Interactive Object Oriented Programming in Java

Learn and Test Your Skills

Vaskaran Sarcar

Apress®

Interactive Object Oriented Programming in Java

Learn and Test Your Skills

Vaskaran Sarcar

Apress®

Interactive Object Oriented Programming in Java: Learn and Test Your Skills

Vaskaran Sarcar
Bangalore, Karnataka, India

ISBN-13 (pbk): 978-1-4842-2543-1 ISBN-13 (electronic): 978-1-4842-2544-8
DOI 10.1007/978-1-4842-2544-8

Library of Congress Control Number: 2016962147

Managing Director: Welmoed Spahr
Lead Editor: Celestin Suresh John
Editorial Board: Steve Anglin, Pramila Balan, Laura Berendson, Aaron Black, Louise Corrigan, Jonathan Gennick, Robert Hutchinson, Celestin Suresh John, Nikhil Karkal, James Markham, Susan McDermott, Matthew Moodie, Natalie Pao, Gwenan Spearing
Coordinating Editor: Prachi Mehta
Copy Editor: Lori Jacobs
Compositor: SPi Global
Indexer: SPi Global
Artist: SPi Global

Distributed to the book trade worldwide by Springer Science+Business Media New York, 233 Spring Street, 6th Floor, New York, NY 10013. Phone 1-800-SPRINGER, fax (201) 348-4505, e-mail orders-ny@springer-sbm.com, or visit www.springeronline.com. Apress Media, LLC is a California LLC and the sole member (owner) is Springer Science + Business Media Finance Inc (SSBM Finance Inc). SSBM Finance Inc is a Delaware corporation.

For information on translations, please e-mail rights@apress.com, or visit www.apress.com.

Apress and friends of ED books may be purchased in bulk for academic, corporate, or promotional use. eBook versions and licenses are also available for most titles. For more information, reference our Special Bulk Sales–eBook Licensing web page at www.apress.com/bulk-sales.

Any source code or other supplementary materials referenced by the author in this text are available to readers at www.apress.com. For detailed information about how to locate your book's source code, go to www.apress.com/source-code/. Readers can also access source code at SpringerLink in the Supplementary Material section for each chapter.

Printed on acid-free paper

Contents at a Glance

Contents

About the Author

Vaskaran Sarcar completed his Master of Engineering in Software Engineering from Jadavpur University, Kolkata. He has 10+ years of teaching and industry experience. He began his career in teaching in various engineering colleges (2005-2007) and later shifted to the software industry. He is presently working as a Senior Software Engineer and Team Lead in a reputed R&D organization in India. He also received the MHRD-GATE Scholarship for the period 2007-2009. Reading and learning new things are passion for Vaskaran.

Other books by Vaskaran Sarcar are:

- Java Design Patterns (Apress, 2016).
- Design Patterns in C# (Computer Science Interview Series) (2015).
- C# Basics: Test Your Skill (2015).
- Operating System (Computer Science Interview Series) (2014).

About the Technical Reviewers

Shekhar Kumar Maravi is a System Software Engineer, whose main interest and expertise are Programming languages, Algorithms, and Data Structures. He obtained his M.Tech degree from Indian Institute of Technology, Bombay in Computer Science & Engineering. Since graduating, he has worked for Hewlett Packard India R&D Hub on Printer Firmware. Currently he is working as a Technical lead in InsightBitz. He can be reached by email at shekhar.maravi@gmail.com or you can find him on LinkedIn at: https://www.linkedin.com/in/shekharmaravi

Anupam Chakraborty is a Principal Software Engineer with more than 12 years of experience in developing object oriented software. He has obtained his Master degree in Computer Engineering from Indian Institute of Technology, Kharagpur& Bachelor degree from Indian Institute of Engineering Science and Technology, Shibpur.

Acknowledgments

My sincere thanks to my family, my friends, my great teachers, and all those individuals who supported this project directly or indirectly. Though it is my book, I believe it was only with the help of these extraordinary people that I was able to complete this work. Again, thanks to all of them who helped me to fulfill this project and motivate others in object-oriented programming in Java.

Introduction

Dear Reader,

Welcome to the journey. It is my privilege to present you with *Interactive Object-Oriented Programming in Java: Learn and Test Your Skills*. Before you jump into the topics, I want to highlight few points about the topic and its contents:

1. The aim of this book is to help you to get a feel of a Java classroom environment. I was involved in teaching since 2005. I have taken classes in both engineering and non-engineering colleges. And, fortunately, most of my teaching involvement was based on Java and its advanced topics. That is the true motivation to introduce a book like this.

2. This book will not discuss how to write an if-else statement or a simple while loop. Your teacher expects that before you attending the class, you have done your basic homework. Here your teacher will focus on the basic object-oriented concepts that we can implement in Java.

3. But before that, to assist you to ask better questions in the classroom, I dedicate an entire section at the beginning of the book to some key concepts in Java. These concepts will help you to evaluate your skills in the language basics. So, even if you are new to programming or you have some idea about some other programming languages, this section will be of great assistance. This section will also help you to prepare yourself for a job interview or a semester examination.

4. This book uniquely presents a two-way communication between teachers and students. With this book, you will have the feel of learning object-oriented programming in Java in a classroom environment—where your teacher will discuss some problems/topics, ask you questions, and give you assignments. You will be encouraged to do those simple assignments before beginning a new topic. If you are dedicated to this subject and do those assignments, you will surely develop confidence in this language.

5. In a semester, you need to attend a certain number of lectures to complete the fundamental topics, and we all know that learning is a continuous process. So, this book is not for those who want to learn Java in 24 hours or in 7 days. It is up to you only. I can only say that the book is designed for you in such a way that upon its completion, you will develop an adequate knowledge of the topic, you will learn the key features of this powerful language and object-oriented programming, and you will learn how we should write programs in Java and, most importantly, how to go further.

6. I have taken care to provide codes that are compatible with all the latest versions. Also, it is not mandatory for you to learn Eclipse. You can simply run these programs in your preferred IDE (integrated development environment). I have chosen Eclipse because it is widely used to develop Java applications.

7. No book can be completed without readers' feedback and supports. So, please share your comments to truly complete this book and enhance future work.

—The Author

Preface: Review the core terms and start the journey

In June 1991, James Gosling, Mike Sheridan and Patrick Naughton initiated the project of Java language. There was an Oak tree outside Gosling's office. And people say that due to the presence of that tree, originally the language was named Oak. Later they renamed the project as Green (Their team name was also Green team). And finally they renamed it to Java. The Green project was chartered by Sun Microsystem.

The team members wanted such a name that will be very much unique in nature and at the same time, it should reflect the essence of upcoming technologies .So, they picked up names like "Dynamic", "Revolutionary", "Silk", "Jolt", "DNA" etc.

James Gosling later told that Java was one of the top choices along with Silk. But finally they selected Java because most of the team mates liked this name.

Java became Open source on November13, 2006. Sun finished the process by making all of Java's core code available under free software/open-source distribution terms, (aside from a small portion of code to which they did not hold the copyright) on May 08, 2007.

Later Oracle Corporation purchased Sun Microsystem and the acquisition process was finished on January 27, 2010.

These qualities were the primary focus area for Java:

- Simple, object-oriented, and familiar programming style.
- Robustness and Security.
- Architecture-neutral and portability.
- High Performance capabilities.
- Interpreted, Threaded, and Dynamic.

Basic Terms

JVM

-It stands for Java Virtual Machine. When we compile the java file, we get a .class (not an .exe).This file contains java byte code which is interpreted by JVM. It is responsible for loading, verifying and executing the code .We say that JVM is platform dependent because it is responsible to convert the bytecodes into the machine language for the specific computer/machine.

JRE

-It stands for Java Runtime environment. It contains the JVM, the library files and the other supporting files. To run a java program, the JRE must be installed in the system. So, we can simply say JRE=JVM+ some packages.

JDK

-It stands for Java Development Kit. It provides the tool which we need to develop java programs and JRE. This tools contains javac.exe, java.exe etc. When we launch a java application, it will open the JRE and load the class and then, in turn, it will execute the main method. So, we can conclude that JDK=JRE+ Development tools.

Bytecode

-Bytecodes are machine language of the JVM. They provide the instruction set for a JVM. In other words, it is a virtual machine language in which java code is compiled. JVM comes into the picture because it stands between these bytecodes and our physical machine.

Platform

-We use the term platform to mean where the program will run. It can be your machine, your fully developed OS etc. When we say a language is platform independent, we mean that the code of a programmer will not vary across different platforms.

So once the java program is compiled, we get the bytecodes. These bytecode format is same for every platform (Windows/Linux/Solaris etc.).So, we need an interpreter who will interpret these bytecodes and then in turn he/she will produce the machine specific codes. Now JVM comes into the picture. Here in Java, these bytecodes are interpreted by JVM which is available for all OS. So, to port the java program into a new platform, we need to port the java interpreter.

So the pair -JVM and bytecode make Java portable.

Note: So the bottom line is that the trio- JVM, JRE and JDK are platform dependent (because of the OS dependence) but Java is platform independent.

We must remember the simple fact: Any machine language is dependent on the OS of the machine. So, if we have dependency on the machine specific OS, we are not platform independent. Java is platform independent because once the source code is compiled into standard bytecodes, those bytecodes are platform independent. Because of this facility Sun Microsystem is created the slogan WORA (Write Once Run Anywhere) for Java.

IDE

-It stands for Integrated Development Environment. They provide the facilities for software development. In general, they are very smart- they provide us intelligent code completion technique. They can also highlight/suggest about different kinds of possible fixes in our code. An IDE should have a source editor, a debugger and the automation tools to build the application. IDE's, in general, contain a compiler or an interpreter (or both). We have used eclipse here which contains both of these.

Installation

We need two major things.

1. JDK
2. IDE

[Please note that the following links may be altered in future. Till the writing of the book ,these are working fine.]

Visit the page:

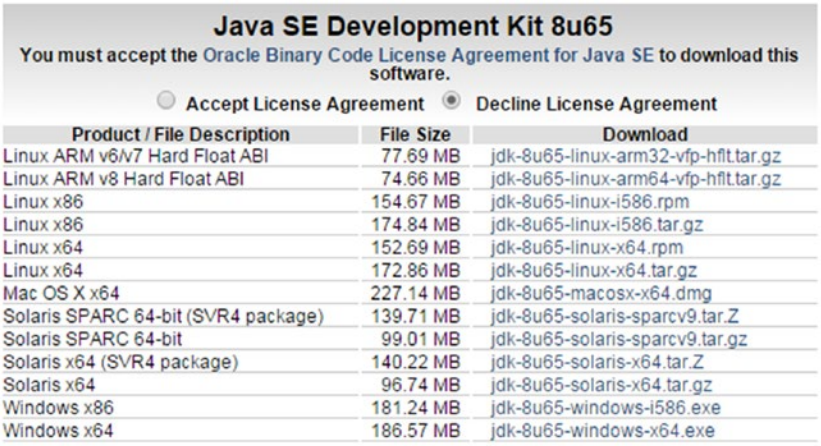
`http://www.oracle.com/technetwork/java/javase/downloads/index.html`

Or

To download JDK, directly go here:

http://www.oracle.com/technetwork/java/javase/downloads/jdk8-downloads-2133151.html

You'll get a screen containing something like this:



Java SE Development Kit 8u65

You must accept the Oracle Binary Code License Agreement for Java SE to download this software.

○ Accept License Agreement ◉ Decline License Agreement

Product / File Description	File Size	Download
Linux ARM v6/v7 Hard Float ABI	77.69 MB	jdk-8u65-linux-arm32-vfp-hflt.tar.gz
Linux ARM v8 Hard Float ABI	74.66 MB	jdk-8u65-linux-arm64-vfp-hflt.tar.gz
Linux x86	154.67 MB	jdk-8u65-linux-i586.rpm
Linux x86	174.84 MB	jdk-8u65-linux-i586.tar.gz
Linux x64	152.69 MB	jdk-8u65-linux-x64.rpm
Linux x64	172.86 MB	jdk-8u65-linux-x64.tar.gz
Mac OS X x64	227.14 MB	jdk-8u65-macosx-x64.dmg
Solaris SPARC 64-bit (SVR4 package)	139.71 MB	jdk-8u65-solaris-sparcv9.tar.Z
Solaris SPARC 64-bit	99.01 MB	jdk-8u65-solaris-sparcv9.tar.gz
Solaris x64 (SVR4 package)	140.22 MB	jdk-8u65-solaris-x64.tar.Z
Solaris x64	96.74 MB	jdk-8u65-solaris-x64.tar.gz
Windows x86	181.24 MB	jdk-8u65-windows-i586.exe
Windows x64	186.57 MB	jdk-8u65-windows-x64.exe

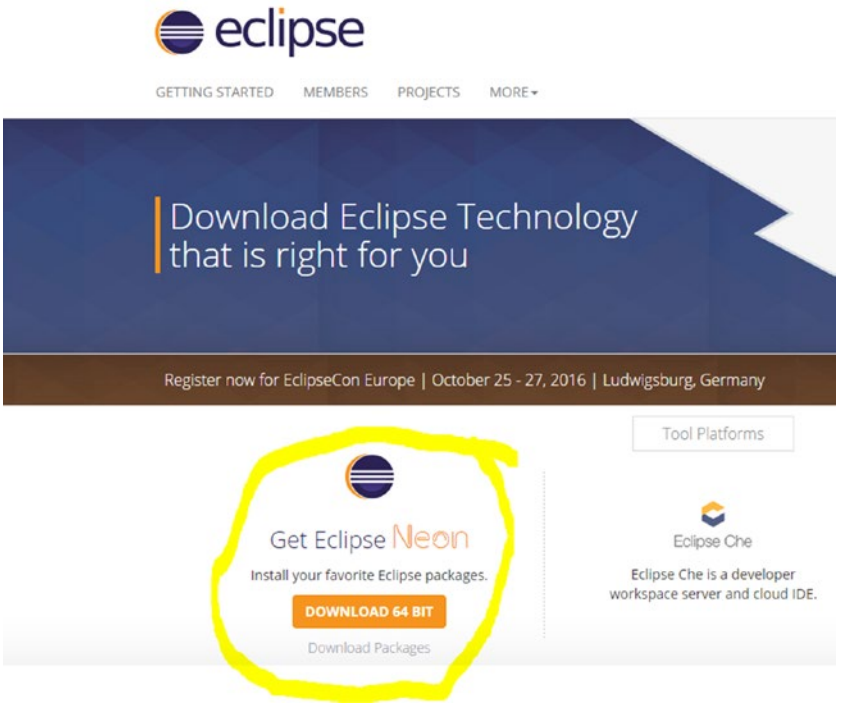
Try to download the latest version based on your system configuration (e.g. 32 bit/64 bit, Windows/Linux etc.)

To download eclipse IDE:

Go here.

https://eclipse.org/downloads/

As mentioned above, try to download the latest version based on your system configuration e.g. you can see something like this:



NAMING CONVENTIONS

For those familiar with Java, these are the naming conventions I'll follow in this book:

- Class- They should start with an uppercase letter and should be a noun e.g. `MyClass`, `String` etc.
- Interface/s- They should start with an uppercase letter and should be an adjective e.g. `Runnable`, `Remote`
- Method/s- They should start with a lowercase letter and be a verb e.g. `main()`, `showMyMethod()`, etc.
- Variable/s- They should start with lowercase letter e.g. `myIntegerValue`, `myDoubleValue`, `myName` etc.
- Package/s- They should be all in lower case latter e.g. `mypackage`, `package1` etc.
- Constants- They should be in uppercase letters e.g. `MY_CONSTANT` etc.

Apart for few special cases, we have tried to maintain these conventions across the book.

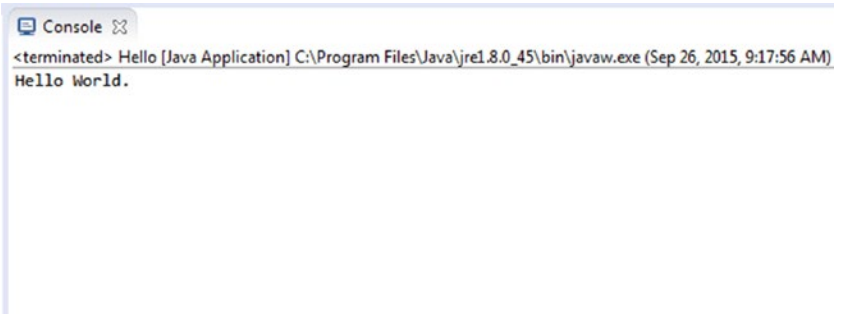
Our First Program

Now let us go through our first program and follow the analysis section carefully.

```
package javaclassnotes.programs;

public class HelloWorld
{
        public static void main(String args[])
        {
                System.out.println("Hello World.");
        }
}
```

Output

```
Hello World.
```



Explanation

First of all, throughout the book, we have organized the programs into package/s. But for this program, it was not mandatory. Once we go through the chapter on package, it will be clear to us.

This is the basic structure of the main method. The meaning and significance of each keyword will be clear to you gradually. So, for the time being, you must follow this structure.

Our source file name is `HelloWorld.java`. We need to use `.java` extension for our java files. It is the requirement for the compiler.

Java is case-sensitive.

Here are some key points which we need to remember:

- `main`- The program will start from here. It is a method. A method is basically a set of statements grouped together with the curly braces. (For the time being, you can think it as a function or procedure).
- `public`- The access specifier. Access specifiers are used to control the visibility of the members.

- `static`- It allows us to call `main()` without instantiating a particular instance of the class. We'll do detailed analysis on the keyword `static` later.
- `void`-return type of the method.
- `String args[]`-`args` is the name chosen for this String array. String arrays are used to store character strings. The name `args` is chosen to represent arguments to the method.
- `println`-It is used to display information.
- `System.out`- Difficult to explain at this point. Just we can know that `System` is a class and `out` is output stream associated with the console.

CHAPTER 1

Test your skill in language fundamentals

Now go through some fundamental concepts. To make an impression in your brain, these fundamental concepts are discussed through some programs (and their corresponding outputs) and with some Q&A. Author suggests you to go through each of them before you enter into your classroom. This section is added to help you to understand the discussions in the classroom better. Author also believes that you can ask better questions to your teacher, once you brush up these fundamentals in the Java language before you enter into the classroom.

SET 1

What will be the output?

