We saw how the data path can be used to spawn a replication manager; it can also spawn a lazy replication manger to control the creation of view object clones, so that a user's computer isn't overwhelmed with demands for resources. Lazy replication limits the number of view-based objects by restricting their use to currently observable objects.

The paged dataset is an additional optimization technique that can be layered on top of lazy replication to make large datasets quickly available without downloading the entire dataset. This breaks the total download time into segments based on pages, whereby each page is loaded on an as-needed basis.

Finally, the data path's `setNodes` method can be used to loosen the tight coupling between bound objects and their XML data nodes, by introducing alternate processing filters into the data path replication manager architecture. These filters can provide many types of specialized processing, including the mapping and merging of disparate XML data feeds into a single XML data feed. This provides a powerful mechanism for integrating data from many different sources.

In many ways, the data path is the "main thoroughfare" of Laszlo; it permeates so many facets of the system that describing all its features has consumed several chapters. This chapter focused on optimization at the data-loading level; the next chapter looks at optimization at the system level.

18 Laszlo system optimization

This chapter covers:

- Creating dynamic libraries
- Understanding object initialization
- Controlling an object's initialization
- Using performance utilities

After all, all he did was string together a lot of old,
well-known quotations.

—H. L. Mencken, journalist and satirist,
writing about Shakespeare

Once your application is behaving the way you want, it may seem that you should be able to sit back, congratulate yourself on a job well done, and enjoy a cold drink. We're afraid not. The most difficult part of development comes next; your application must be optimized for acceptable performance across a wide array of target platforms. In fact, we already started this process in the previous chapter, with data display optimizations.

Although we use the general term *optimizing* here, it embodies two separate goals: *startup time* and *responsiveness.* Startup time is the delay until the application is ready to use. Responsiveness is the time from a user action until an indication of response appears. Overall performance involves a trade-off between these two. Performance goals for web-based applications have different trade-offs than for desktop applications. A desktop application has a captive audience, so it can tilt this trade-off toward a longer startup time to preload resources in exchange for better responsiveness. On the Web, users have a wider choice and are conditioned by HTML-based web applications that load quickly. If a Laszlo application doesn't start up quickly, a significant number of users will go elsewhere. Consequently, an optimized web application has a different trade-off between startup time and responsiveness.

In this chapter, we focus only on general system optimization strategies. The reason is that one goal of Laszlo is a single source-code distribution across Flash and DHTML targets. Consequently, we won't cover Flash and browser-specific optimization techniques here. General system optimization strategies work across all target platforms and deliver the largest return. Target-specific optimization techniques should be a last resort.

We'll begin by focusing on initialization. Once an application has a reasonable startup time, you can tackle responsiveness issues.

18.1 Dynamically loading optional elements

In the previous chapter, paged datasets were used to redistribute data-loading costs. Now we'll use a similar approach to redistribute an application's startup time. We'll start by segmenting our application into critical and optional elements. Critical elements are objects necessary for the core operation of the application. Optional elements include such things as unit testing, help sections, and other auxiliary functions. In this case, the critical elements will be loaded first, with the optional elements loaded on an "as-needed" basis. This makes them good candidates for being loaded as *dynamic libraries.*

18.1.1 Importing dynamic libraries

A dynamic library contains a library file whose loading can be deferred. A dynamic library can contain any Laszlo assets normally stored in a static library: classes, instances, scripts, datasets, and resource or font definitions. It works identically for applications implemented in either server or SOLO mode. Laszlo doesn't automatically keep track of loaded libraries, so you'll need to maintain a private record of downloaded libraries to prevent repeated downloads. Dynamic loading works best with relatively small libraries. Loading a large library can make an application momentarily unresponsive.

The `import` tag is used to dynamically import a library file. The attributes for the `import` tag are shown in table 18.1.

Table 18.1 Import attributes

Name	Data Type	Tag or Script	Attribute Type	Description
`href`	`URL`	Both	Setter	A reference to a target file whose content is treated as a loadable module.
`name`	`string`	Both	Setter	The name used to bind an object for invoking the load method on a defer library.
`stage`	`string: "late" or "defer"`	Both	Setter	A late library is loaded after the main application file has completed loading. A defer library is loaded programmatically through JavaScript.

The `href` attribute, which is required, can refer to a local file on the server, a SOLO file, or a URL for a networked resource. The `stage` attribute, also required, contains a `late` or `defer` string. A `late` setting imports the library at an opportune time, after the rest of the application has completed loading. With a `defer` setting, the library is loaded programmatically through a `load` method. When the `defer` setting is specified, the `name` attribute also must be specified to provide this object with a name. An `onload` event is always sent when the library is loaded, and `onerror` and `ontimeout` events are sent when a network error prevents a library from being returned.

Let's start with an example that dynamically loads the library modules corresponding to a menu of items. A dynamic library should only be loaded on the first access. After loading, the resident class definitions are used to instantiate the objects. Laszlo doesn't keep track of loaded libraries, so it's your responsibility to maintain a record.