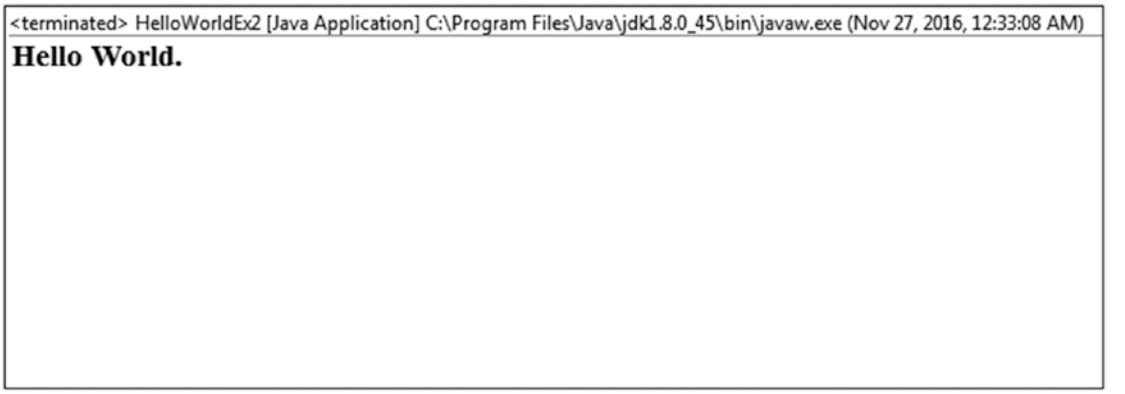
```
package fundamentals;
public class HelloWorldEx2
{
        static public void main(String args[])
        {
                System.out.println("Hello World.");
        }
}
```

Electronic supplementary material The online version of this chapter (doi:10.1007/978-1-4842-2544-8_1) contains supplementary material, which is available to authorized users.

V. Sarcar, *Interactive Object Oriented Programming in Java*, DOI 10.1007/978-1-4842-2544-8_1

Output:

Hello World.

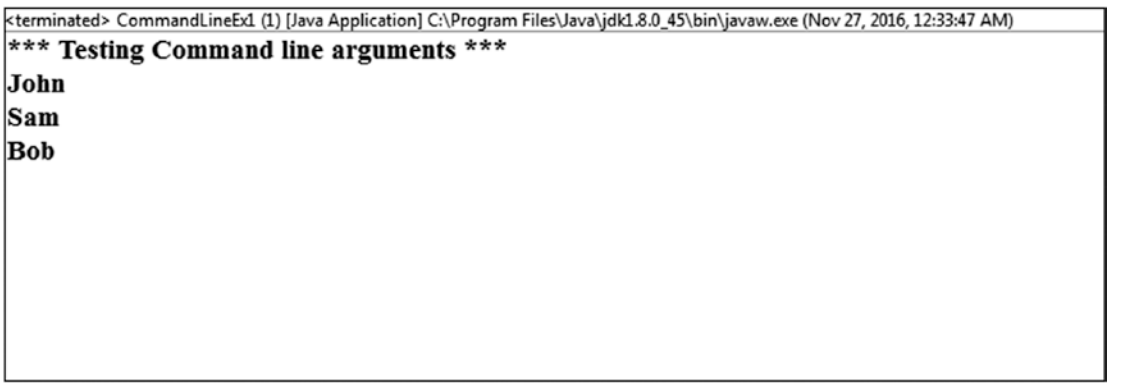


So, we can see that we can change the order. Instead of public static void main(…), we can write: static public void main(…). But we'll always use the convention inside this book.

What will be the output if we pass some arguments through command line (e.g. *java CommandlIneEx1 John Sam Bob*)?

```
package fundamentals;
public class CommandLineEx1 {
        public static void main(String args[])
        {
                System.out.println("*** Testing Command line arguments ***");
                //java CommandlIneEx1 John Sam Bob
                System.out.println(args[0]);
                System.out.println(args[1]);
                System.out.println(args[2]);
        }
}
```

Output:



Explanation:

When we use command line arguments like:

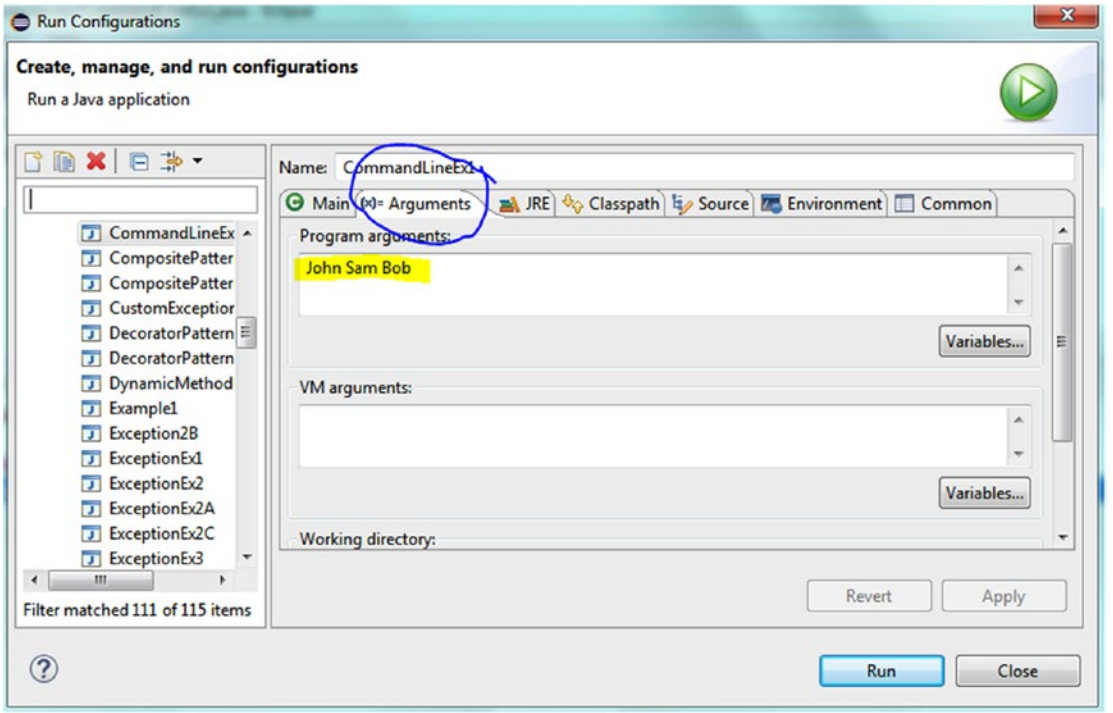
java <your program name> a0 a1 a2 ...

values will be assigned in the String array ***arg*** as arg[0]=a0, arg[1]=a1, arg[2]=a2 etc. *Also note that you can choose any name you want for your String array*. e.g.

```
public static void main(String myStringArray[])
        {...}
```

is perfectly fine.

For eclipse users, you can enter arguments in the Arguments tab under Run Configurations like this:



Do we need a main() for each of the class?

No. The class with a main() method is used to denote the starting point of your application. So, other classes can exist without a main() method.

SET 2

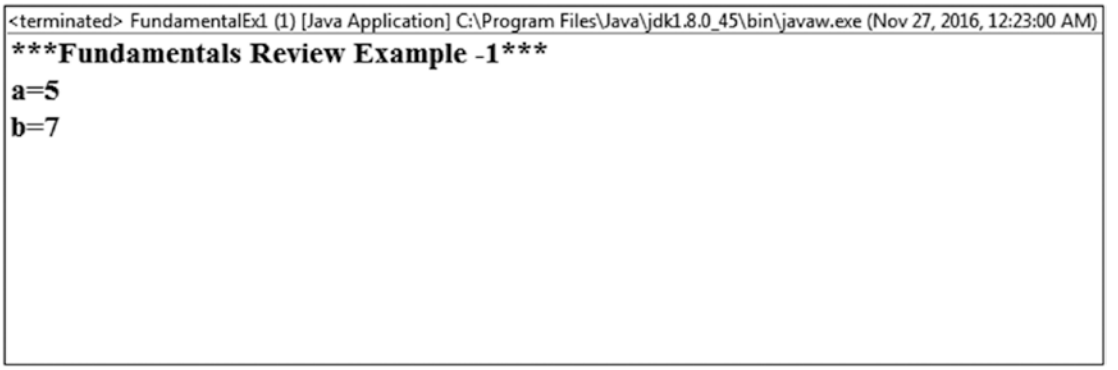
What will be the output?

```
package fundamentals;
public class FundamentalEx1 {
        public static void main(String args[])
        {
                System.out.println("***Fundamentals Review Example -1***");
```

```
                int a=5;//ok
                int b=07;//ok
                System.out.println("a="+a );
                System.out.println("b="+b);
        }
}
```

Output:

a=5 b=7

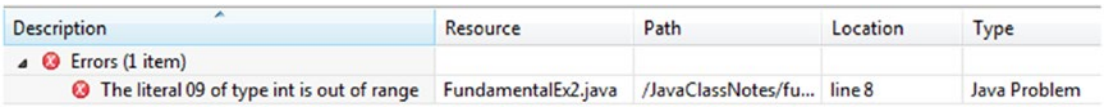


What will be the output?

```
package fundamentals;
public class FundamentalEx2 {
        public static void main(String args[])
        {
                System.out.println("***Fundamentals Review Example -2***");
                int a=5;//ok
                int b=07;//ok
                int c=09;//Error
                System.out.println("a="+a );
                System.out.println("b="+b);
                System.out.println("c="+c);
        }
}
```

Output:

Compilation error: The literal 09 is out of range.



Explanation:

When we put a leading 0, Java treats it as an octal representation. So, in that case, the range it can support is 0 to 7. In this example, we have crossed that boundary. To print 8 , you have to code like:

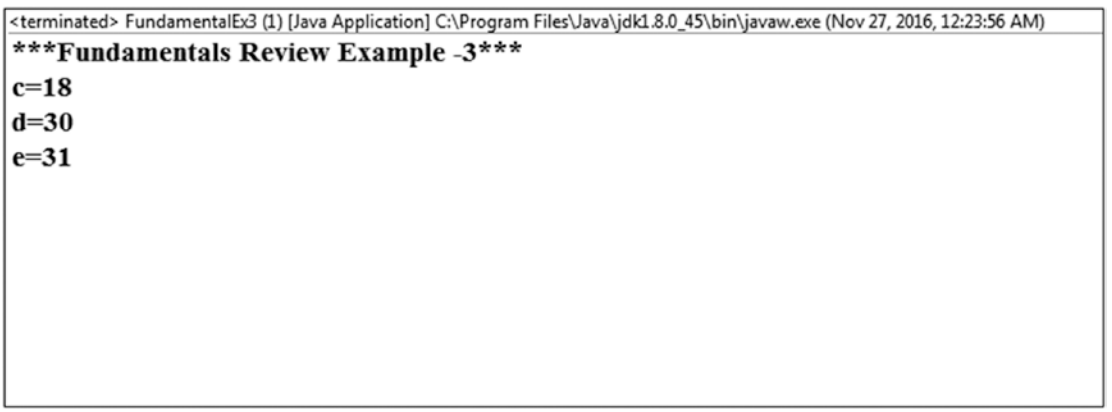
```
int d=010;//ok. It will print 8
System.out.println("d="+d );
```

What will be the output?

```
package fundamentals;

public class FundamentalEx3 {
        public static void main(String args[])
        {
                System.out.println("***Fundamentals Review Example -3***");
                int c=0x12;//ok, will print 18
                int d=0x1E;//ok, will print 30
                int e=0X1F;//ok, will print 31
                System.out.println("c="+c);//18
                System.out.println("d="+d);//30
                System.out.println("e="+e);//31
        }
}
```

Output:

c=18 d=30 e=31



Explanation:

When we prefix 0x or 0X, Java treats them as a hexadecimal integer literal representation. So, in this case, the range it can support is 0 to 15. A to F is used to represent the digits with values 10 to 15.

What will be the output?

```
package fundamentals;

public class FundamentalEx4
{
        public static void main(String args[])
        {
                System.out.println("***Fundamentals Review Example -4***");
                int a=5;
                double const=3.14;//Error
                System.out.println("const value is ="+const);//Error
        }
}
```

Output:

Compilation error.

Description	Resource	Path	Location	Type
Errors (2 items)				
Syntax error on token "const", invalid Expression	FundamentalEx4.java	/JavaCl...	line 10	Java Problem
Syntax error on token "const", invalid VariableDeclaratorId	FundamentalEx4.java	/JavaCl...	line 9	Java Problem

Explanation:

Some words are explicitly reserved for used by the Java languages only. These are reserved keywords in java. In the above example, we have being used such a keyword *const* like a variable and hence encountered this issue.

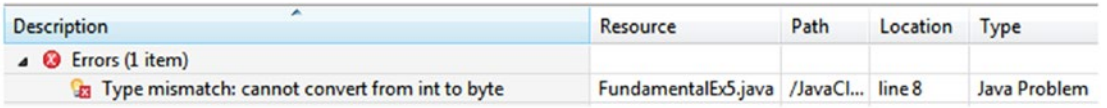
SET 3

What will be the output?

```
package fundamentals;

public class FundamentalEx5 {
        public static void main(String args[])
        {
                System.out.println("***Fundamentals Review Example -5***");
                byte b1=127;//ok
                byte b2=128;//Error
                System.out.println("b1="+b1);
                System.out.println("b2="+b2);
        }
}
```

Output:

Compilation error.



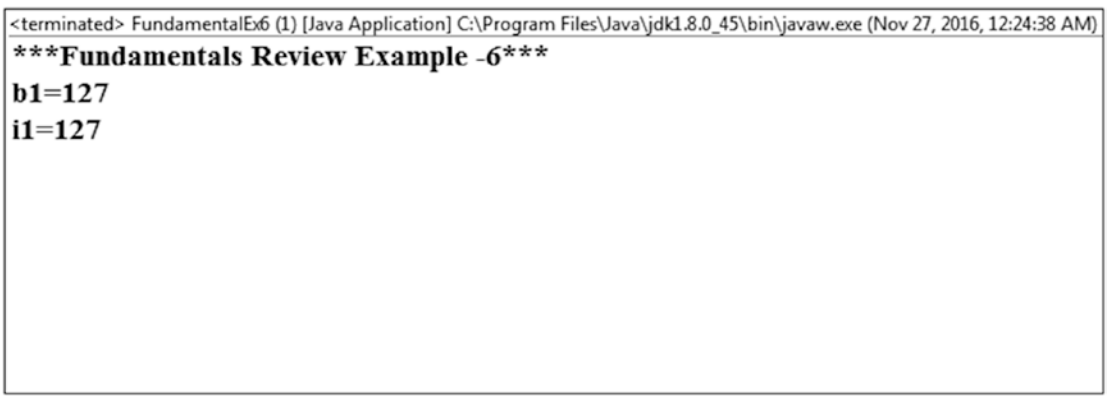
Explanation:
Range of byte is -128 to 127.

What will be the output?

```
package fundamentals;

public class FundamentalEx6 {
        public static void main(String args[])
        {
                System.out.println("***Fundamentals Review Example -6***");
                byte b1=127;//ok
                int i1=b1;//ok
                System.out.println("b1="+b1);
                System.out.println("i1="+i1);
        }
}
```

Output:

b1=127 i1=127



Explanation:
The two types- int and byte are compatible and here we are putting the byte into an int means that the destination type is larger than source type. So, compiler is ok with this conversion.

What will be the output?

```
package fundamentals;

public class FundamentalEx7 {
        public static void main(String args[])
        {
                System.out.println("***Fundamentals Review Example -7***");
                byte b1=127;
                int i1=b1;//ok: small to big
                b1=i1;//Error: big to small
                System.out.println("b1="+b1);
                System.out.println("i1="+i1);
        }
}
```

Output:

Compilation error.

Description	Resource	Path	Location	Type
Errors (1 item)				
Type mismatch: cannot convert from int to byte	FundamentalEx7.java	/JavaCl...	line 9	Java Problem

Explanation:
Here the destination type is smaller than source type. So, compiler is raising the concern.

SET 4

What will be the output?

```
package fundamentals;

public class FundamentalEx8 {
        public static void main(String args[])
        {
          System.out.println("***Fundamentals Review Example -8***");
                int i=2147483647;
                System.out.println("i="+i);
                int j=++i;
                System.out.println("Now i is="+i);
                System.out.println("j="+j);
        }
}
```

Output:

```
<terminated> FundamentalEx8 [Java Application] C:\Program Files\Java\jdk1.8.0_45\bin\javaw.exe (Nov 26, 2016, 11:50:05 PM)
***Fundamentals Review Example -8***
i=2147483647
Now i is=-2147483648
j=-2147483648
```

Explanation:

The maximum value of integer is 2,147,483,647 and the minimum value is -2,147,483,648. Here in j (with post increment of i), we have crossed the maximum limit of an integer.

What will be the output?

```
package fundamentals;
public class FundamentalEx9
 {
        public static void main(String args[])
        {
        System.out.println("***Fundamentals Review Example -9***");

        int i=5;
        int j=i++;//j becomes 5, i becomes 6
        System.out.println("j now="+j);
        System.out.println("i now="+i);
        int k=++j;//j and k both becomes 6
        System.out.println("j="+j);
        System.out.println("k="+k);
        }
}
```

Output:

The program will compile and run successfully.

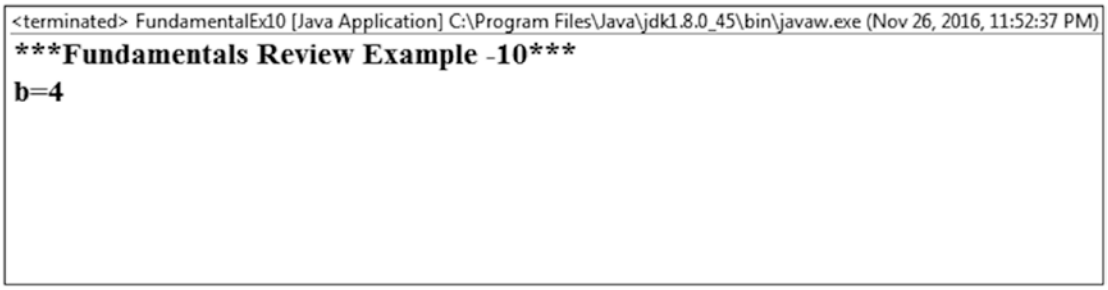
```
<terminated> FundamentalEx9 [Java Application] C:\Program Files\Java\jdk1.8.0_45\bin\javaw.exe (Nov 26, 2016, 11:51:55 PM)
***Fundamentals Review Example -9***
j now=5
i now=6
j=6
k=6
```

What will be the output?

```
package fundamentals;
public class FundamentalEx10 {
        public static void main(String args[])
        {
        System.out.println("***Fundamentals Review Example -10***");
        int i=260;
        byte b=(byte) i;
        System.out.println("b="+b);
        }
}
```

Output:

b=4



Explanation:

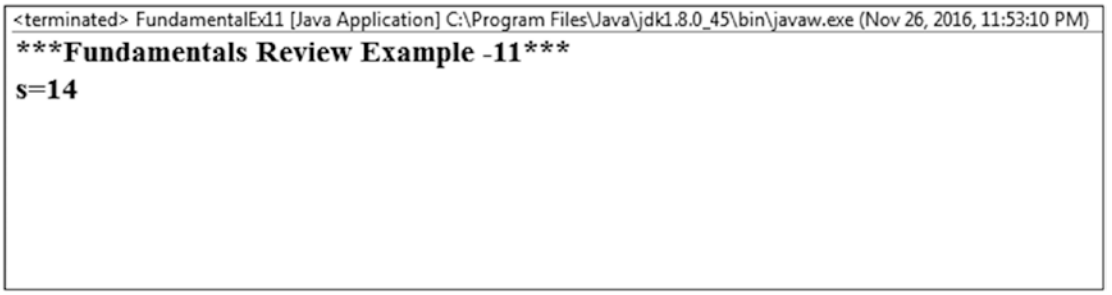
Here we are trying to type cast a larger variable(int) into a smaller variable(byte). So, in this type of case, java calculates modulo of larger variable by the range of smaller variable. Our byte range is *-128 to 127*. So final result would be *260 % 256* i.e. 4

What will be the output?

```
package fundamentals;

public class FundamentalEx11 {
        public static void main(String args[])
        {
        System.out.println("***Fundamentals Review Example -11***");
        int i=65550;
        short s=(short)i;
        System.out.println("s="+s);
        }
}
```

Output:

s=14



Explanation:

See the above explanation. The range for short datatype is: *-32768 to 32767* i.e. total ranges it covers is 65536. So here the result is: 65550 % 65536=14

What will be the output?

```
package fundamentals;

public class FundamentalEx12 {
        public static void main(String args[])
        {
        System.out.println("***Fundamentals Review Example -12***");
        char c1=65;
        char c2='a'+3;
        System.out.println("c1="+c1);
        System.out.println("c2="+c2);
        }
}
```

Output:

c1=A c2=d

Explanation:
ASCII value of A is 65 and ASCII value of a is 97.
97+3=100 which is the ASCII value of d.

SET 5

Why do you prefer double over float?

double is used for double precision, *float* is for single precision. So, to maintain the accuracy of calculation, double is a better choice over float.

What is the difference between char in Java vs char in C/C++?

In C/C++, char is an integer type (8 bit wide). But Java uses Unicode (UTF-16) to represent them. In Java, char is of 16 bit type.

What do we mean by Unicode?

Unicode defines a fully international character set that can be found in world's most human languages/ writing systems. So, it is a unification of all those character sets. This allows us to encode, represent and handling texts in those languages in a standard way.

What do we mean automatic type conversion?

Two basic criteria must be followed for an automatic conversion.

- *The destination type should be larger.*
- *The types are compatible.*

e.g. following conversion is automatic:

```
int i1=15;
double d=i1;//ok
System.out.println("d="+d);//15.0
```

But below conversion is not allowed:

```
boolean b=true;
int i2=b;//error
```

Why does Java not convert primitive types to objects?

Unnecessary overhead will be created due to these conversions and Java may lose the efficiency in performance.

Why do all Java primitive types have a fixed range?

To support portability, Java supports this concept.

What do we mean by the word portability?

In simple language, suppose you have developed an application in a machine. Now you want to reuse it in other environment (e.g. in different hardware/software platforms or versions or different operating systems etc) without a major rework (in ideal scenario: no rework).If you can do that you can claim that your application is portable

We also remember that the pair -JVM and bytecode make Java portable. Go through the *"Basic Terms"* explained previously to revise your concepts.

SET 6

What will be the output?

```
package fundamentals;
public class FundamentalEx13
{
  public static void main(String args[])
 {
   System.out.println("***Fundamentals Review Example -13***");
   int x = 10;
   int result=++x*5;
  System.out.println(" The result is : "+ result);
 }
}
```

Output:

```
<terminated> FundamentalEx13 [Java Application] C:\Program Files\Java\jdk1.8.0_45\bin\javaw.exe (Nov 26, 2016, 11:57:03 PM)
***Fundamentals Review Example -13***
 The result is : 55
```

Explanation:

See Table 1-1 below. ++ has higher precedence than *. So, ++x will be evaluated first and then the result will be multiplied.

Table 1-1. *Operator Precedence Table (highest to lowest):*

Postfix	*expr*++ *expr*--
Unary	++*expr* --*expr* +*expr* -*expr* ~ !
Multiplicative	* / %
Additive	- +
Shift	>>> << >>
Relational	<= >= <>instanceof
Equality	== !=
Bitwise AND	&

(*continued*)

Table 1-1. (*continued*)

Bitwise exclusive OR (XOR)	^
Bitwise OR(inclusive)	|
Logical AND	&&
Logical OR	||
Ternary	? :
Assignment	= += -= *= /= %= &= ^= |= <<= >>= >>>=

Other points to remember:

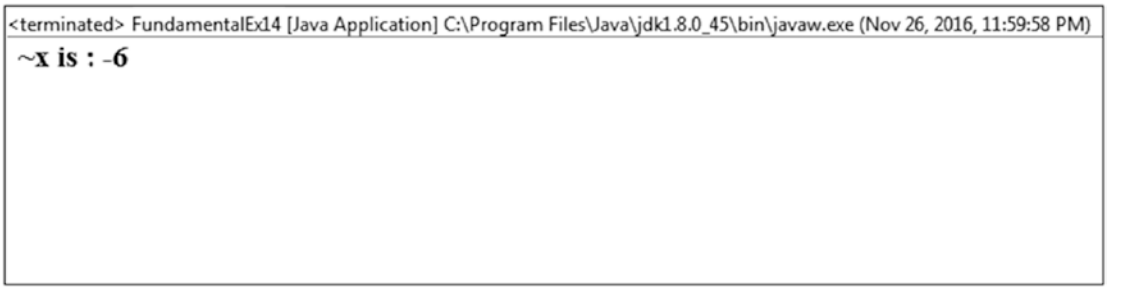
- Only assignment operators are evaluated right to left.
- All other binary operators are evaluated in reverse direction (i.e. left to right).
- (), [], dot operators (.) have the highest precedence. *These are called separators* but they act like operators in an expression. Basically to alter the precedence of an operation, we need to use parentheses.

What will be the output?

```
package fundamentals;
public class FundamentalEx14 {
        public static void main(String args[])
        {
                int x=5;
                System.out.println(" ~x is : "+ ~x);
        }
}
```

Output:

~x is: -6



Explanation:

5 is represented by 0000 0101. ~5 will make it 1111 1010. Which is for -6.

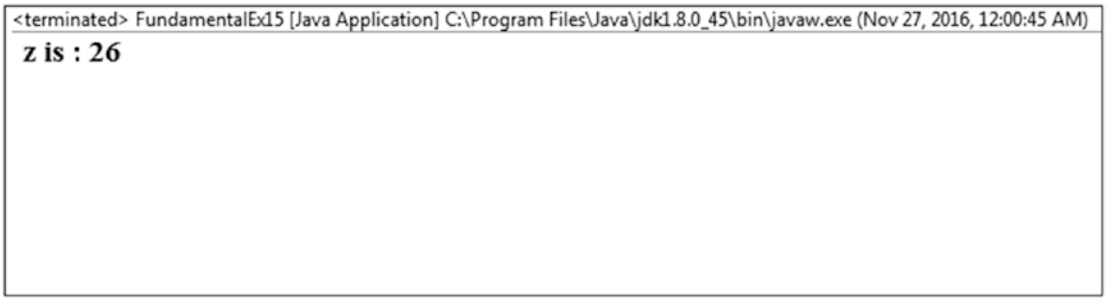
To understand it better, represent 6 in binary: 0000 0110. Now 2's complement of it (inverting all bits first and then add a 1 to that) will make it 1111 1010.

What will be the output?

```
package fundamentals;
public class FundamentalEx15 {
        public static void main(String args[])
        {
                int x=21;
                int y=15;
                int z=x^y;
                System.out.println(" z is : "+ z);
        }
}
```

Output:

z is :26



Explanation:

- 21 in binary is 0001 0101
- 15 in binary is 0000 1111

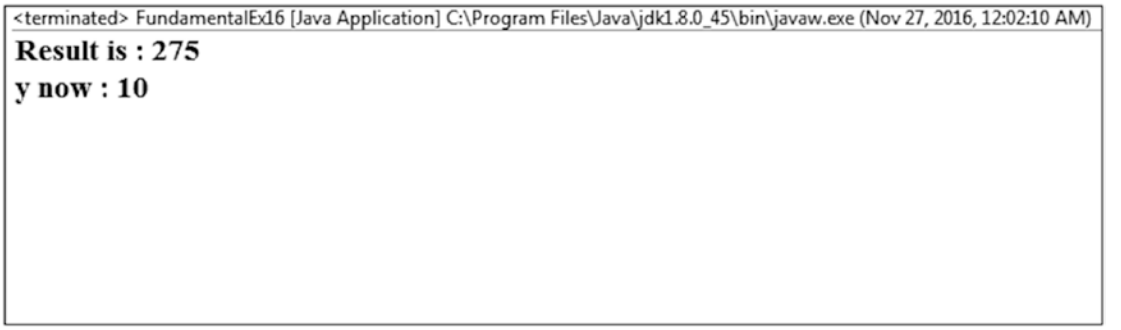
XOR combines bits with the rule: if exactly one operand is 1 then the result is 1. So our result becomes: 0001 1010 i.e 26

What will be the output?

```
package fundamentals;
public class FundamentalEx16 {
        public static void main(String args[])
        {
                int x=24;
                int y=11;
                int result= ++x * y--;
                System.out.println("Result is : "+ result);
                System.out.println("y now : "+ y);
        }
}
```

Output:

Result is :275
y now: 10



Explanation:
++x =25, 25*y--=275 (After this operation y will be decremented)

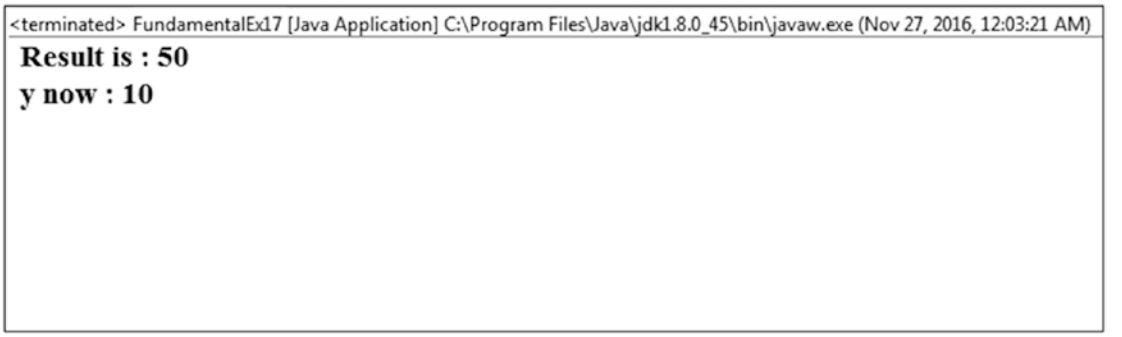
What will be the output?

```
package fundamentals;
public class FundamentalEx17 {
        public static void main(String args[])
        {
                int x=24;
                int y=11;
                int z=100;
                //int result= ++x * y--;//275
                int result= ++x *--y %z;
                System.out.println(" Result is : "+ result);
                System.out.println(" y now : "+ y);
        }
}
```

Output:

Result is: 50

y now:10



Explanation:

Pre increment happened to x and pre decrement happened to y before the multiplication operation which results 250. Finally the modulo operation is resulting 50. (250%100=50).

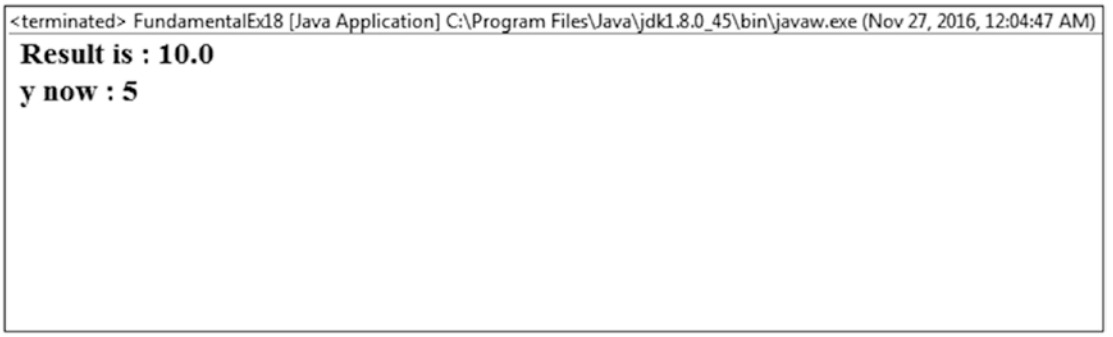
What will be the output?

```
package fundamentals;
public class FundamentalEx18 {
        public static void main(String args[])
        {
                int x=10;
                int y=4;
                double result= ++y*x/y;
                System.out.println(" Result is : "+ result);
                System.out.println(" y now : "+ y);
        }
}
```

Output:

Result is :10.0

y now: 5



Explanation:

We must notice that y incremented first and becomes 5. So, *5*10/5 becomes 10.0 because we are storing the result in a double datatype.*

What will be the output?

```
package fundamentals;

public class FundamentalEx19 {
        public static void main(String args[])
        {
                System.out.println("***Fundamentals Review Example -19***");
                int a=7,b=12;
                System.out.println(a+b);//19
                System.out.println("a+b=" +a+b);//a+b=712
                System.out.println(a+b+"=a+b=" +a+b);//19=a+b=712
        }
}
```

Output:

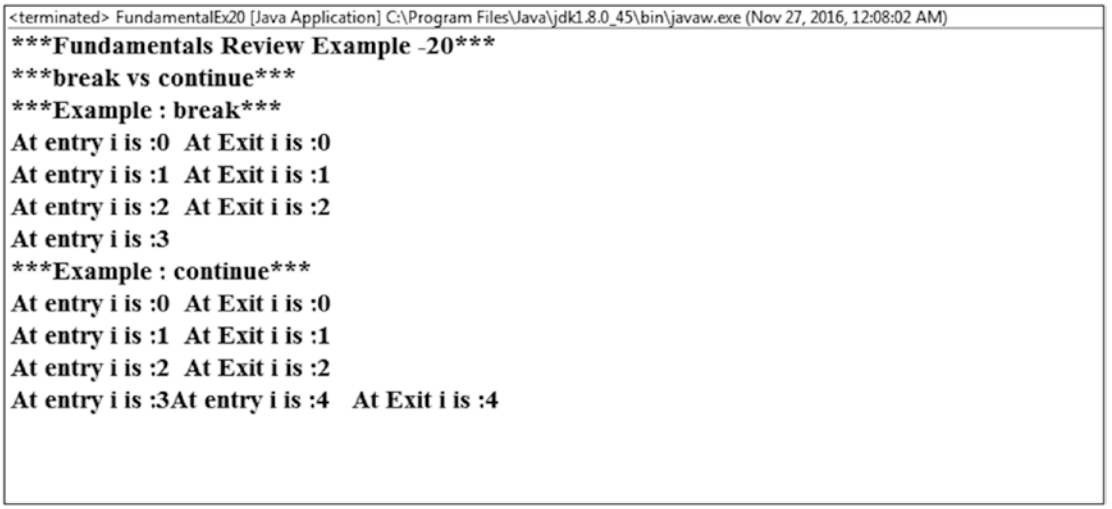
Explanation:

You must note this behavior: once the string is encountered, we started seeing the concatenation instead of addition. That's why initially a and b is added and resulted 19 but after that it encountered a string "a+b", so now onwards it will start string concatenation operations.

Differentiate: break vs continue with an example.

```
package fundamentals;

public class FundamentalEx20 {
        public static void main(String args[])
        {
                System.out.println("***Fundamentals Review Example -20***");
                System.out.println("***break vs continue***");
                System.out.println("***Example : break***");
                for(int i=0;i<5;i++)
                {
                        System.out.print("At entry i is :"+i);
                        if( i==3)
                                break;
                        System.out.print("\t At Exit i is :"+i);
                        System.out.println();
                }
                System.out.println();
                System.out.print("***Example : continue***\n");
                for(int i=0;i<5;i++)
                {
                        System.out.print("At entry i is :"+i);
                        if( i==3)
                                continue;
                        System.out.print("\t At Exit i is :"+i);
                        System.out.println();
                }
        }
}
```

Output:



Explanation:

From the above code, we can see that once we encounter break (at i=3), control has come out from the for loop block but in case of continue, it just skipped remaining portion for that iteration (i.e. did not print Exit statement for i=3) and continued looping to the end.

Oracle Java documentation says: break can have two forms-labeled and unlabeled. An unlabeled break statement can terminate the innermost for, while, do-while, switch statements, but a labeled break can terminate an outer statement.

What will be the output?

```
package fundamentals;
public class ConditionalOperatorEx {
        public static void main(String args[])
        {
                System.out.println("*** Conditional Operator Demo***");
                int a=10;
                int b=5;
                int c=a>b?a:b;
                System.out.println("c is : "+c);
        }
}
```

Output:

```
<terminated> ConditionalOperatorEx (1) [Java Application] C:\Program Files\Java\jdk1.8.0_45\bin\javaw.exe (Nov 27, 2016, 12:09:30 AM)
*** Conditional Operator Demo***
c is : 10
```

Explanation:
This is a very common use of the conditional operator.

What will be the output?

```
package fundamentals;
public class ConditionalOperatorEx2 {
        public static void main(String args[])
        {
        System.out.println("*** ConditionalOperator Demo-2***");
                int a=10;
                int b=5;
                //Ex5.10
                //Error:Type mismatch
                     String result=a<0?"Negative":a;
                System.out.println("result is : "+result);
        }
}
```

Output:

Compilation Error.

Description	Resource	Path
Errors (1 item)		
Type mismatch: cannot convert from int to String	ConditionalOperator...	/JavaClassNotes/ja...

Explanation:
You cannot put an integer inside a string. So, be careful for the following type of comparison:
expression1? expression2: expression3
expression 2 and expression3 must be same.

SET 7

What is an array?

It is container object that can hold a fixed number of a particular type.