18.1.2 Loading optional elements with dynamic libraries

Suppose we have an application containing a Modules menu with Fish and Cows menu items. In this application, the menu items are resident but a selected item's action is dynamically loaded. Since this menu could contain any number of menu items, our approach needs to scale to easily support a larger numbers of items. Each library consists of a `fish` and `cows` class respectively, stored in the files fish.lzx and cows.lzx, where each class defines a labeled rectangle.

Only one menu item can be in use at one time. This is a textbook example for using a generic object. Instead of maintaining a collection of objects, one for each menu item—and there can easily be tens, or even hundreds, of them—a single generic object is recycled for each item. When a new menu item is selected, the memory resources associated with the generic object are freed with the `destroy` method and reused for the new object.

Listing 18.1 shows a single menu with two items, labeled Fish and Cows.

Listing 18.1 Initializing optional functions with dynamic libraries

```
<canvas>
   <include href="load.lzx"/>
   <script>
      obj = null                     <-- 1 Contains generic object handle
   </script>
   <menubar>
      <menu text="Modules" width="100">
         <menuitem text="Fish">
            <method event="onselect">
               if (obj) {
                  if (obj.name == "fish")
                     return;
                  Debug.write(                        2 Checks for existing fish object
                     "obj cows destroyed");
                  obj.destroy(); }
               if (loadedModules["fish"]
                      == 1) {
                  obj = new fish(canvas);             3 Checks if fish library is loaded
                  Debug.write(
                     "Local fish: ", obj); }
               else {
                  Debug.write("load fish");
                  importFish.load(); }   <-- 4 Loads library on first Fish selection
            </method>
         </menuitem>
         <menuitem text="Cows">
            <method event="onselect">
```

```
               if (obj) {
                  if (obj.name == "cows")
                     return;
                  Debug.write(
                     "obj fish destroy");
                  obj.destroy(); }
               if (loadedModules["cows"]
                      == 1) {
                  obj = new cows(canvas);
                  Debug.write(
                     "Local cows: ", obj); }
               else {
                  Debug.write("load cows");
                  importCows.load(); }
            </method>
         </menuitem>
      </menu>
   </menubar>
</canvas>
```

❺ **Checks for existing cows object**

❻ **Checks if cows library is loaded**

❼ **Loads library on first Cows selection**

Assuming that a Fish item is initially selected ❷, the generic object handle is checked ❶ to see if it points to an object. Since it doesn't, the `loadedModules` array is checked ❸ to determine whether the `fish` library has been downloaded. Since it hasn't, the `import` object's `load` method is executed ❹ to load the library.

Later, if Cows is selected, the generic object is checked ❺. Because it's no longer null, the object name is checked. If its name is `cows`, we just return. But since it isn't, the object's memory is released. Now it's necessary to check ❻ whether the `cows` library has been downloaded. Since it hasn't, it needs to be downloaded ❼.

Because both libraries have now been downloaded, on subsequent selections of either menu item, depending on the object's name, control either returns or the memory for the existing object is released and allocated to the other object.

We next need a scalable architecture to organize all these library modules:

```
<library>
   <script>
      loadedModules = new Array();
      loadedModules["fish"] = 0;
      loadedModules["cows"] = 0;
   </script>

   <import name="importFish" href="fish.lzx"
                             stage="defer">
      <handler name="onload">
         loadedModules["fish"] = 1;
```

❶ **Records downloaded libraries**

❷ **Specifies deferred loading for fish library**

❸ **Marks fish library as loaded**

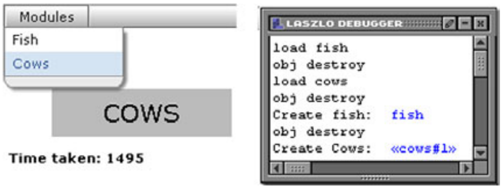
```
         if (obj) obj.destroy();           ❹ Checks for
         obj = new fish(canvas);             existing object
      </handler>
      <handler name="onerror">
         ...
      </handler>
      <handler name="ontimeout">
         ...
      </handler>
   </import>

   <import name="importCows" href="cows.lzx"
                             stage="defer">
      <handler name="onload">
         loadedModules["cows"] = 1;
         if (obj) obj.destroy();
         obj = new cows(canvas) ;
      </handler>
      <handler name="onerror">
         ...
      </handler>
      <handler name="ontimeout">
         ...
      </handler>
   </import>
</library>
```

First we allocate the `loadModules` array ❶, stating whether a particular downloadable object has been loaded. Then we specify a dynamic library ❷ referencing the fish.lzx file that is to be programmatically imported. The event handler ❸ is triggered when the dynamic library has been loaded over the network. The record of loaded dynamic libraries is updated to ensure that no library is loaded twice. If a generic view object already exists ❹, then we destroy it to release its memory and re-create it as a `fish` view object. All the previous steps are repeated for `cows`. Since these libraries are loaded only once, this creates the initial object.

Finally, the two downloaded classes in fish.lzx and cows.lzx must be defined:

```
<library>
   <class name="fish" x="40" y="50" width="150"
          height="40" bgcolor="0xCCCCCC">
      <text align="center" valign="middle"
            fontsize="20" text="FISH"/>
   </class>
</library>

<library>
   <class name="cows" x="40" y="70" width="150"
          height="40" bgcolor="0xBBBBBB">
      <text align="center" valign="middle"
            fontsize="20" text="COWS"/>
   </class>
</library>
```

The first time a menu item is accessed, the associated library module is dynamically loaded, as shown in figure 18.1. On subsequent accesses, the class library is resident so an instance is simply created. The total startup time is shown as 1.495 seconds, an acceptable startup loading time.



```
load fish
obj destroy
load cows
obj destroy
Create fish:   fish
obj destroy
Create Cows:   «cows#1»
```

Figure 18.1 On the initial access, the library is loaded. Subsequently, the object is just instantiated.

Any optional code module is a candidate for dynamic loading. The extra cost of dynamic loading, compared to a static library, is small enough to justify extensive use of dynamic libraries.

Now let's turn to the harder task of dealing with the initialization times for critical elements. Since an application relies on these elements for base functionality, they can't simply be dynamically loaded. To redistribute their startup costs, we'll modify their `initstage` attribute. But before we can do this, we need a better understanding of initialization of a Laszlo object.

18.2 Optimizing critical elements

Our strategy to reduce startup time for critical elements is to redistribute their loading to a later period. But since critical elements typically must be available for an application's initial display, there is less latitude for manipulating their initialization timing than with dynamic libraries. To help redistribute their initialization costs, all declarative objects have an `initstage` attribute to control their initialization. Knowing how to use this attribute correctly requires understanding the instantiation sequence for Laszlo objects.

18.2.1 Instantiating objects

The instantiation of a Laszlo application begins with the construction of the `canvas` tag and ends with the initialization of the `canvas` tag. The progression of instantiation traverses through the tags comprising the application, instantiating a parent tag's entire tree of child tags before moving to the next parent. The instantiation process is a continuing sequence in which a parent node completes its construction phase and then begins its initialization phase. The initialization phase consists of three sections: object initialization, creation of its children through the createChildren method, and, finally, its local initialization (contained within its init method). After creating its children, the parent must wait for all its children to complete their initialization before moving to its local initialization phase. This creates a chain of parents, as shown in figure 18.2, waiting for the last child node to fully complete, so the completions can propagate upward and allow each parent to complete.

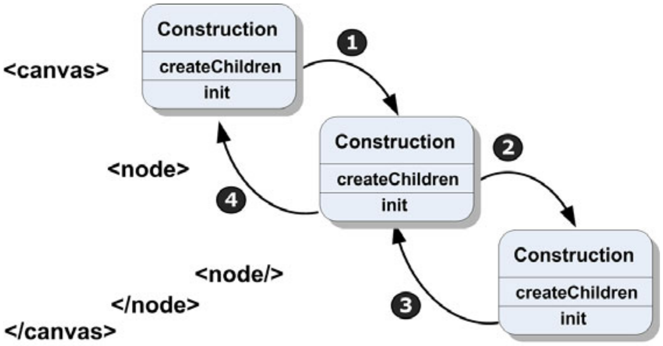


Figure 18.2
A parent tag can't complete its initialization until all its children have completed their initialization. So the construction phases occur top-down from parent to child, and the initialization phases occur in the reverse direction, from the last child to the parent.

Controlling the setting of the initstage attribute is the key for supporting different strategies to achieve initialization trade-offs.

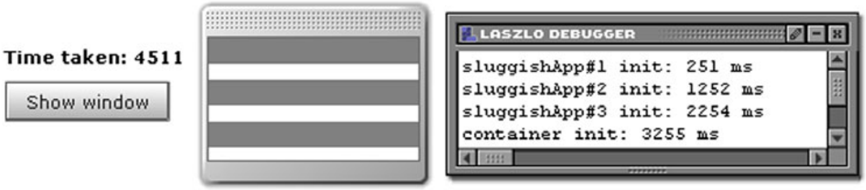
18.2.2 Manipulating instantiation with initstage

To demonstrate how the initialization period of a Laszlo application can be manipulated by its initstage settings, we'll simulate the performance of a sluggish application, which we'll call sluggishApp, by adding a time delay into its init method (the final step of initialization). This delay is created by repeatedly looping until 1,000 milliseconds (one second) has passed. To more easily demonstrate these results, we'll work in the debugger. Although normally you shouldn't run the debugger to obtain timing results, the values are large enough not to be unduly affected by the debugger.