Show an example of creating an array and displaying the contents inside it.

```
package fundamentals;
public class ArrayEx1
{
        public static void main(String args[])
        {
                System.out.println("***Fundamentals Review Examples -Arrays***");
                int[] myIntArray=new int[5];
                for(int i=0;i<5;i++)
                {
                        myIntArray[i]=i*10;
                }
                System.out.println("Contents of Array:");
                for(int i=0;i<5;i++)
                {
                        System.out.print("\t"+myIntArray[i]);
                }
        }
}
```

Output:

```
<terminated> ArrayEx1 (1) [Java Application] C:\Program Files\Java\jdk1.8.0_45\bin\javaw.exe (Nov 27, 2016, 12:10:27 AM)
***Fundamentals Review Examples -Arrays***
Contents of Array:
        0       10      20      30      40
```

Explanation:

Here we have created an array which can hold 5 integers only (notice the declaration new int[5]).

In the above program suppose you have used the following declaration:

```
int myIntArray[]=new int[5];
```

Is it a valid declaration?

Yes. We can use either form : *int[] myArray* or *int myArray[]*.

How can you alter the size of an array?

Once created, we cannot alter the size of it, it is fixed then.

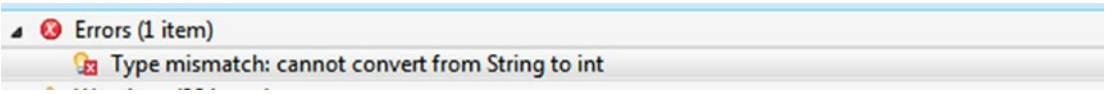
Can you shorten the code size in the example 6.2.
We can directly initialize the array like this:

```
int  myIntArray[]={0,10,20,30,40};//ok
```

Can you compile the code below?

```
int[] myIntArray=new int[3];
myIntArray[0]=10;
myIntArray[2]=20;
myIntArray[3]="Thirty";
```

No. We cannot put a string into an integer array.



Will the code compile?

```
package fundamentals;

public class ArrayEx2
{
        public static void main(String args[])
        {
                System.out.println("***Fundamentals Review Examples -Arrays***");
                int[] myIntArray=new int[3];
                myIntArray[0]=10;
                myIntArray[2]=20;
                System.out.println("Contents of Array:");
                for(int i=0;i<5;i++)
                {
                        System.out.print("\t"+myIntArray[i]);
                }
        }
}
```

Answer:
Yes. *There is no compilation error but we'll encounter a runtime exception* as we are trying to access locations beyond the boundary (our array size is 3 but we are trying to access indexes more than index 2). We'll discuss on exceptions later in this book.

```
<terminated> ArrayEx2 (1) [Java Application] C:\Program Files\Java\jdk1.8.0_45\bin\javaw.exe (Nov 27, 2016, 12:11:38 AM)
***Fundamentals Review Examples -Arrays***
Contents of Array:
        10    0     20Exception in thread "main" java.lang.ArrayIndexOutOfBoundsException: 3
        at fundamentals.ArrayEx2.main(ArrayEx2.java:14)
```

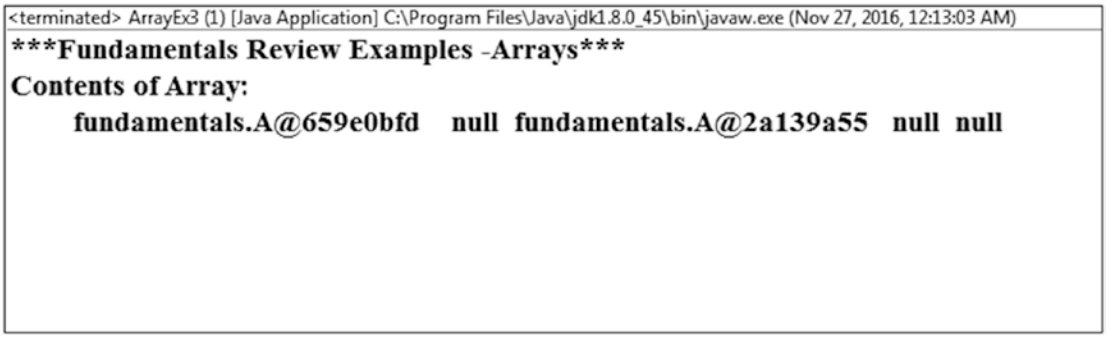
In the above output, we can see MyIntArray[1] is printed as 0 but we have not supplied 0 in it. Is this array initialized with default values?

Yes. Default value for integers is 0.

What will be the output?

```
package fundamentals;
class A
{
        int i;
        A(int i)
        {
                this.i=i;
        }
}
public class ArrayEx3
{
        public static void main(String args[])
        {
                System.out.println("***Fundamentals Review Examples -Arrays***");
                A[] myArray=new A[5];
                myArray[0]=new A(10);
                myArray[2]=new A(25);
                System.out.println("Contents of Array:");
                for(int i=0;i<5;i++)
                {
                        System.out.print("\t"+myArray[i]);
                }
        }
}
```

Output:



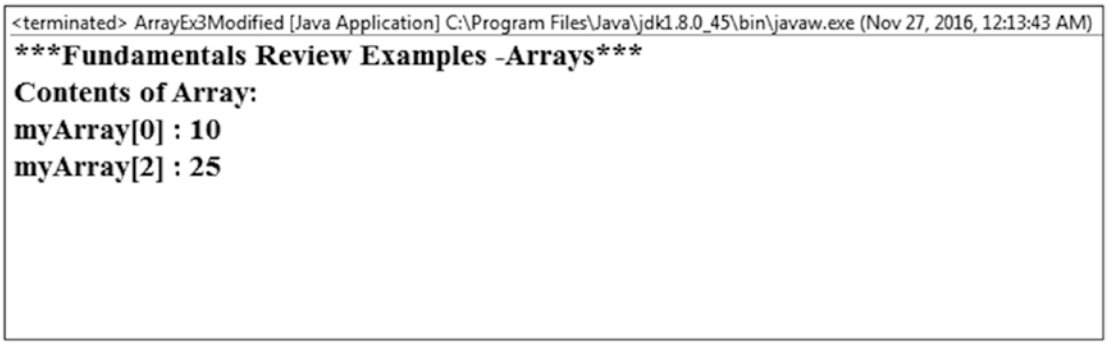
Explanation:

We can see that all object references are initialized to their default values i.e. null. Here we did not provide values for indexes 1, 3 and 4. So, those locations are holding null.

How can we modify the program above to see the values stored inside the objects?

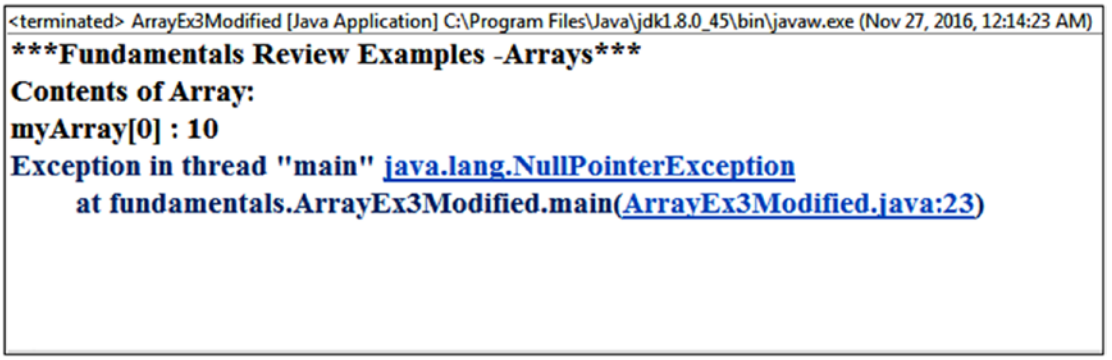
```
package fundamentals;
class A1
{
        int i;
        A1(int i)
        {
                this.i=i;
        }
}
public class ArrayEx3Modified
{
        public static void main(String args[])
        {
                System.out.println("***Fundamentals Review Examples -Arrays***");
                A1[] myArray=new A1[5];
                myArray[0]=new A1(10);
                myArray[2]=new A1(25);
                System.out.println("Contents of Array:");
                for(int i=0;i<5;i++)
                {
                        if(myArray[i]!=null)
                        {
                         System.out.println("myArray["+ i+"] : "+myArray[i].i);
                        }
                }
        }
}
```

Output:



Explanation:

Notice carefully, we need to put an extra guard to do a null check inside the for loop. Otherwise we'll encounter "NullPointerException" because some of the values inside the array are null.



Task:

Can you write a simple array-handling program where you need to supply 4 integers between 1 to 5 (No repetition is allowed). Then your program need to response back to you saying which number you have not used.

```
package fundamentals;

import java.util.Scanner;
public class ArrayEx4 {
        public static void main(String args[])
        {
                System.out.println("***Fundamentals Review Examples -Find a missing number
                using Arrays***");
                System.out.println("Type any 4 integers between 1 and 5( no repetition is
                allowed) :");
                int[] myStore=new int[5];
                int accumulatedSum=0;//To sum up the numbers you //have entered
```

```
                for(int i=0;i<4;i++)
                {
                        Scanner in = new Scanner(System.in);
                        int input= in.nextInt();
                        myStore[i]=input;
                }
                System.out.println("You have entered:");
                for(int i=0;i<4;i++)
                {
                        //if(myArray[i]!=null)
                        {
                                System.out.println("myStore["+ i+"] : "+myStore[i]);
                                accumulatedSum=accumulatedSum+myStore[i];
                        }
                }
                int expectedSum=5*(5+1)/2;//Sum of n integers=n*(n+1)/2;
                int missingNumber=expectedSum-accumulatedSum;
                System.out.println("The missing number is : "+missingNumber);
        }
}
```

Output:

```
<terminated> ArrayEx4 (1) [Java Application] C:\Program Files\Java\jdk1.8.0_45\bin\javaw.exe (Nov 27, 2016, 12:15:39 AM)
***Fundamentals Review Examples -Find a missing number using Arrays***
Type any 4 integers between 1 and 5( no repetition is allowed) :
1
3
2
5
You have entered:
myStore[0] : 1
myStore[1] : 3
myStore[2] : 2
myStore[3] : 5
The missing number is : 4
```

Explanation:

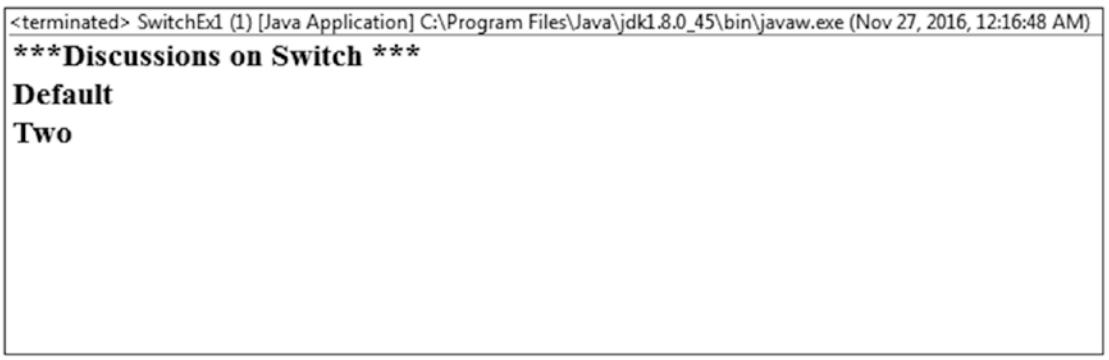
Sum of n numbers=n*(n+1)/2. Replace n with 5 for 5 integers to get the sum of 5 numbers (expectedSum). Now you sum up the 4 numbers which you have entered through keyboard (accumulatedSum). So, the difference between expectedSum and accumulatedSum is the missing number for this scenario.

SET 8

Will the code compile?

```
package fundamentals;
public class SwitchEx1 {
        public static void main(String args[])
        {
                System.out.println("***Discussions on Switch ***");
                int myNumber=6;
                switch (myNumber)
                {
                  case 1:  System.out.println("one");
                           break;
                 default: System.out.println("Default");
                 case 2:  System.out.println("Two");
                           break;
                }
        }
}
```

Yes. We'll get following output:
Default
Two



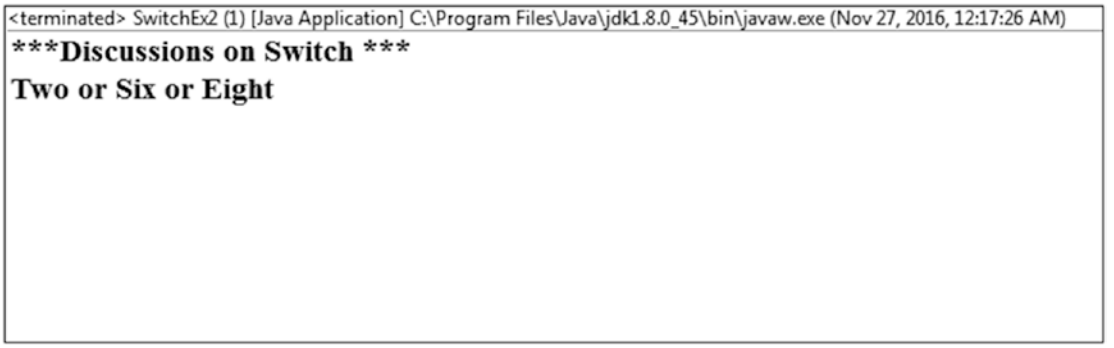
Explanation:
We must note that we can put default case anywhere in the switch block. And also if there is no break statement, control will continue to fall through until a break statement is encountered/end of block is reached-it does not matter whatever be the case.

Will the code compile?

```
package fundamentals;
public class SwitchEx2 {
        public static void main(String args[])
        {
                System.out.println("***Discussions on Switch ***");
                int myNumber=6;
                switch(myNumber)
```

```
                {
                case 1:  case 5:
                        System.out.println("One or Five");
                        break;
                default: System.out.println("Default");
                        break;
                case 2:  case 6:case 8:
                        System.out.println("Two or Six or Eight");
                        break;
}
        }
}
```

Output:

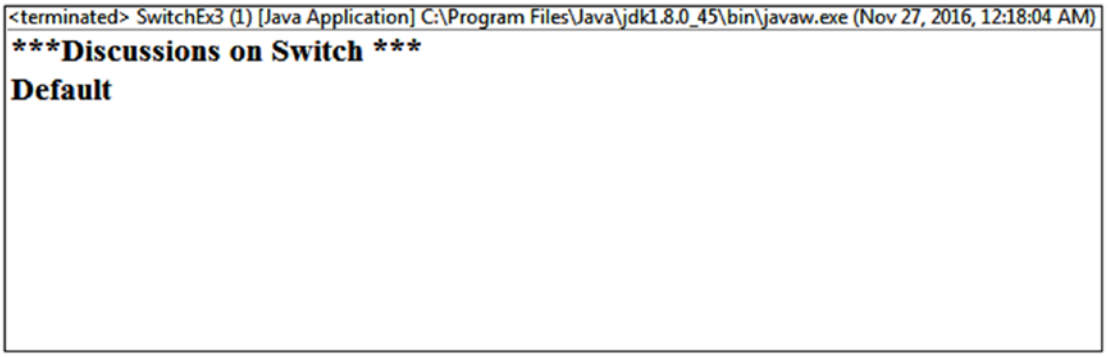


Explanation:
Multiple case labels are possible like this in a switch statement.

Will the code compile?

```
package fundamentals;
public class SwitchEx3 {
        public static void main(String args[])
        {
                System.out.println("***Discussions on Switch ***");
                char myChoice='e';
                switch (myChoice)
                {
                case 'b':
                        System.out.println("b");
                        break;
                default: System.out.println("Default");
                        break;
                case 'a':
                        System.out.println("a");
                        break;
                }
        }
}
```

Output:



Explanation:

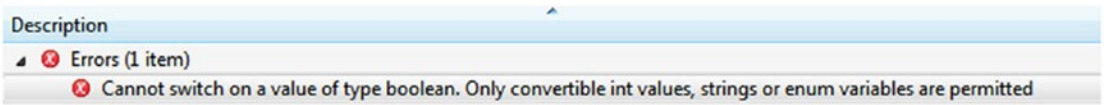
It is not necessary that in the switch statement's expression, we need to put integers only. Other built in data types like byte, short, char and enums also supported here. Java 7 or above starts supporting String objects also inside those expression.

What will be the output?

```
package fundamentals;
public class SwitchEx4 {
        public static void main(String args[])
        {
                System.out.println("***Discussions on Switch ***");
                boolean value=true;
                switch (value) //compile error
                {
                case true:
                        System.out.println("true");
                        break;
                case false:
                        System.out.println("false");
                        break;
                default: System.out.println("Default");
                        break;
                }
        }
}
```

Output:

Compilation error.



Explanation:
Booleans are not supported for switch.

When should we prefer switch over if-else?
There is no universal rule. It depends on the situation or demand of your program. But we must remember the fact that if-else can test conditions or range of values where switch works on an integer, enum or String object.

Name the different type of iteration statements in Java.

- while loop
- do…while loop
- for loop
- for-each loop(From J2SE5)

Why we need these statements?
To create loops. Or simply to execute our code up to some specified number of times repeatedly.

What is the key difference between a while loop and a do…while loop?
In case of do…while, the condition is checked at the end of the loop. So, even the condition is false, do…while loop executes at least once.

Consider the program below. Note that, we are checking whether the value of j is less than 10 or not in the *while* part. Even we're able to print the statement in do{..}.

```
package fundamentals;
public class DoWhileEx
{
 public static void main(String args[])
{
  System.out.println("***do...while Demo***");
  int j=10;
  do
  {
    System.out.println("j is now " + j);
    j++;
   } while (j < 10);
 }
}
```

Output:

```
<terminated> doWhileEx [Java Application] C:\Program Files\Java\jdk1.8.0_45\bin\javaw.exe (Nov 27, 2016, 12:18:48 AM)
***do...while Demo***
j is now 10
```

What will be the output?

```
package fundamentals;

public class WhileDemo2
{
        public static void main(String args[])
        {
                System.out.println("***while Demo-2***");
                int x=10;
                while(x)
                {
                        System.out.println("I am inside the loop");
                }
        }
}
```

Output:

Compilation error.