The inittimer tag measures an application's total startup time; it can be added right after the canvas tag. Although the debugger doesn't normally produce accurate timing, the time differences are large enough to be meaningful. The following example results in three instances of the sluggishApp class displayed in a window and produces a 3-second delay:

```
<canvas>
   <inittimer fontstyle="bold"/>
   <script>ts = (new Date()).getTime()</script>

   <class name="sluggishApp" width="150" height="16"
          bgcolor="gray">
      <attribute name="spin" value="1000"/>
      <attribute name="num"/>
      <method name="init">
         <![CDATA[
         Debug.write("sluggishApp#" + num + " init: " +
                    ((new Date()).getTime()-ts) + " ms");
         var d = new Date();
         while ( (new Date() ) - d < spin ){};
         super.init();
         ]]>
      </method>
   </class>

   <button y="25"
         onclick="container.show()">Show window</button>
   <window name="container" x="130" width="150"
           height="125" visible="false">
      <method name="show">
              this.setVisible(true);
      </method>
      <simplelayout inset="5" axis="y" spacing="10"/>
      <sluggishApp num="1"/>
      <sluggishApp num="2"/>
      <sluggishApp num="3" />
      <method name="init">
         Debug.write("container init: " +
                    ((new Date()).getTime()-ts) + " ms");
         super.init();
      </method>
   </window>
</canvas>
```

When this application is executed, the `inittimer` tag informs us that initialization required over 4 seconds to complete (see figure 18.3).



```
Time taken: 4511
```

```
sluggishApp#1 init: 251 ms
sluggishApp#2 init: 1252 ms
sluggishApp#3 init: 2254 ms
container init: 3255 ms
```

Figure 18.3 This figure shows the elapsed time for the unoptimized version of the application. Our objective is to significantly decrease this startup time of over 4 seconds.

Since this application doesn't require that all objects be immediately available for display—the window isn't displayed until the button is clicked—our strategy focuses on deferring the initialization of these nondisplayed objects until they are needed.

18.2.3 Controlling initialization through initstage

Objects are entered into the `LzInstantiator` queue in the order established by their instantiation, and normally initialized in that sequence. However, each object has a priority, specified by its `initstage` attribute, that can modify this order. Table 18.2. shows the values for this `initstage` attribute.

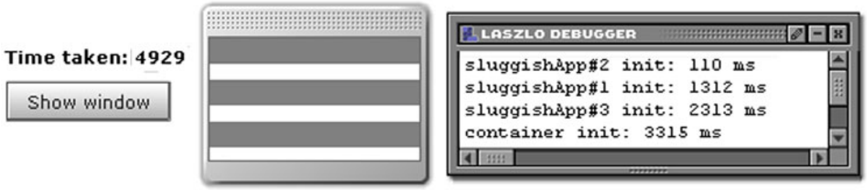
Table 18.2 `initstage` values

Attribute value	Description
`immediate`	Initialization occurs with highest priority. Invoked by `completeInstantiation`.
`early`	Initialization occurs with a high priority.
`normal`	Initialization occurs with a normal priority.
`late`	Allows the parent to complete their initialization. Child node initialization occurs with a low priority.
`defer`	Instantiation does not happen automatically. The object's parent must call the `completeInstantiation` method.

When this attribute is set to `normal`, the instances have equal priority and their `init` method is executed in order. But this order can be changed by setting an instance's `initstage` to `early`, like this:

```
<sluggishApp num="2" initstage="early"/>
```

As you can see in figure 18.4, the priority of the second `sluggishApp` has been raised, so its `init` method will be executed prior to the other instances. Although we were able to modify the execution order, this didn't result in any time savings.



Time taken: 4929

```
sluggishApp#2 init: 110 ms
sluggishApp#1 init: 1312 ms
sluggishApp#3 init: 2313 ms
container init: 3315 ms
```

Figure 18.4 The effect of setting the `initstage` attribute for the second instance is to increase its priority and allow it to execute before the other instances. But this doesn't change the application's startup time.

All this setting does is change the order; it doesn't save any time. A parent still must wait for all its children to complete.

The problem is the sequential nature of the node hierarchy processing. The key to decreasing startup time is to break the sequence of this processing.

Breaking the initialization sequence

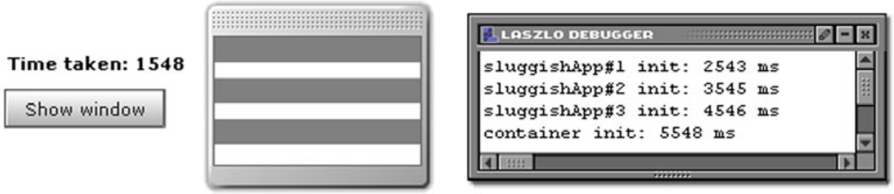
Since the window isn't displayed until the Show Window button is clicked, its initialization can be deferred until needed. Changing the window's `initstage` attribute to `defer` breaks the initialization sequence by freeing its parent to complete without waiting for the window and its children to complete.

When the `defer` option is set, the child nodes are entered into the `LzInstantiator` queue, but their priority is set below the runnable level. When it is time for the child nodes to execute, the parent node calls its `completeInstantiation` method to raise their priority level, to ensure they are immediately executed. Once the child nodes have completed their initialization, the parent node is finally able to complete its initialization and display the window with the three gray bar instances.