Description	Resource	Path	Location	Type
Errors (1 item)				
Type mismatch: cannot convert from int to boolean	WhileDemo2.java	/JavaClassNotes/ja...	line 9	Java Problem

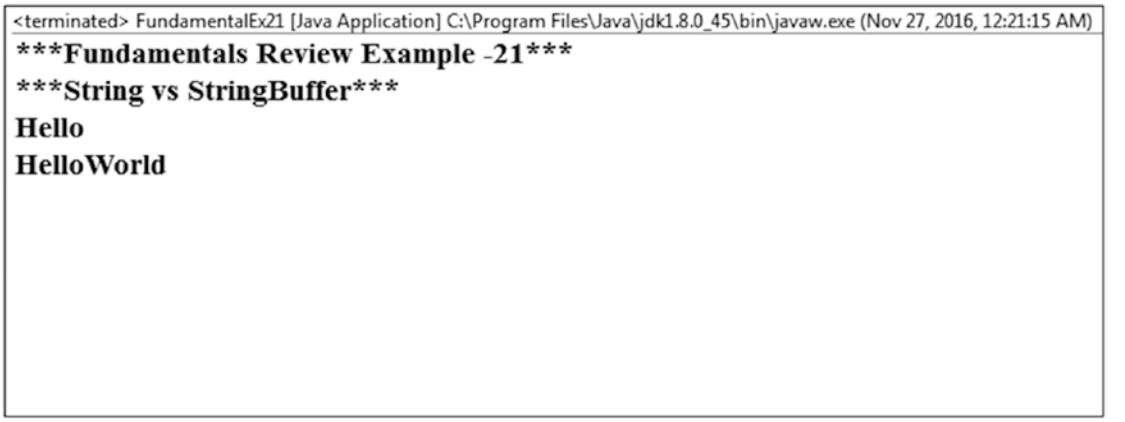
Explanation:
In Java, boolean and int are not compatible. In the above case, we need to use a boolean variable inside the while loop.

SET 9

Predict the output.

```
package fundamentals;

public class FundamentalEx21 {
        public static void main(String args[])
        {
                System.out.println("***Fundamentals Review Example -21***");
                System.out.println("***String vs StringBuffer***");
                String str1="Hello";
                str1.concat("World");
                System.out.println(str1);//Hello

                StringBuffer str2=new StringBuffer("Hello");
                str2.append("World");
                System.out.println(str2);//HelloWorld
        }
}
```

Output:

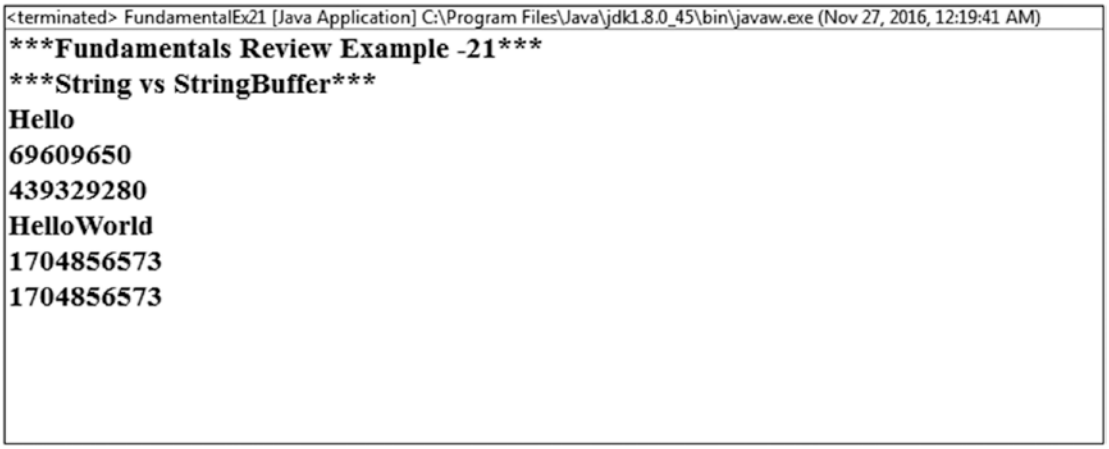


Explanation:

String is immutable i.e. cannot be modified but StringBuffer is mutable. For the String object, when you are concatenating "World", actually a new object is created inside memory. But for StringBuffer, the value of the object modified. To see it properly, you can do a simple test-check the hash codes of them. So, we have included some additional lines of code to the previous program to analyze the output once again. Now you can see that for the String object, we are getting different hash codes but for the StringBuffer object, we are getting the same hash code.

```
package fundamentals;

public class FundamentalEx21 {
        public static void main(String args[])
        {
                System.out.println("***Fundamentals Review Example -21***");
                System.out.println("***String vs StringBuffer***");
                String str1="Hello";
                str1.concat("World");
                System.out.println(str1);//Hello
                System.out.println(str1.hashCode());
                System.out.println(str1.concat("World").hashCode());
                StringBuffer str2=new StringBuffer("Hello");
                str2.append("World");
                System.out.println(str2);//HelloWorld
                System.out.println(str2.hashCode());
                System.out.println(str2.append("World").hashCode());
        }
}
```

Output:



What is the fundamental difference between StringBuffer and StringBuilder ?

StringBuffer is synchronized i.e. in a multithreaded environment it is much preferred than StringBuilder because StringBuilder is not synchronized.

On the other hand, Java Oracle documentation recommends that if speed is the primary concern and synchronization is not important then StringBuilder is preferred over StringBuffer.

What is the difference between applets and applications?

The easiest distinction is application contains main() method and requires JRE. Whereas in an applet you'll not see main(). An applet needs a browser (e.g. Chrome). An applet should be executed in a secured environment whereas an application does not need so much of security compared to an applet. In this book, we have focused only on applications.

CHAPTER 2

Class

Object Oriented Programming (OOP) is based on the following two concepts.

Class:

A class is a blueprint or a template. It will describe about the behaviors of its objects.

Object:

An object is an instance of a class.

With a class, we are creating a new datatype and objects are used to hold the data (fields) and methods. Object behavior can be exposed through these methods.

Suppose, we say, Sachin is a cricketer. If we have some idea about cricket, we can predict that either Sachin plays as a batsman or as a bowler or as a wicketkeeper (or as an all-rounder). Here `Cricketer` is a class and Sachin can be considered as an object of that class.

Now come back to our `Cricketer` class again. Let us say, Sourav is a cricketer. Like the same manner, we can predict Sourav is a batsman or a bowler or a wicketkeeper. Now we can see both Sachin and Sourav are objects of `Cricketer` class but they have individual identity. Obviously Sourav and Sachin show their skills in the game differently even though they are participating in the same game.

Consider a different domain. We can consider our pet dog or cat as an object of an `Animal` class.

Basically all real world objects have two basic characteristics- state and behavior. If you notice the objects- Sachin or Sourav, you can notice that they can also have states-"playing state" or "non-playing state". In playing state, they can show different behaviors-they can bat, they can bowl, they can do fielding etc.

Similarly in non-playing state, they can take meals or they can sleep or they can do some other activities like reading a book, watching a movie etc. Similarly your table lamp can be either in "on" state or in "off" state and it shows different behavior –when you "Switch on the light" or "Turn Off the light".

So, to start with the OOPS programming it is always suggested that you ask yourself these two questions

- What states are possible for your objects?
- What are the functions (behaviors) it can perform?

Once you identify the answers for these questions, you are ready to program because software objects also follow the same pattern- their states are stored in fields/variables and their capabilities/behaviors are represented through functions/methods.

Now come to the programming. Suppose our class name is `MyClass`.

```
Class MyClass {}
```


V. Sarcar, *Interactive Object Oriented Programming in Java*, DOI 10.1007/978-1-4842-2544-8_2

Then we can create an object obA of the class MyClass with the following statement:

```
MyClass obA=new MyClass ();
```

Actually, the above line can be decomposed of two lines as below:

```
MyClass obA;//Line-1
obA=new MyClass ();//Line-2
```

At the end of the first line, obA is a reference. Till this point, there is no memory allocated. But once the new comes into picture, the memory is allocated.

You must note that in the second line, class name is followed by a parentheses. These are for constructors. Constructors are used to describe what will happen when an object is created. Constructors can have different attributes. But if our class does not specifically define a constructor, Java will supply a default one. In the above example, we have used a default no argument constructor.

A simple class demonstration

A class can have variables and methods. The variables defined in a class are called instance variables because each instance of a class will hold their own copy of these variables. On the other hand, methods will contain the actual codes. Instance variables, in general, accessed by these methods (or acted on these methods). These variables and methods are collectively termed as class members.

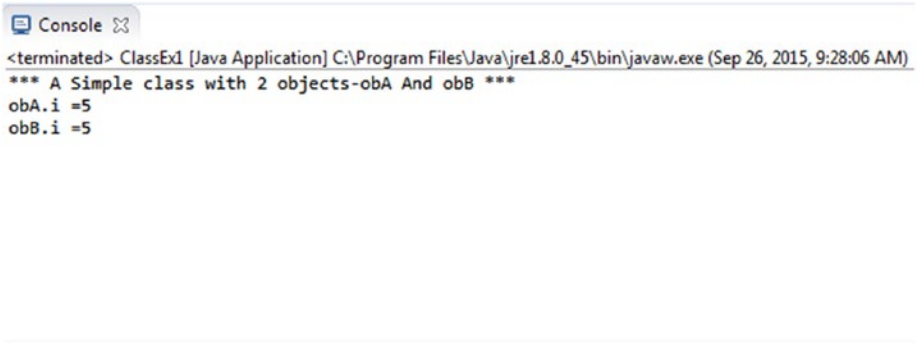
Now go through a simple example. Here our class name is ClassA. It has only one field-i which is of type int. Here the value of i already has the value 5 associated with it. So, we can predict that if we create an object for this class, the object of that class will have an integer named i and the value of i in it will be 5.

For your ready reference, we have created 2 objects obA and obB for our class ClassA here. We have tested the values of the variable i inside the objects. You can see that both have the value 5.

Demonstration-1

```
package classes.examples;

class ClassA
{
        int i=5;
}
class ClassEx1
{
        public static void main(String args[])
        {
                System.out.println("*** A Simple class with 2 objects-obA And obB ***");
                ClassA obA=new ClassA();
                ClassA obB=new ClassA();
                System.out.println("obA.i ="+ obA.i);
                System.out.println("obB.i ="+ obB.i);

        }
}
```

Output



Students ask:

Sir, can you tell me something more on constructors?

Teacher says: Constructors are used to initialize objects. It must have the same name as the class in which it stays and it does not have any return type.

Basically there are 2 types of constructors- constructors with no argument and constructors with parameter/s (termed as parameterized constructors). But when constructors with no arguments is created by Java, it is termed as a default constructor. And if constructors with no argument is created by us, we term it as no argument constructor.

So, we can say that we do some common initialization for all the variables inside a class through the constructors when an instance of that class is created.

Students ask:

Sir, constructors do not have any return type. Did you mean to say that their return type is `void`?

Teacher says: Not at all. Even void is also considered as a return type. We do not have return type for constructors because implicitly their return type is same as their class type.

Students ask:

Sir, then no argument constructor and default constructor appears same to me. What is the actual difference between them?

Teacher says: We already mentioned that default constructors are supplied by Java only, if you do not supply any constructor for your class. Yes, since they also do not have arguments- both may appear to be the same. But in a no argument constructor, you can add your own body and that's why it is not default.

In addition to this, all access modifiers like –private, protected, public or default can be used for developer created no-argument constructor whereas Java provided default constructors have access modifiers decided based on the class access specifier.

Below is an example of a no-argument constructor:

Demonstration-2

```
package classes.examples;

class MyClass
{
        protected MyClass()
        {
                System.out.println("I am a no argument constructor");
```

```
                System.out.println("I Can have additional logic");
        }
}
public class ExperimentWithConstructorEx1
{
        public static void main(String args[])
        {
                System.out.println("*** Experiment with constructors  ***");
                MyClass myOb=new MyClass();
        }
}
```

Output

```
*** Experiment with constructors  ***
I am a no argument constructor
I Can have additional logic
```

So, we can see the messages inside the no-argument constructor are printed in console because Java knows that we have defined a constructor and so it does not need to supply a default one.

Also default constructors have `super()` calls only but developer created no-argument constructors can have both the logic and super calls.

■ **Note** You will learn about access specifiers in the chapter of Package and you will learn about super in the chapter of inheritance. So, you can come back to this question once you have completed those topics.

Students ask:

Sir, we can have method to initialize all these variables. Then why do we need constructors?

Teacher says: In that case also, you need to make an explicit call to your method to initialize all those variables. It means that your call is not automatic.

On the other hand, automatic initialization happens for constructors when each time we create an object.

Teacher asks:

Can you predict the output?

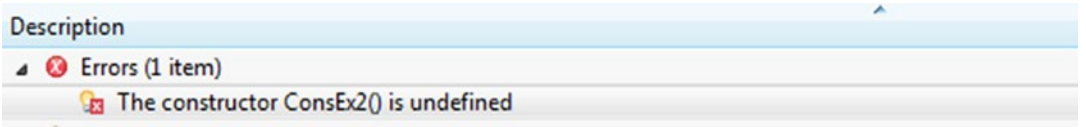
Quiz

```
package classes.examples;

class ConsEx2
{
        int i;
        ConsEx2(int i)
        {
                this.i=i;
        }
}
```

```
public class ExperimentWithConstructorEx2 {
        public static void main(String args[])
        {
                ConsEx2 ob2=new ConsEx2();
        }
}
```

Output

Compilation error.



Explanation

See the Q&A below. We'll also discuss about the keyword "this" shortly.

Students ask:

Sir, Java was supposed to provide a default constructor in this case as we did not provide a no argument constructor. Then why we have encountered this error?

Teacher says: Java will provide a default constructor if and only if there is no constructor provided by us. But in this case, We have already defined a constructor which is basically a parameterized constructor. So, now Java will not supply a default constructor for us. This is why, when we try to create an object through a default constructor, compiler will not find such a constructor in the program. Hence it will raise the issue.

To remove the compilation error either you provide a no argument constructor or remove the parameterized constructor or try to use something like

```
ConsEx2 ob2=new ConsEx2(5);
```

Students ask:

Sir, from the above examples it appears to me that class is a custom type-is this understanding correct?

Teacher says: Yes.

Students ask:

Sir, can you please elaborate the concept of reference?

Teacher says: Yes. When we write `ClassA obA=new ClassA();` an instance of `ClassA` will be born in memory and it creates a reference to that instance and stores the result inside the variable `obA`.

Students ask:

Sir, then reference is basically used to point an address. Is the understanding correct?

Teacher says: Yes.

Students ask:

Sir, then references are pointers. Is the understanding correct?

Teacher says: We must remember Java does not support pointers. It may appear that references are special kind of pointers. But we must note the key difference: with a pointer, we can point any address (which is actually a number slot in a memory). So, it is quite possible that with a pointer, we can point an

invalid address also and then we may face surprises issues during runtime. But reference types will always point to valid addresses or they will point to null.

Students ask:

Sir, how can we check whether my reference variable is pointing to null or not?
Teacher says: This simple check can serve your purpose.

```
if(obA==null)
{
  System.out.println("obA is  null");
}
else
{
  System.out.println("obA is   NOT null");
}
```

Students ask:

Sir, Can multiple variables reference a same object in memory?
Teacher says: Yes. Following declaration is perfectly fine:

```
ClassA obA=new ClassA();
ClassA obB=obA;
```

Demonstration-3

In this example we will provide our own constructor here. We can see that we can now initialize objects with different values. `obA` has initialized integer `i` with the value 20 and `obB` has initialized the integer `i` with the value 30.

```
package classes.examples;

class ClassA3
{
        int i;
        ClassA3(int i)
        {
          this.i=i;
        }
}
class ClassEx3
{
        public static void main(String args[])
        {
                System.out.println("*** A Simple class with 2 objects-obA And obB ***");
                System.out.println("*** obA.i And obB.i are different here ***");
                ClassA3 obA=new ClassA3(20);
                ClassA3 obB=new ClassA3(30);
                System.out.println("obA.i ="+ obA.i);
                System.out.println("obB.i ="+ obB.i);
        }
}
```

Output

```
*** A Simple class with 2 objects-obA And obB ***
*** obA.i And obB.i are different here ***
obA.i =20
obB.i =30
```

Students ask:

Sir, what is the use of this here?

Teacher says: Good question. `this` is used to refer the current object. We can omit the use of `this` if we write the code like this:

```
class ClassA
{
        int i;//instance variable
        ClassA(int myInt)//myInt-local variable
        {
          i=myInt;
        }
}
```

We are familiar with the code like this: `a=5;` here we are assigning 5 into `a`. But can we write `5=a;`? No.

Here `myInt` is our local variable (seen inside methods, blocks or constructors) and `i` is our instance variable (declared inside a class but outside a method, block or constructor).

So, instead of `myInt` , if we use `i` , we need to tell compiler about our intention.

```
class ClassA
{
        int i;//instance variable
        ClassA(int i)//i-local variable
        {
          this.i=i; //instance variable is assigned with the value of local variable
        }
}
```

It should not be confused about "which value is assigned where". Here we are assigning the value of the local variable to the instance variable and compiler should clearly understand our intention. With `this.i=i;` compiler will clearly understand the value of the local variable `i` is assigned to instance variable `i`.

We can also explain the scenario from another point of view. For the time being, suppose, you, by mistake have written `i=i` in the above scenario. Then what will happen? Then compiler will see that you are dealing with 2 local variables only and these two are same, so ultimately there is no effect at all. So, now if you create an object say `obA` for `ClassA` and try to see the value of `obA.i`, you will get 0 (default value of an integer) , So, your instance variable cannot get the intended value. Our Eclipse IDE also raise an warning in this case like this:

■ **Note** When we have a local variable which has a same name as an instance variable, the local variable hides the instance variable- this scenario sometimes referred as “Instance variable hiding”.

Demonstration-4

Here we have used two constructors. Go through the program. Notice that we can initialize an object with a default value here. If the default constructor is used during the creational process of an object, the instance variable i will be initialized with 7. We can also supply different values through the non-default constructor.

```
package classes.examples;

class ClassA4
{
        int i;
        ClassA4()
        {
                this.i=7;
        }
        ClassA4(int i)
        {
                this.i=i;
        }

}
class ClassEx4
{
        public static void main(String args[])
        {
                System.out.println("*** A Simple class with 2 objects-obA And obB ***");
                System.out.println("*** Different type of constructors are used here ***");
                ClassA4 obA=new ClassA4();
                ClassA4 obB=new ClassA4(25);
                System.out.println("obA.i ="+ obA.i);
                System.out.println("obB.i ="+ obB.i);
        }
}
```

Output

```
*** A Simple class with 2 objects-obA And obB ***
*** Different type of constructors are used here ***
obA.i =7
obB.i =25
```

Note that inside the no-argument constructor, we could use the keyword `this` differently to achieve the same result. Here we call it as this constructor.The use is shown here:

```
 ...
ClassA()
   {
           //this.i=7;
           //Also valid
           this(7);
   }
 ....
```

Oracle Java documentation clearly states that if present, the invocation of another constructor must be the first line in the constructor i.e.in other words, if present, `this()` should be the first statement inside that block.