To switch to deferred instantiation, we need to make these two changes in the parent node called `container`:

```
<window name="container" x="120" width="150" height="135"
        visible="false" initstage="defer">
   <method name="show">
      this.setVisible(true);
      this.completeInstantiation();
   </method>
   <simplelayout inset="5" axis="y" spacing="10"/>
   <sluggishApp num="/>
   <sluggishApp/>
   <sluggishApp/>
</window>
```

Figure 18.5 shows that the startup initialization period has decreased from 4-plus seconds down to 1.5 seconds. Now the 3-second delay has been redistributed to



```
sluggishApp#1 init: 2543 ms
sluggishApp#2 init: 3545 ms
sluggishApp#3 init: 4546 ms
container init: 5548 ms
```

Figure 18.5 Setting `initstage` to `defer` allows the application to decrease its startup time from over 4 seconds to 1.5 seconds. The startup time still exists; it has just been redistributed onto the display of the window and its three instances.

start when the button is clicked. The total amount of time necessary for the entire display to be rendered is still approximately the same. The extra time is attributable to the amount of time spent moving the mouse to click the button.

There is still room for improvement. Currently, when the user clicks the button there is a long pause before the window with the gray bar instances appears. This display can be further enhanced by uncoupling the display of the window from the results of the sluggishApp instances. Setting initstage to late allows this uncoupling to occur.

Uncoupling a component from its contents

An entire component and its children can be uncoupled from an application by giving it a defer setting; when needed, its completeInstantiation method is called to force its contents to immediately initialize, thus freeing the component itself.

In this section, we take this one step further by uncoupling the component from its contents. This allows the component to appear immediately, while the contents appear later. To do this, we set each child's initstage attribute to late. This frees their parent component, in this case the window, to complete immediately.

Setting an object's initstage attribute to late is akin to a child telling its parent not to wait up because it will be staying out late. In technical terms, a child with a late setting has a low priority, causing it to execute slowly in the background. At the same time, its parent needn't wait for it.

We update our sluggishApp instances like this:

```
<sluggishApp num="1" initstage="late"/>
<sluggishApp num="2" initstage="late"/>
<sluggishApp num="3" initstage="late"/>
```

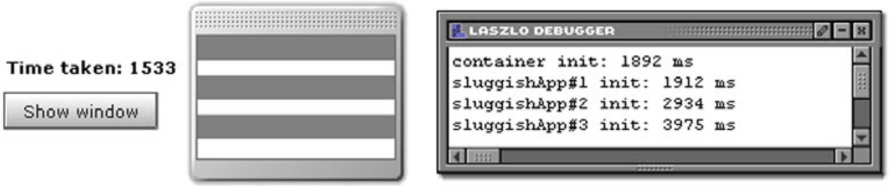
This frees the window to display, but there is still a problem. When the window issues its completeInstantiation command, it forces the immediate instantiation of its sluggishApp children. A sluggishApp consists only of a single child node, so this just recouples the contents back with the container. But here's a solution: we only need to add an intermediary container node inside sluggishApp:

```
<class name="sluggishApp" width="150" height="16"
                       bgcolor="gray">
   ...
   <node name="intermediary">
      <attribute name="spin" value="1000"/>
      <method name="init">
         <![CDATA[
         Debug.write("sluggishApp init: " +
                    ((new Date()).getTime()-ts) + " ms");
         var d = new Date();
         while ( (new Date() ) - d < spin ){};
         super.init();
         ]]>
      </method>
   </node>
</class>
```

Since the low priority specified by the late setting for a sluggishApp instance also applies to its contents, both the sluggishApp and its intermediary node initialize with low priority. However, when the parent window's completeInstantiation method is called, it affects only the outer sluggishApp instance, which heeds the order for immediate instantiation. Their inner intermediary nodes are free to initialize slowly in the background.

Although this sounds like a stratagem from some "prison escape" movie, it allows both sides to carry out their function. The window component immediately appears, with the gray stripes appearing later.

Figure 18.6 shows that the container has been decoupled by printing its init debug message before any of the sluggishApp class instances. Controlling the instantiation order decreases the display time from 4.5 seconds to 1.5 seconds. It also displays the window contents incrementally as the individual components are instantiated.



Time taken: 1533

```
container init: 1892 ms
sluggishApp#1 init: 1912 ms
sluggishApp#2 init: 2934 ms
sluggishApp#3 init: 3975 ms
```

Figure 18.6 The parent container is now decoupled from its children, so it is able to complete its initialization prior to the contained instances. This allows the window to appear quickly, and each instance appears sequentially in the window at one-second intervals.

It's not so important to understand the intricacies of Laszlo instantiation; what is important is how to apply them. These techniques allow almost any application, no matter how large and ungainly, to be optimized with a reasonable startup time.

18.3 Reducing the Market's startup time

We would like to reduce the Laszlo Market's startup time to be comparable with HTML-based web applications. This startup time reduction will be achieved by redistributing its initialization costs and moving noncritical application sections into dynamic libraries. Unit testing is auxiliary functionality and is always a good candidate for dynamic loading. Performing these optimizations on our application should significantly reduce its startup time.