Now suppose we are breaking the declaration `ClassA obA=new ClassA();` into two parts as below:

```
Class obA;//line-1
ObA=new ClassA();//line-2
```

Can you tell me : **how much memory will be allocated for line-1?**

Answer: Already we mentioned at the beginning that no memory will be allocated till this point. `obA` will point to `NULL`. Memory will be allocated only after the keyword `new` comes into picture.

Demonstration-5

We mentioned that a class can have both variables and methods. So, now we are going to create a class with a method which will return an integer. This method is used to accept 2 integer inputs and in turn, it will return the sum of those integers.

```
package classes.examples;

class Demo5
{
        int sum(int x, int y)
        {
                return x+y;
        }
}
public class ClassEx5
{
        public static void main(String args[])
        {
                System.out.println("*** A Simple class with a method returning an integer ***");
                Demo5 ob=new Demo5();
                int result=ob.sum(10,20);
                System.out.println("Sum of 10 and 20 is : "+ result);
        }
}
```

Output

```
<terminated> ClassDemo4 [Java Application] C:\Program Files\Java\jdk1.8.0_45\bin\javaw.exe (Aug 23, 2016, 8:29:45 PM)
*** A Simple class with a method returning an integer ***
Sum of 10 and 20 is : 30
```

Students ask:

Sir, can we pass variable number of arguments inside a method?
Teacher says: yes. You can. Java supports this concept. Go through the below program:

Quiz

```
package classes.examples;

class A
{
        int sum;
        int sumOfNumbers(int...values)
        {
                int length=values.length;
                for(int i=0;i<=length;i++)
                {
                        sum=sum+i;
                }
                return sum;
        }
}
public class VarArgsEx {
        public static void main(String args[])
        {
                System.out.print("***Test on a method with variable arguments***\n\n ");
                System.out.println("1+2+3 = "+ new A().sumOfNumbers(1,2,3));
                System.out.println("1+2+3+4+5 ="+ new A().sumOfNumbers(1,2,3,4,5));
        }
}
```

Output

```
<terminated> VarArgsEx [Java Application] C:\Program Files\Java\jdk1.8.0_45\bin\javaw.exe (Aug 23, 2016, 8:34:14 PM)
***Test on a method with variable arguments***

 1+2+3 = 6
1+2+3+4+5 =15
```

Notice that we haven't created any new variable to represent the objects of the class. To reduce the code size directly we have used the format: `new className().methodName(parameter1, parameter 2 etc)`.

Likewise in the previous example of `ClassEx5`, you could directly use:

```
System.out.println("Sum of 10 and 20 is : "+ new ClassEx5().sum(10, 20));
```

Demonstration-6

Students ask:

Sir, what should be the expected output for this program?

```
package classes.examples;

class A6
{
        int i;
        A6()
        {
                this.i=7;
        }
}
class ClassEx6
{
        public static void main(String args[])
        {
                System.out.println("*** Simple class: quizes ***");
                A6 ob1;
                ob1=new A6();
                System.out.println(ob1);
        }
}
```

Output

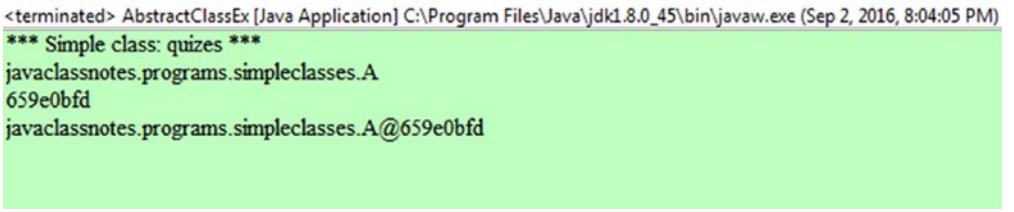
```
<terminated> AbstractClassEx [Java Application] C:\Program Files\Java\jdk1.8.0_45\bin\javaw.exe (Aug 20, 2016, 12:53:33 PM)
*** Simple class: quizes ***
javaclassnotes.programs.simpleclasses.A@659e0bfd
```

Explanation

Teacher says: To understand the above output, we can introduce 2 additional lines inside `main()` in the above program as below:

```
public static void main(String args[])
{
  System.out.println("*** Simple class: quizes ***");
  A6 ob1;
  ob1=new A6();
  System.out.println(ob1.getClass().getName());
  System.out.println(Integer.toHexString(System.identityHashCode(ob1)));
  System.out.println(ob1);
}
```

And you will receive output like this:



So now you can understand the rule: if we simply use `System.out.println(Any object reference)`, we are expected to get output like:

```
OurObject.getClass().getName() + '@' + Integer.toHexString(hashCode())
```

Students ask:

Sir, in the above example we have not defined any method called `getClass()`.Then how could we call this method via an object of class `A6`?

Teacher says: Class `Object` is top of class hierarchy. It is the superclass for all other class by default. And in the `Object` class, that method is defined. You'll understand the concept of superclass when we'll cover the topic on inheritance.

Students ask:

Sir, here we see that codes are always bundled inside objects. What are the benefits with this type of design in real world scenarios?

Teacher says: Actually there are many advantages. Think from a real world scenario, e.g., your laptop or your printer. If any parts of your laptop starts malfunctioning or your print cartridge runs out of ink, you can simply replace those parts, you do not need to replace the entire laptop or entire printer. Same concept applies for other real world objects also.

Secondly, you can reuse the similar parts in similar model of a laptop or a printer.

Apart from this, you must agree that we do not care how these functionalities are actually implemented in those parts. If they are working fine and serve our needs we are happy.

In object oriented programming, objects play the same role-they can be reused and they can be plugged in. At the same time, they are hiding the implementation details e.g. in the `Demonstration 5` example, we can see when we invoke `ob.sum(10,20)`, we know we'll get the result of the sum of 2 integers-10 and 20. But outside user is totally unware how the calculation is happening inside the method. So, we can provide a level of security by hiding these information from outside world.

Lastly from another coding point of view: assume the following scenario. Suppose, you need to store an employee information through your program. If we start coding like this:

```
String empName= "emp1Name";
String deptName= "Comp.Sc.";
int empSalary= "10000";
```

Then for a 2nd employee, we have to write something like this:

```
String empName2= "emp2Name";
String deptName2= "Electrical";
int empSalary2= "20000";
```

and so on. Now, can we really continue like this? Answer is no. To make it simple, it is always a better idea to make an `Employee` class and process like this:

```
Employee emp1, emp2;
```

It is much cleaner, readable and obviously a better approach.

Students ask:

Sir, so far we have talked about constructors but not destructors. Why?

Teacher says: Some other language e.g C++ releases dynamically allocated resources explicitly (by calling delete operator).But Java gives us a facility. It introduced the concept of garbage collection technique. They occur periodically to do their job and reclaim the memory. In general, different Java runtime have their own approaches to garbage collection and they give the relief to users by taking the headache of reclaiming memory.

Students ask:

Sir, how do GCs work?

Teacher says: Simple answer is they will search for references. If there is no reference to an object exist, they assume that the object is no longer needed and so we can reclaim the memory occupied by that object.

Students ask:

Sir, then there is no memory leak in Java. Is this understanding correct?

Teacher says: No. In some cases, your object can hold some other non-java resources (e.g. file handles etc.). And if you do not free them explicitly then you can encounter memory leak over the period of time. So, it is suggested that you use the `finalize()` method in those scenarios and put your intended operations inside your `finalize()` method.

But we must remember that `finalize()` will be called just prior to garbage collection. And we do not know when it will come into picture or whether GC will be called within a certain period of time (because we do not know when GC will be called).So, experienced developers always suggest that it is better to put our own efforts/mechanisms to release system resources etc. (once the job is done) which can be held by an object.

Consider the below example with output and then go through the analysis:

Demonstration-7

```
package classes.examples;
class ConsEx3
{
        int i;
        ConsEx3(int i)
        {
                this.i=i;
        }
        protected void finalize()
        {
                System.out.println("Freeing all resources");
                System.out.println("ConsEx2 object is null already");
        }
}
```

```
public class ExperimentWithConstructorEx3 {
        public static void main(String args[])
        {
                System.out.println("***Experimenting with GC***");
                ConsEx3 ob2=new ConsEx3(5);
                ob2=null;
                //System.runFinalization();
                System.gc();
        }
}
```

Output

```
***Experimenting with GC***
Freeing all resources
ConsEx2 object is null already
```

Analysis

- Note the last 3 lines inside `main()`.We have made the object reference null and make it prepare for garbage collection.
- Then we send a request to GC to invoke its operation. But we must remember it is a request from our side. Garbage collector can obey this command or not. In this case (by seeing the output), it appeared to us that it accepted our request.
- We can use `System.runFinalization()` also to execute the `finalize()` method. But again, it's a request and you are not sure whether the command will be obeyed or not.
- To make a difference in the output, you can try to comment out the line as below:

```
//ob2=null;
```

And upon execution the program, we may notice that `finalize()` is not called yet.

```
***Experimenting with GC***
```

Students ask:

Sir, how can we represent structures in Java?

Teacher says: Java doesn't support structures. Java developers believe that we can replace a structure or a union with a class declaration with appropriate instance variable/s declaration.

Since we can change the visibility with different access specifiers e.g. private, public etc. So, we have the freedom to hide our implementation details differently from different objects.

Assignment

1. Create a class `Vehicle`. The class should have two fields-`no_of_seats` and `no_of_wheels`. Create two objects-`Motorcycle` and `Car` for this class. Your output should show the descriptions for `Car` and `Motorcycle`.

2. Create a class with a method. The method has to decide whether a given year is a leap year or not.

   [Note- A year is a leap year if:

   a. It has an extra day i.e. 366 instead of 365.

   b. It is divisible by 4 year e.g. 2008, 2012 are leap years. The exception to this is the next rule.

   c. For every 100 years a special rule applies-1900 is not a leap year but 2000 is a leap year. In those cases, we need to check whether it is divisible by 400 or not.]

3. Create a class with two functions-one recursive and one non recursive. Either of these function should be capable of calculating the factorial of a number.

CHAPTER 3

Inheritance

The main objective of inheritance is to promote reusability and eliminate redundancy (of code). Here a child class obtains the features of its parent class. By parent class we mean the class which is at the higher level in the class hierarchy compared to another class (which is termed as a child class).

Types of inheritance:

In general, we deal with 4 types of inheritance.

- Single inheritance: One child class is derived from one base class.

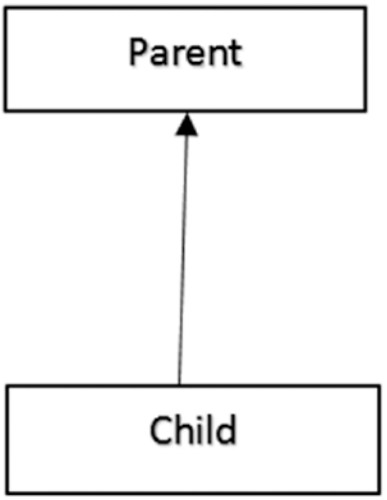


Figure 3-1. *Single inheritance*

The format of code is like this:

```
class Parent
{
 //your code...
}
class Child extends Parent
{
 //your code...
}
```


V. Sarcar, *Interactive Object Oriented Programming in Java*, DOI 10.1007/978-1-4842-2544-8_3

- Hierarchical inheritance: Multiple child class can be derived from one base class.

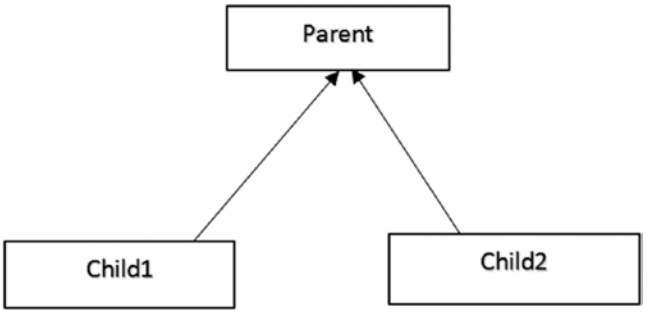


Figure 3-2. *Hierarchical inheritance*

The format of code is like this:

```
class Parent
 {
  //your code...
 }
 class Child1 extends Parent
 {
  //your code...
 }
 class Child2 extends Parent
 {
  //your code...
 }
```

- Multilevel inheritance: Here the parent class has the grandchild.

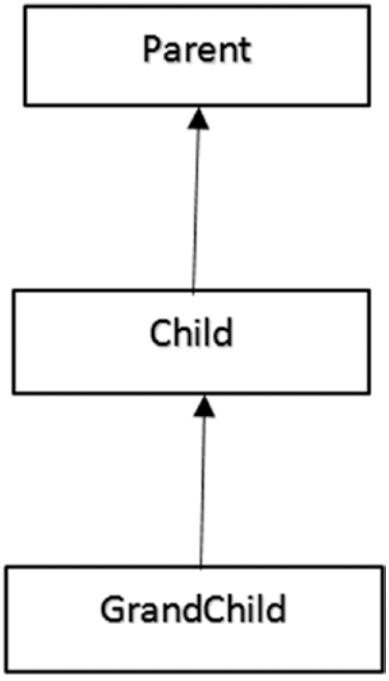


Figure 3-3. *Multilevel inheritance*

Teacher asks:

Now try to implement the concept with Java code.

Solution:

The format of code is like this:

```
class Parent
{
 //your code...
}
class Child extends Parent
{
 //your code...
}
class GrandChild extends Child
{
 //your code...
}
```

- Multiple inheritance: Here a child can derive from multiple Parents.

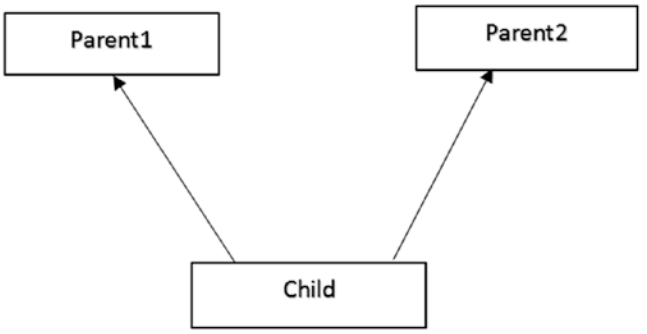


Figure 3-4. *Multiple inheritance*

Note that Java does not support this type of inheritance (through class).i.e.in Java, a child class cannot derive from more than one parent class. To deal with this type of situation we need to understand interfaces.

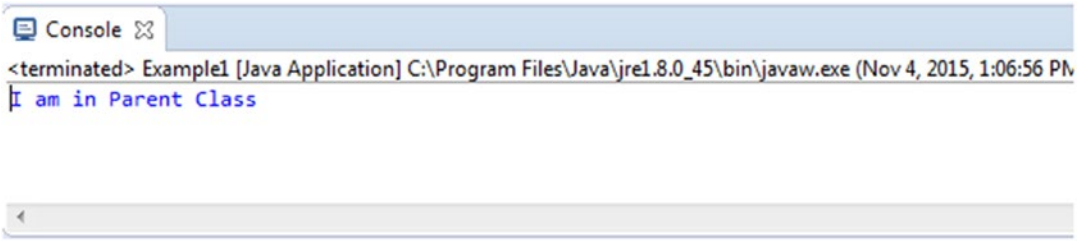
■ **Note** There is another type of inheritance which is termed as hybrid inheritance. This is a combination of one or more types of the above inheritance/s.

A simple program on Inheritance:

Demonstration-1

```
package inheritance.examples;

class ParentClass
{
 public void show()
  {
    System.out.println("I am in Parent Class");
  }
}
class ChildClass extends ParentClass
{
}

class InheritanceEx1
{
        public static void main(String args[])
        {
                ChildClass child1=new ChildClass();
         //Calling show() through ChildClass object
                child1.show();
        }
}
```

Output



■ **Note** Remember that in Java, `Object` (in `java.lang` package) is the superclass for all classes. Because all other classes directly or indirectly is an inheritor of that class.

Students ask:

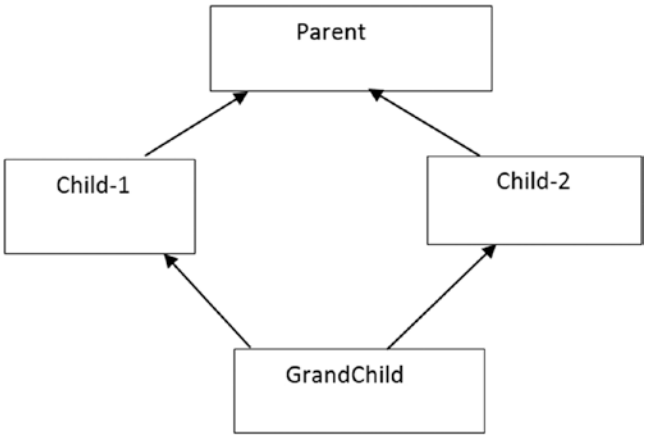
Why Java does not support multiple inheritance through class?

The main reason is to avoid ambiguity. They can cause some confusion in some typical scenarios like this:

Suppose, in our parent class, we have a method named `show()`. The parent class has multiple children- say `child1` and `child 2` who are overriding the method differently for their own purpose. The code may look like this:

```
class Parent
 {
     public void show()
     {
         System.out.println ("I am in Parent");
     }
 }
 class Child1 extends Parent
 {
     public void show()
     {
         System.out.println ("I am in Child-1");
     }
 }
 class Child2 extends Parent
 {
     public void show()
     {
         System.out.println ("I am in Child-2");
     }
 }
```

Now say our `GrandChild` derives from both `Child1` and `Child2` but it has not overridden the method `show()`.



So, now we have an ambiguity-from which class, `GrandChild` will inherit/call `show()`- `Child1` or `Child2`. In order to remove this type of ambiguity Java does not support multiple inheritance through class. This problem is known with a famous name- the **Diamond problem**.

Students ask:

So, there is no programming language that supports multiple inheritance- is this understanding correct?

No. It depends on the designers of the programming language e.g. C++ supports the concept of multiple inheritance.

Students ask:

Why C++ designers support multiple inheritance? The same diamond problem can appear to them also.