18.3.1 Redistributing the Market's initialization

We'll redistribute the initialization time by decoupling the display of the Login window and its children from the rest of the application. This reduces our startup period to only the time needed by these objects to complete their initialization. We'll assume that users must first be authenticated before they can access the Laszlo Market, and authentication involves a round-trip visit to the HTTP server. The time spent waiting for authentication provides a convenient time slot to execute the application's remaining initialization.

To separate the initialization of the Login window from these other nodes, we'll set their `initstage` attribute to `defer` to ensure they aren't instantiated until `completeInstantiation` is called:

```
<canvas>
   ...
   <view name="main" width="100%" height="100%"
      opacity="0" initstage="defer">
   ...
   </view>
   <view name="checkout" x="${main.x+main.width}"
      width="75%" height="100%" initstage="defer">
   ...
   </view>
   <node name="operators" initstage="defer">
   ...
   </node>
   ...
</canvas>
```

After login completion, the `doLogin` method sends a request to the HTTP server. While waiting for the authentication result to be returned in `getLoginResult`, we'll complete the instantiation of these deferred top-level nodes with `completeInstantiation`:

```
<class name="login" … >
  …
  <method name="doLogin">
    gDataservice.sendRequest(this,
               src, null, "getLoginResult");
    canvas.main.completeInstantiation();
    canvas.checkout.completeInstantiation();
    canvas.operators.completeInstantation();
    this.close();
  </method>
```

Sends login to server

Uses delay to instantiate critical parts

```
</class>
```

The authentication response determines whether users stay at the Login window or continue to the next state in the state controller:

```
<method name="getLoginResult" args="status, data">
   var status = ds.getFirstChild().
             getElementsByTagName("status");
   if (status == true)
      gController.setAttribute("appstate",
                        "Login to Main");
   else {
      login.message("text",
                        "Invalid Login");
      gController.setAttribute("appstate",
                       "Splash to Login"); }
</method>
```

Gets login validity

Displays main screen

Redisplays login screen

Although our results are platform-specific and probably won't be the same on your system, `inittimer` indicates that this decoupling reduced our startup time from roughly 5.5 to 2.8 seconds, which is approximately a 50 percent reduction.

18.3.2 Dynamically loading noncritical elements

Unit-testing code is an auxiliary element that is always a good candidate for dynamic loading, since it's not needed for regular operation. We first need to move the unit-testing interface inside the `doLogin` method to ensure that all access is authenticated; otherwise we'd have a bad security hole:

```
<method name="getLoginResult" args="status, data">
   var status = ds.getFirstChild().
                       getElementsByTagName("status");
   if (status == true)
      if (LzBrowser.getInitArg("lzunit") == "true")
         importUnitTest.load();
      else
         gController.setAttribute("appstate", "Login to Main");
   ...
</method>
```

We only have unit-testing code that tests a small fraction of our application's functionality. A real-world application would contain significantly larger and more encompassing unit-testing code modules. This unit-testing code can easily be moved from the main file into a separate library file:

```
<library>
   <class name="unitTest" extends="TestSuite">
      <TestCase name="testcase">
         <method name="testCheckout">
            gController.setAttribute("appstate",
                                    "Main to Checkout");
         </method>
         <method name="checkout_test">
            assertEquals(-(canvas.width*.75), main.x);
            gController.setAttribute("currstate",
                                    "Checkout to Main");
         </method>
         <method name="main_test">
            assertEquals(0, main.x);
            Debug.write('test complete');
         </method>
      </TestCase>
   </class>
</library>
```

We'll use a separate load.lzx file to organize our loaded libraries:

```
<script>
   loadedModules = new Array();
   loadedModules["testsuite"] = 0;
</script>

<import name="importUnitTest"
        href="testsuite.lzx"
        stage="defer">    ❶ Defers loading unit-testing library
   <handler name="onload">
      loadedModules["testsuite"] = 1;
      new unitTest(canvas, { name : "testsuite" });
      testsuite.run();    ❷ Starts unit testing
   </handler>
</import>
```

To ensure that our unit-testing library is loaded ❶ only when it has been explicitly requested, we set its `initstage` attribute to `defer`. The library is instantiated directly under the canvas with the name `testsuite` to conform to its original placement. Since `testsuite` is dynamically instantiated, it must be manually started with a `run` method ❷.

Although our unit-testing code module is small enough not to significantly impact our startup time, real-world unit-testing code would have a considerable impact on an application's startup and should always be dynamically loaded.

We'll next look at ways to measure the performance of the Laszlo Market.

18.4 Performance utilities

To optimize an application, we need a quantitative measurement of its performance. A set of base values forms a performance snapshot known as an *initial benchmark*. Benchmarks serve as a gauge to determine whether subsequent optimizing efforts yield significant improvements. An issue that arises with benchmarks is their validity and the limits of their accuracy. For instance, when a series of benchmarks is performed, it generates a distributed set of results. The distribution of these results is described through its mean, range, and standard deviation.