I'm trying to explain from my point of view. Probably they did not want to discard the case of multiple inheritance (i.e. they wanted the feature to be included to make the language rich). They supplied developers the support but leaves the control of proper use of the concept to them only.

On the other hand, Java designers wanted to avoid any unwanted situation in future due to this kind of support, they wanted to make the language simple and less error prone.

Teacher asks:

Can we have hybrid inheritance in Java?

Interesting question. Think carefully. Hybrid inheritance can be a combination of two or more type of the above inheritance/s. So, the answer to this question is yes till the point where we are not trying to combine any multiple inheritance through class. And if our intention is to make such a hybrid inheritance in which we need to have any kind of multiple inheritance (through class), Java will not support that concept.

Teacher asks:

Suppose we have a parent class and a child class. Can we guess in which order constructors of the classes will be called?

We must remember that constructor's calls follows the path from parent class to child class. Let's test this with a simple example: here we have a Parent, child and grandchild class. `Child` derives from `Parent`, `GrandChild` derives from `Child`. Now when we create an object of `GrandChild` class, notice the output-we can easily see that constructors are called in the order of their derivation.

Demonstration-2

```
package inheritance.examples;

class Parent
{
        Parent()
        {
                System.out.println("Inside Parent Constructor");
        }
}
class Child extends Parent
{
        Child()
        {
                System.out.println("Inside Child Constructor");
        }
}
class GrandChild extends Child
{
        GrandChild()
        {
                System.out.println("Inside GrandChild Constructor");
        }
}

public class TestConstructorCallingSequence
{
        public static void main(String args[])
        {
                System.out.println("***Inheritance Example***");
                System.out.println("***Testing constructor calling sequence***");
                GrandChild grandChild=new GrandChild();
        }
}
```

Output

```
***Inheritance Example***
***Testing constructor calling sequence***
Inside Parent Constructor
Inside Child Constructor
Inside GrandChild Constructor
```

Students ask:

Sir, sometimes I am confused-which class should be the parent class and which should be a child class in an inheritance hierarchy. Is there any specific guideline to resolve this conflict?

Teacher says: You can try to remember a simple statement-A cricketer is a `Player` but the reverse is not true. This "IS-A" test can help you to decide who should be the `Parent` e.g. here `Player` is the parent class and `Cricketer` is the child class.

We make this "IS-A" test to get some idea in advance whether they can have similar behavior or not and whether we should put them in an inheritance hierarchy or not.

Demonstration-3

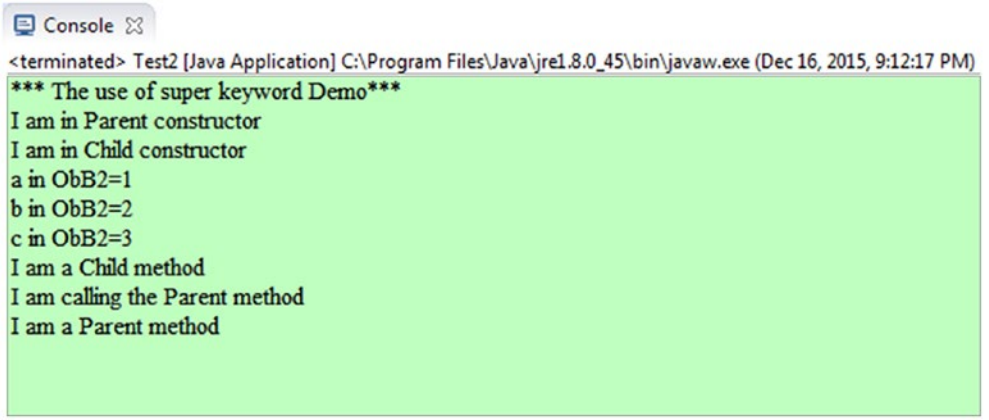
A special keyword: super

In Java, we have a special keyword-super. It is used to access the members of the parent class (super class) in an efficient way. Whenever a child class wants to refer its immediate parent, it should use this keyword.

We can examine the use of super with this simple example.

```
package inheritance.examples;
class A2
{
        int a;
        int b;
        A2(int a,int b)
        {
                System.out.println("I am in Parent constructor");
                this.a=a;
                this.b=b;
        }
        void parentMethod()
        {
                System.out.println("I am a Parent method");
        }
}
class B2 extends A2
{
        int c;
        B2(int a, int b,int c)
        {
                super(a,b);
                System.out.println("I am in Child constructor");
                this.c=c;
        }
        void childMethod()
        {
                System.out.println("I am a Child method");
                System.out.println("I am calling the Parent method");
                super.parentMethod();
        }

}

class TestingSuperEx1
{
        public static void main(String args[])
        {
                System.out.println("*** The use of super keyword Demo***");
                B2 obB2=new B2(1,2,3);
                System.out.println("a in ObB2="+ obB2.a);
                System.out.println("b in ObB2="+ obB2.b);
                System.out.println("c in ObB2="+ obB2.c);
                obB2.childMethod();
        }
}
```

Output



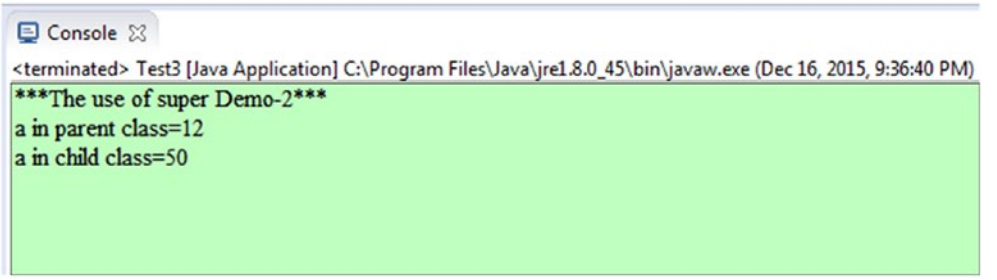
We'll examine another use of `super` with the following example. Here we'll see that even if the instance variable of the parent class becomes hidden by the child class's instance variable, `super` can allow us to access the instance variable in the super class.

Demonstration-4

```
package inheritance.examples;

class A3
{
        int a;
        A3()
        {
                a=25;//some default value
        }
}
class B3 extends A3
{
        int a;//this will hide a in A3
        B3()
        {
                super.a=12;//for a in parent class
                a=50;//for a in B(child class)
        }
        void display()
        {
                System.out.println("a in parent class="+ super.a);
                System.out.println("a in child class="+ a);
        }
}

class TestingSuperEx2
{
        public static void main(String args[])
```

```
        {
                System.out.println("***The use of super Demo-2***");
                B3 obB3=new B3();
                obB3.display();
        }
}
```

Output



Students ask:

Can we use super keyword to call methods that are hidden by a subclass?
Yes.

Students ask:

It appears to me that a subclass can use its superclass methods. But is there any way by which a superclass can use its child class methods?

Teacher says: No. First of all, you must remember that superclass is completed before its subclass, so it has no idea about its subclass methods-it only announces something (you can think about some contract/methods) that can be used by its childs. It is only giving without any expectation to get return from its children.

Also if you notice carefully, you will find that "IS-A" test is one-way e.g. A `Cricketer` *is a* `Player` *always but the reverse is not true. So, there is no concept of backward inheritance.*

Students ask:

Sir, it appears to me whenever I want to use a super class method but want to add something more into it, we can make use of a super call and then put our additional stuff. Is the understanding correct?

Teacher says: Yes. But at the same time, you must have noticed the difference from a constructor call and from a method call. For a constructor call, `super` statement must be the first statement, otherwise you'll get an error like this:

But for a method call, we are seeing that Eclipse is allowing us to put some additional staff before the super call here. But it is always suggested to use super at the beginning.

Java documents states like this: Invocation of a superclass constructor must be the first line in the subclass constructor. If a constructor does not explicitly invoke a superclass constructor, the Java compiler automatically inserts a call to the no-argument constructor of the superclass. If the super class does not have a no-argument constructor, you will get a compile-time error.

It also ensures us that `Object` class has such a constructor. A child class constructor always invokes its parent class constructor explicitly or implicitly and the calling chain runs the way back to the constructor of `Object` class. This scenario is also termed as *chaining of constructors*.

Students ask:

Sir, consider the below program. It is running successfully. But is it not funny that we need to restructure our code i.e. we need to comment out Line-12 and Line-13 below and we need to use Line-15 whereas basically in both ways, looks like, we are doing the same?

Demonstration-5

```
package inheritance.examples;

class SuperA {
        public SuperA(int x)
        {
                System.out.print(x);
        }
}
class SuperB extends SuperA
{
        public SuperB(int a, int b)
        {
                //int c = a + b;//Line-12
                //super(c); //error//Line-13
                //correct coding
                super(a+b); //Line-15
        }
}
public class TestingSuperEx3 {
        public static void main(String[] args)
        {
                SuperB sb=new SuperB(10,15);
        }
}
```

Teacher says: From the Java documentation, it appears to me that Java developers did not want to break the constructor chaining. Looks like, they felt that if we are allowed to put statements like this before a `super` call, someone may do the misuse- they can perform some invalid operations before the creation of the parent object itself.

Students ask:

Sir, previously you told us that `this()` should be the first statement. Now you are telling us `super()` should be the first statement. Then what will happen if we have both in the same constructor?

Teacher says: Good question. But you must notice that both are constructor calls. And if we have a constructor call, it must be the first statement. So, we cannot have both in the same constructor.

Students ask:

Sir, then it is a very restrictive design. Isn't it?

Teacher says: Following program analysis can remove your doubt. So, let's go through the following program and output:

Demonstration-6

```
package inheritance.examples;

class ParentA {
        int i;
        ParentA()
        {
                System.out.println("Parent no -argument constructor");
        }
}
class ChildA extends ParentA
{
        int b;
        ChildA()
        {
                //both this() and super() cannot be used together
                //super();
                this(2);
                System.out.println("Child no -argument constructor");
        }
        ChildA(int b)
        {
                this.b=b;
                System.out.println("Child constructor . b= " +b);
        }
}
public class TestingthisEx1
{
        public static void main(String[] args)
        {
                ChildA obCA=new ChildA();
        }
}
```

Output

```
<terminated> TestingthisEx1 [Java Application] C:\Program Files\Java\jdk1.8.0_45\bin\javaw.exe (Oct 8, 2016, 11:46:42 PM)
Parent no -argument constructor
Child constructor . b= 2
Child no -argument constructor
```

Now enable the `super` statement and comment out the `this()` constructor in the above program as below:

```
ChildA()
        {
                //both this() and super() cannot be used together
                 super();
                //this(2);
                System.out.println("Child no -argument constructor");
        }
```

And run the program again. We'll see the following output:

```
<terminated> Testingthis [Java Application] C:\Program Files\Java\jdk1.8.0_45\bin\javaw.exe (Sep 19, 2016, 9:31:59 PM)
Parent no -argument constructor
Child no -argument constructor
```

So, have you noticed an interesting scenario?

It does not matter whether you make an explicit call to the parent class constructor through `super()` or not, `Parent` class constructor is always called. So, we can assume that if Java developers allowed us to put `super()` and `this()` in the same constructor, ultimately we'll end up with multiple super calls in the calls of constructor chaining which is obviously not a good design.

Students ask:

Sir, in OOP, inheritance is helping us to reuse the behavior. Is there any other way to achieve the same?

Teacher says: Yes. Though the concept of inheritance is used in most of the places but it will not provide the best solution always. To understand it better, you need to understand the concept of design patterns and our earlier release Java Design Patterns, Apress 2016 can help you a lot to develop those tricks.

Students ask:

Sir, it appears to me if any one already made a method for his application, we should always reuse the same method through the concept of inheritance to avoid the duplicate effort. Is the understanding correct?

Teacher says: Not at all. You cannot generalize like that. It depends on the behavior of your application. Suppose, someone has made a method `show()` to print something for his `Car` class. Now if you want to reuse the method for your `AnimalFactory` class, you'll write something like:

```
Class AnimalFactory extends Car{...}
```

Is it a good design? You must agree that there is no relationship between these two classes and we should not relate them in the same inheritance hierarchy.

Students ask:

Can we inherit a constructor?

No.

Assignment

1. Write a simple program to implement hierarchical inheritance.
2. Write a simple program to implement multilevel inheritance.

CHAPTER 4

Overloading

Teacher asks:

Consider the below program segments. Do you notice any specific pattern?

```
int sum(int x,int y)
{
        return x+y;
}
double sum(double x,double y)
{
        return x+y;
}
String sum(String s1,String s2)
{
        return s1.concat(s2);
}
```

Students respond:

Yes sir. We are seeing all of the methods have the same name `sum` but from their method bodies it appears that each method is doing different things.

Teacher says: yes. You are correct. When we do this kind of coding, we term it as method overloading. But you should notice that though method names are same but method signatures are different here.

Students ask:

What is a method signature?

Ideally method name with number and types of the parameters consist the method signature. Java compiler can distinguish among methods with same name but different parameter list e.g. for Java compiler, `double sum(double x, double y)` is different from `int sum(int x, int y)`.

Consider the below program. Here we represent method overloading with the following example:

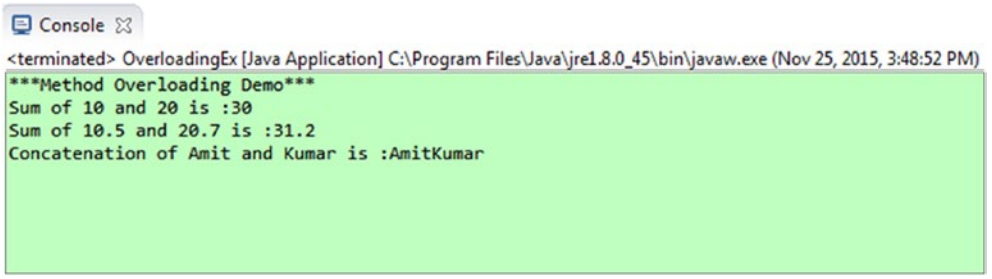
Demonstration-1

```
package overloading.examples;

class Addition
{
        int sum(int x,int y)
        {
                return x+y;
        }
```


V. Sarcar, *Interactive Object Oriented Programming in Java*, DOI 10.1007/978-1-4842-2544-8_4

```
        double sum(double x,double y)
        {
                return x+y;
        }
        String sum(String s1,String s2)
        {
                return s1.concat(s2);
        }
}

public class OverloadingEx
{
        public static void main(String args[])
        {
                System.out.println("***Method Overloading Demo***");
                Addition additionOb=new Addition();
                int sumOfIntergers=additionOb.sum(10,20);
                System.out.println("Sum of 10 and 20 is :"+sumOfIntergers);
                double sumOfDoubles=additionOb.sum(10.5,20.7);
                System.out.println("Sum of 10.5 and 20.7 is :"+sumOfDoubles);
                String sumOfStrings=additionOb.sum("Amit","Kumar");
                System.out.println("Concatenation of Amit and Kumar is :"+sumOfStrings);
        }
}
```

Output



Teacher asks:

Is it an example of method overloading?

```
int sum(int x,int y)
        {
                return x+y;
        }
int sum(int x,int y,int z)
        {
                return x+y+z;
        }
```

Answer: Yes.

Teacher asks:

Is it an example of method overloading?

```
int sum(int x,int y)
        {
                return x+y;
        }
double sum(int x,int y)
        {
                return x+y;
        }
```

Answer: No. Compiler will not consider return type to differentiate these methods. Return type is not considered as a part of method signature.

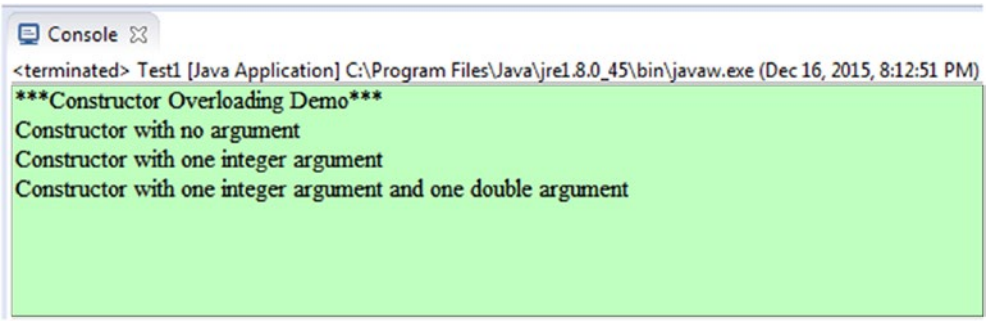
Students ask:

Sir, can we have constructor overloading?
Definitely. You can write a similar program for constructor overloading.

Demonstration-2

```
package overloading.examples;

class A1
{
        A1()
        {
                System.out.println("Constructor with no argument");
        }
        A1(int a)
        {
                System.out.println("Constructor with one integer argument");
        }
        A1(int a,double b)
        {
                System.out.println("Constructor with one integer argument and one double argument");
        }
}
class Test1
{
        public static void main(String args[])
        {
                System.out.println("***Constructor Overloading Demo***");
                A1 ob1=new A1();
                A1 ob2=new A1(2);
                A1 ob3=new A1(2,3.7);
        }
}
```

Output



Students ask:

Sir, it appears to me that it is also method overloading. What is the difference between a constructor and a method?

We already discussed about constructors in the discussion of classes. For your ready reference-a constructor has the same name as class and also it has no return type. So, you can consider a constructor as a special kind of method which has the same name as a class and no return type. But there are many other differences: the main focus of a constructor is to initialize objects. They cannot be called directly.

Students ask:

So Sir, can we write code like this?

Demonstration-3

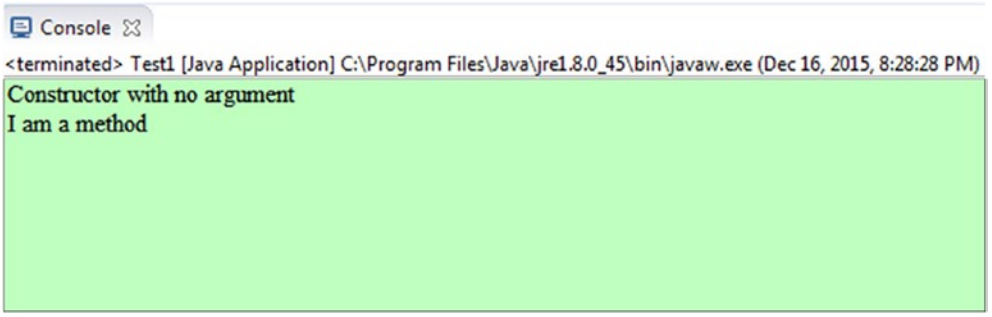
```
class A1

{
        //It is a constructor. It has no return type.
           A1()
        {
                System.out.println("Constructor with no argument");
        }
            //It is a method. It has return types.
        void A1()
        {
                System.out.println("I am a method");
        }
}
```

Sure. Now, the following lines inside `main` function

```
                A1 ob1=new A1();
                ob1.A1(5);
```

can create output like this:

Output



Students ask:

Sir, can we overload the `main()` method?
Teacher says: Yes. Let us go through the following program.