Before we start, it might be helpful to have a short review of some statistical terminology. The *mean* is the mathematical average of all the benchmark results. The *range* is the difference between the maximum and minimum value in these results. The *standard deviation* measures the statistical dispersion and indicates how tightly the values are clustered around the mean. When the standard deviation is large, this indicates that extremes on either side of the mean are canceling each other out and the result isn't statistically meaningful. If the standard deviation is large, this indicates that the benchmarking procedures are not returning reliable results and need to be reconsidered.

Different situations require different levels of accuracy. In the next sections, we'll start with a simple timing measurement demonstration, useful for rough development estimates, and progress to the Laszlo performance utilities designed for more accurate timing measurements.

18.4.1 Measuring time with getTime

In many development situations, a simple and quick timing measurement is needed to get a ballpark estimate of performance. The easiest way to get this type of information is to use JavaScript's `date` object and its `getTime` method.

The `getTime` method returns the number of milliseconds that have elapsed since January 1, 1970. Laszlo applications run in a single-threaded environment (both DHTML and the Flash player only use one thread for script execution), which means that we can use the difference of these timestamps to measure the time taken by any sequence of JavaScript calls. Here's an example using `getTime` to measure the amount of time to execute a loop:

```
<canvas>
   <handler name="oninit">
      measureTime(50000);
   </handler>
   <method name="measureTime" args="iters">
      <![CDATA[
      var t = (new Date()).getTime();
      for (var i = 0; i < iters;) { i++; }
      Debug.write((new Date()).getTime() - t + " ms");
      ]]>
   </method>
</canvas>
```

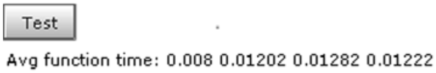
Although this provides useful information, it is important to understand its limitations. Code compiled for debugging runs slower than regular compiled code. To obtain more accurate timing information, you can add a timing framework to your application. This provides an adequate display and minimally impacts the application's performance. A timing framework is centralized, so measurement code doesn't need to be distributed throughout the application.

18.4.2 Building a simple timing framework

The following timing framework provides more flexibility by allowing any Laszlo object, method, or iteration count to be passed as an argument. Our benchmark consists of an object's increment method being executed 5,000 times:

```
<canvas>
    <simplelayout/>
    <button>Test
        <method event="onclick">
            TimingFramework.doTest(top, "increment", 5000);
        </method>
    </button>
    <node name="timingFramework">
        <method name="doTest" args="caller, method, reps">
            <![CDATA[
            var a = 0;
            var t = (new Date()).getTime();
            for (var i = reps; i >=0; i-- ){
                a = caller[method](a); }
            t = (new Date()).getTime() - t;
            report.addText(' ' + (t/reps));
            ]]>
        </method>
    </node>

    <view name="top">
         <method name="increment" args="a">
             return a + 1;
         </method>
    </view>
    <text name="report" resize="true">
        Avg function time:
    </text>
</canvas>
```

Our timing framework is executed whenever we click the Test button. Multiple executions can be performed and the results appear in sequential order on our screen (see figure 18.7). Then, we can enter these values into a calculator to determine the mean and standard deviation.



```
Avg function time: 0.008 0.01202 0.01282 0.01222
```

Figure 18.7 The timing framework lists a series of timing values.

But when you need the most accurate performance measurements to benchmark small-scale routines, you need to turn to the Laszlo performance utilities, which are covered in the next section.

18.4.3 Using the Laszlo performance utilities

The Laszlo performance utilities are the tools you should use when you need direct benchmarking comparisons between individual Flash and DHTML elements. They are most suitable for checking different optimization scenarios, such as the relative cost of using a function (approximately the cost of three assignment statements in Flash) or the cost of local versus global variables. Statistics are accumulated in two loops: a short-term loop to correct for the overhead of getting the time, and a long-term loop to minimize the perturbation due to background or other processes. To minimize outside interference, all measurements are performed in a JavaScript function during the script phase, before any of the Laszlo nodes have been instantiated. The long-term loop, by default, has 30 trials of 500 iterations:

```
<canvas>
   <include href="utils/performance"/>
   <script>
      var iterations = Measurement.defaultIterations;
      function empty () {
         for (var i = 0; i &lt; iterations; i++) {} }
      function measureIncrement () {
         var a = 0;
         for (var i = 0; i &lt; iterations; i++) {
            a = increment(a); } }
      function increment(a) {
           return a + 1; }
      (new Measurement({'empty': empty,
                        'Increment': measureIncrement})).run();
   </script>
</canvas>
```

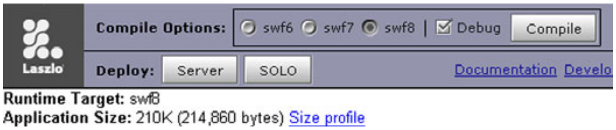
The `empty` function is created to account for the overhead of obtaining the time information:

```
iterations = 500
                       empty:    24.80us ±16.40    [0.00..60.00]/30
                   Increment:    78.73us ±37.12    [0.00..120.00]/30
```

After we correct and convert this value, the value from our previous timing framework is within the standard deviation. This deviation is expected, since times were obtained in different execution states. We isolated this value so it would be measured before the rest of the application is initialized, whereas the previous results were recorded in the midst of full application execution.

18.4.4 Using the developer console

The developer console for an application (see figure 18.8) appears at the bottom of the browser window. On the top line, it displays the uncompressed and gzipped sizes of an application. To the right is a link to the size profiler.



Compile Options: ○ swf6 ○ swf7 ● swf8 | ☑ Debug Compile

Deploy: Server SOLO

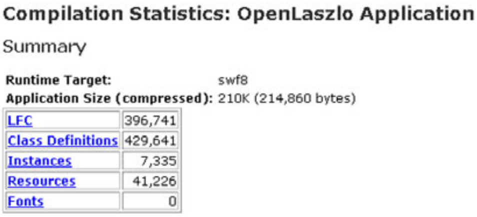
Runtime Target: swf8
Application Size: 210K (214,860 bytes) Size profile

Figure 18.8 The developer console contains uncompressed and gzipped application sizes, as well as a link to the size profiler.

The size profiler contains statistics for the uncompressed size of the application and is a useful tool for finding size-related hot spots in your application. The size profiler breaks down the application into the groupings displayed in figure 18.9. These values are the same whether the application is implemented with dynamic or statically linked libraries. Laszlo checks for the existence of a dynamic library during compilation and updates the statistics at that time.

Each of these major sections is further broken up to allow the size of individual class definitions, instances, resources, and fonts to be viewed. Size reports provide a quick way to check the application for unneeded classes or resources that were accidentally left behind from development.

Size profiles are generated by viewing the application with a ?lzt=info appended to the URL. There is a link at the bottom of the generated development page labeled "Size profile" that refers to compilation statistics.



Compilation Statistics: OpenLaszlo Application

Summary

Runtime Target: swf8
Application Size (compressed): 210K (214,860 bytes)

LFC	396,741
Class Definitions	429,641
Instances	7,335
Resources	41,226
Fonts	0

Figure 18.9 The size profiler maintains a record of sizes for all of the class definitions, instances, resources, and fonts.

18.5 Summary

We have seen how an application that was considered feature-complete was made ready for deployment on the Web. Being "made ready" involves two steps: making the application responsive enough to have a short startup time, and optimizing its overall performance. Laszlo possesses a rich set of tools to assist in completing these steps.

Dynamic libraries can be used to postpone the loading of noncritical objects on an "as-needed" basis. Modifications to an object's `initstage` attribute provide a means for redistributing its initialization time costs until a later period to shorten startup time. A wide array of benchmarking and profiling tools are provided to spotlight any potential performance bottlenecks. The result is that even large Laszlo applications can be optimized to start quickly.

This brings us to the end of the development of our Laszlo Market application. Its development started in chapter 5 with some rough scribbling on paper and continued throughout the book. It has served us well as a vehicle for illustrating the subject material for each chapter by providing concrete examples of their use and also for demonstrating stylistic concepts. Its incremental construction allowed features to be added in one chapter and later extended in other chapters. But now, this application's construction has completed, and this also marks the end of our book.

index

WEB DEVELOPMENT

LASZLO in Action

Norman Klein and Max Carlson with Glenn MacEwen

If you have not seen a Laszlo-driven site before, nor one of the Laszlo demos, you are missing a wonderful experience. Laszlo sites are continuous and flowing, often implemented as single-page apps that users can navigate with ease. Laszlo is a declarative, open-source programming language and environment that compiles to Flash, DHTML (i.e., AJAX), or Java/J2ME, with other platforms coming.

Written by an expert Laszlo developer together with a company founder, **Laszlo in Action** is and will remain the authoritative resource for this new technology. It gives you solid coverage of the Laszlo LZX language and then shows in very practical and complete terms how to use it. The authors develop a working example of an online store; they illustrate how to implement a site with Struts and Ruby on Rails; they show how to provide streaming video using Red5 server; and much more.

The book is written for web developers interested in creating improved, interactive internet applications.

Norman Klein, a former consultant with Laszlo Systems, has been a software engineer for over 20 years, and has been involved in internet development since its inception. A founder of Laszlo Systems, **Max Carlson** is a frequent public speaker, and the implementor of many of the features in the current 3.X OpenLaszlo release. **Glenn MacEwen** is a technical editor and writer living in Princeton, NJ.

"Every OpenLaszlo developer should have a copy."
—Ryan Stewart, Adobe

"The definitive source—I highly recommend it."
—Robi Sen, Twin Technologies LLC

"It bridges the gap between the documentation and a practical understanding of the platform."
—Geert Bevin, Uwyn

"... will help you quickly and efficiently develop the service-oriented UIs you'll need!"
—Matt Raible, Raible Designs, Inc.

"... opened my eyes to a new world of internet applications."
—Andrew Glover, Stelligent

"A definitive source for rich internet application developers."
—Edmon Begoli
Oak Ridge National Laboratory

For more information, code samples, and to purchase an ebook visit www.manning.com/LaszloinAction

ISBN-13: 978-1932394832
ISBN-10: 1932394834
54499
9 781932 394832

MANNING $44.99 / Can $44.99