Demonstration-4

```
package overloading.examples;

public class OverloadingMainTest
{
        public static void main(String args[])
        {
                System.out.println("***Method Overloading Demo***");
                System.out.println("From void main(String arg[])");
        }
        public static void main()
        {
                System.out.println("From void main()");
        }
        public static void main(String arg)
        {
                System.out.println("Hello " + arg);
        }
}
```

Output

```
***Method Overloading Demo***
From void main(String arg[])
```

Analysis

We must notice that JVM will always consider the `main(String args[])` or `main(String[] args)` as the entry point. It does not matter how many other overloaded form of `main` method you are using.

Students ask:

Sir, then how can we call the other forms of `main()`?

Teacher says: Just like we do in other scenarios i.e. we can call them from our traditional `main` method (the `main` method with `String` arguments).

In this case, we can add the following two statements (highlighted) in our traditional `main()` method:

```
public static void main(String args[])
{
        System.out.println("***Method Overloading Demo***");
        System.out.println("From void main(String arg[])");
        main();
        main("Vaskaran");
}
```

Now, when we run this application, we'll receive the following output:

```
<terminated> OverloadingMainTest [Java Application] C:\Program Files\Java\jdk1.8.0_45\bin\javaw.exe (Sep 17, 2016, 8:11:32 PM)
***Method Overloading Demo***
From void main(String arg[])
From void main()
Hello Vaskaran
```

Teacher asks:

Can you predict the output?

Quiz

```
package overloading.examples;

public class OverloadingMainTest2
{
        public static void main(String args[])
        {
                System.out.println("***Method Overloading Demo***");
                System.out.println("From void main(String arg[])");

        }
        //Compiler error now
        public static void main(String... args)
        {
                System.out.println("From String... args)");
        }
}
```

This time we will see the compiler error. Now JVM cannot distinguish which one is the entry point.

Output

Description	Resource	Path
◢ Errors (2 items)		
Duplicate method main(String...) in type OverloadingMainTest	OverloadingMainTes...	/JavaClassNotes/ja...
Duplicate method main(String[]) in type OverloadingMainTest	OverloadingMainTes...	/JavaClassNotes/ja...

CHAPTER 5

Overriding

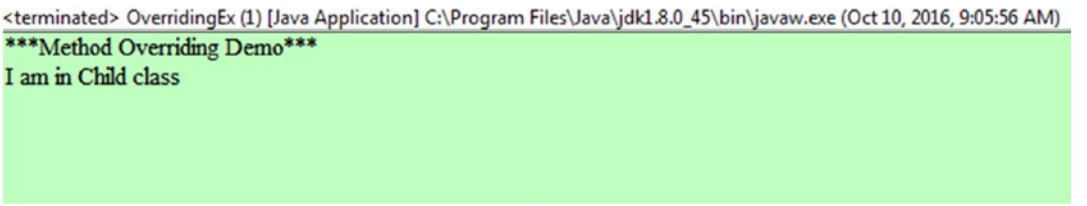
Sometimes we want to redefine or modify the behavior of our parent class. Method overriding comes into picture in such a scenario. Consider the below program. Note that, here `showMe()` method has the same signature in both the parent class and its child class.

Demonstration-1

```
package overriding.examples;
class ParentClass
{
        public void showMe()
        {
                System.out.println("I am in Parent class");
        }
}
class ChildClass extends ParentClass
{
        public void showMe()
        {
                System.out.println("I am in Child class");
        }
}
class OverridingEx
{
        public static void main(String args[])
        {
                System.out.println("***Method Overriding Demo***");
                ChildClass childOb=new ChildClass();
                childOb.showMe();
        }
}
```


V. Sarcar, *Interactive Object Oriented Programming in Java*, DOI 10.1007/978-1-4842-2544-8_5

Output



Analysis

In the above program we are seeing that:

- A method named `showMe()` with same signature and return type is defined in both the `ParentClass` and the `ChildClass`.
- As the name suggests, `ChildClass` is basically derived class whose parent is `ParentClass`.
- In the `main()` method, we created a child class object `childOb`. And when we invoke the method `showMe()` through this object, it is calling the `showMe()` version defined in `ChildClass` i.e. the parent method version is overridden. *Hence the scenario is termed as* ***method overriding.***

■ **Note** There is concept called covariant return type in Java. We'll discuss about this later. Since it is a relatively advanced concept, it is suggested that you understand the original concepts first.

Students ask:

Sir, in method overloading, return types didn't matter. But here it matters. Is the understanding correct?

Teacher says*:* Yes. Here child class method's return type must be the same (or subclass type-which we'll discuss later) as the parent class method's return type (or in simple word, both type must be compatible).

Students ask:

Sir, then will following program receive compilation error?

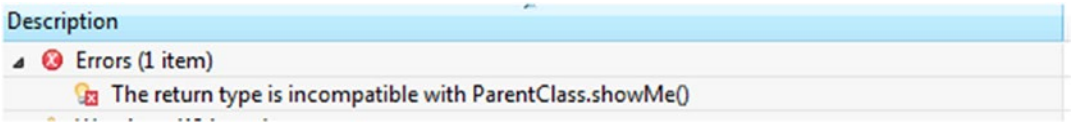
Demonstration-2

```
package overriding.examples;

class ParentClass
{
        public void showMe()
        {
                System.out.println("I am in Parent class");
        }
}
```

```
class ChildClass extends ParentClass
{
        //Error
        public int showMe()
        {
                System.out.println("I am in Child class");
                return 5;
        }
}
```

Teacher says: Yes. Here child class method's return type is `int` and parent class method's return type is void. So, inside `ChildClass`, compiler may try to treat it as overloading and then it will find that only return types are different, so it cannot treat is as overloading also.Then it will give up and raise the compilation error:.

Output



So, to overcome this, you can do this simple change and redefine the method inside `ChildClass` as below:

```
public int showMe(int i)
        {
                System.out.println("I am in Child class");
                return i;
        }
```

Now you can use both the methods –from parent and child classes. *But note that, now you are using overloading but not overriding.*

So, below program is a perfect example where both the concept of method overloading and method overriding are combined together.

Demonstration-3

```
package overriding.examples;

class ParentClassDemo
{
        public void showMe()
        {
                System.out.println("I am in Parent class");
        }
}
class ChildClassDemo extends ParentClassDemo
{
        //Here it is method overriding
        public void showMe()
        {
                System.out.println("I am in Child class");
        }
```

```
        //Error-method type is not compatible
        /*public int showMe()
        {
                System.out.println("I am in Child class");
                return 5;
        }*/
        //Ok-treating as method overloading
        public int showMe(int i)
        {
                System.out.println("I am also in Child class");
                return i;
        }
}
class OverridingEx2
{
        public static void main(String args[])
        {
                System.out.println("***Method Overriding with method overloading Demo***");
                ChildClassDemo childOb=new ChildClassDemo();
                childOb.showMe();
                System.out.println(childOb.showMe(5));//5
        }
}
```

Output

```
<terminated> OverridingEx2 (1) [Java Application] C:\Program Files\Java\jdk1.8.0_45\bin\javaw.exe (Oct 10, 2016, 9:09:39 AM)
***Method Overriding with method overloading Demo***
I am in Child class
I am also in Child class
5
```

Dynamic Method Dispatch

This is an extremely important concept in Java. Java can implement runtime polymorphism through this technique. This technique is considered to implement runtime polymorphism because the call to an overridden method is resolved dynamically at runtime. Java will call the appropriate method based on the object which we are referring.

Demonstration-4

```
package overriding.examples;

class MyParentClass
{
        public void showMe()
        {
                System.out.println("I am in Parent class");
        }
}
class MyChildClass extends MyParentClass
{
        public void showMe()
        {
                System.out.println("I am in Child class");
        }
}
class DynamicMethodDispatchEx
{
        public static void main(String args[])
        {
                System.out.println("***Dynamic Method Dispatch Demo***");
                MyParentClass parent=new MyParentClass();
                parent.showMe();
                MyChildClass childOb=new MyChildClass();
                /*Parent class reference to a child object*/
                parent=childOb;
                parent.showMe();
        }
}
```

Output

```
<terminated> DynamicMethodDispatchEx (1) [Java Application] C:\Program Files\Java\jdk1.8.0_45\bin\javaw.exe (Oct 10, 2016, 9:12:09 AM)
***Dynamic Method Dispatch Demo***
I am in Parent class
I am in Child class
```

Points to remember

Through a parent class reference, we can refer a child class object but the reverse is not applicable.
So,

```
MyParentClass parent=new MyChildClass(); // is ok but
MyChildClass child=new MyParentClass(); // will raise error.
```

Students ask:

Sir, you are saying "The parent class reference can point to a child object but the reverse is not true"- why we support this kind of design?

Teacher says: We must agree about these facts:

All cricketers are players but the reverse is not true. (Because there are many players who play football, golf, basketball, hockey etc.)

Similarly we can say that all buses are vehicles but the reverse is not true because there are other vehicles like trains, ships which are not definitely buses.

Like the same manner, in programming terminology, all derived classes are of type base classes but the reverse is not true e.g suppose we have class called `Rectangle` and it is derived from an another class called `Shape`. Then we can say that all `Rectangle`s are `Shape`s but the reverse is not true.

And you must remember that we already mentioned that we need to do an "IS-A" test for an inheritance hierarchy and an "IS-A" is always one-way e.g. there is no concept of backward inheritance.

Teacher asks:

Now there may be some situation where we want a restriction: A method in the parent should not be overridden by its child. How can we achieve that?

In many interviews, you can face this question. We must remember that we can prevent overriding by the use of `static`, `private` or `final` keywords. But here we have discussed about the use of `final` only. It is very much helpful because compiler itself will prevent the process of overriding.

Use of "final" keyword

```
class ParentClass
{
     //Use of final to prevent overriding
        final public void showMe()
        {
                System.out.println("I am in Parent class");
        }
}
class ChildClass extends ParentClass
{
        //Cannot override now: It is not allowed
        public void showMe()
        {
                System.out.println("I am in Child class");
        }
}
```

So, with the above code, compiler will raise the error.

To understand it better go through the following program. Here we have compared between a final method and a non-final method.

Demonstration-5

```
package overriding.examples;

class MyParentClass1
{
        final public void aFinalMethod()
        {
                System.out.println("In Parent class-a Final method");
        }
        public void aNonFinalMethod()
        {
                System.out.println("In Parent class-Non final method");
        }
}
class MyChildClass1 extends MyParentClass1
{
        //Cannot override now
        public void  aFinalMethod()
        {
                System.out.println("I am in Child class-aFinalMethod");
        }
        //It is ok
         public void aNonFinalMethod()
        {
                System.out.println("I am in Child  class-a Non Final Method");
        }

}
class PreventOverridingEx
{
        public static void main(String args[])
        {
                System.out.println("***Prevent Method Overriding by use of final - Demo***");
                MyChildClass1 childOb=new MyChildClass1();
                childOb.aNonFinalMethod();
                childOb.aFinalMethod();
        }
}
```

Output

Compilation error.

If you comment out the following block inside the child class as below:

```
/*//Cannot override now
public void  aFinalMethod()
{
        System.out.println("I am in Child class-aFinalMethod");
}*/
```

and then try to execute you program, your program will be compiled successfully and upon run, it will generate the following output:

```
<terminated> PreventOverridingEx [Java Application] C:\Program Files\Java\jdk1.8.0_45\bin\javaw.exe (Sep 17, 2016, 10:48:58 AM)
***Prevent Method Overriding by use of final - Demo***
I am in Child  class-a Non Final Method
In Parent class-a Final method
```

■ **Note** We also remember that if a class have only private constructors, it cannot be sub classed.This concept can be used to make a Singleton design pattern where we prevent unnecessary objects creation in the system with the use of `new` keyword.

Quiz

Teacher asks:

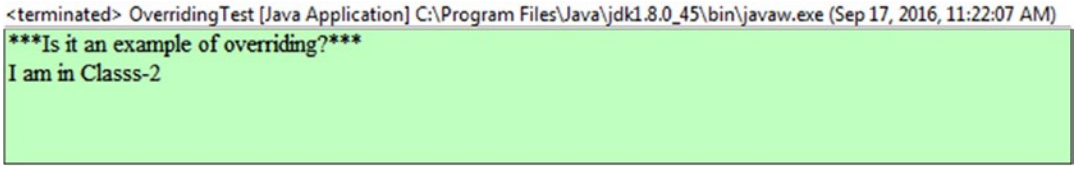
Can you predict the output? Is there any compilation error?

```
package overriding.examples;

//Is it an example of overriding?
class Class1
{
        final public void aFinalMethod()
        {
                System.out.println("I am in Class-1");
        }
}
class Class2
{
        final public void aFinalMethod()
        {
                System.out.println("I am in Classs-2");
        }
}
```

```
public class OverridingTest
{
        public static void main(String args[])
        {
                System.out.println("***Is it an example of overriding?***");
                Class2 obClass2=new Class2();
                obClass2.aFinalMethod();
        }
}
```

Output

The program will compile and run successfully.



We encountered no issue with final this time because the concept of overriding appears between a superclass and a child class. Since `Class2` is not a child class of `Class1`, it can define its final method with the same method name in `Class1`.

Students ask:

Sir, so far you have used the keyword final to methods. Can we apply it to variables also?

Teacher says: Yes. We can apply it to variables as well as class itself. Its nature is same .Consider the below cases:

Case #1: Making an instance variable final

```
class FinalDemo
{
        final double PI=3.14;
}
```

Or,

Case #2: Making the whole class final

```
final class FinalDemo
{
        final double PI=3.14;//instance variable is final
            //Additional code
}
```

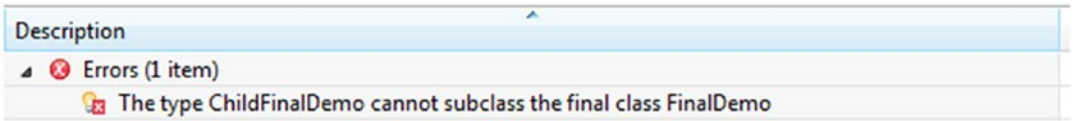
But we must remember that if we make the class final, then we cannot extend it. So, following codes will raise compile time error:

Demonstration-6

```
final class FinalDemo
{
        final double PI=3.14;//instance variable is final
}
//Error if class is final
class ChildFinalDemo extends FinalDemo
{

}
```

Output

Compilation error: The type `ChildFinalDemo` cannot subclass the final class `FinalDemo`.



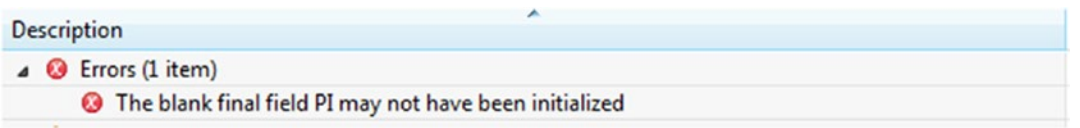
Students ask:

Sir, we are seeing that each time you are initializing final variables. Is it mandatory?

Teacher says: Yes. Otherwise you have to initialize final variables inside the constructors. And when we do that kind of initialization (for those uninitialized final variables), we use the term blank final variables. Consider the below code for better understanding:

```
class FinalDemo2
{
        //Must be initialized inside a constructor
        final double PI;
        double area;
        //final double PI=3.14;
        FinalDemo2()
        {
         PI=3.14;
        }
}
```

If we do not initialize the final variable inside the constructor, we'll receive the compilation error:



Students ask:

Sir, if we have multiple constructors, do we need to initialize the final variables in each of them?

Teacher says: Yes. Otherwise, you can call another constructor who can do that initialization for you. Following program can help you to understand the trick:

Demonstration-7

```
package overriding.examples;
class FinalDemo2
{
        //Must be initialized inside a constructor
        final double PI;
        double area;
        FinalDemo2()
        {
         PI=3.14;
        }
        FinalDemo2(int radius)
        {
                //Calling no-argument constructor to initialize the final variable
                this();
                this.area=this.PI*radius*radius;
        }
}
class Testingfinal
{
        public static void main(String args[])
        {
                System.out.println("*** Testing the behavior of final ***");
                FinalDemo2 fd=new FinalDemo2(5);
                System.out.println("Area of a circle with radious 5 unit is ="+ fd.area+ "
                Sq. Unit");
        }
}
```

Output

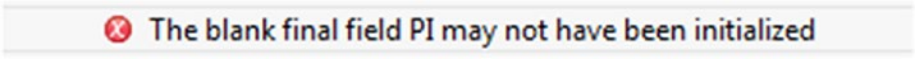
```
<terminated> Testingfinal [Java Application] C:\Program Files\Java\jdk1.8.0_45\bin\javaw.exe (Sep 20, 2016, 8:23:01 PM)
*** Testing the behavior of final ***
Area of a circle with radious 5 unit is =78.5 Sq. Unit
```

Analysis

If you comment out the line

```
//this();
```

In the above program, you'll see the same error again:



Students ask:

Can we have constructor overriding in Java?
Teacher says: No.

Students ask:

Sir, why final variables need to be initialized?
Teacher says: Just think, they are acting like constants throughout your program. If you do not initialize them at the beginning, others cannot supply/modify them in future. By declaring final, you are preventing the change in some later phase.

Students ask:

Sir, suppose I want to use a variable that will be accessible from every places but the value in it cannot be changed. How we can achieve that?
Teacher says: Basically you are trying to bring the concept of global variable which is not supported in Java. But for your case, you can declare a variable like this:

```
public static final  double PI=3.14;
```

You'll know more about `static` later in the book.

Teacher asks:

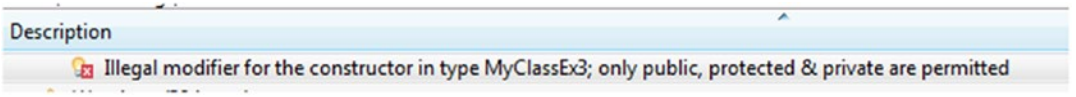
Can you predict the output?

Quiz

```
package overriding.examples;
class MyClassEx3
{
        final MyClassEx3()
        {
                System.out.println("I am a no argument constructor");
        }
}
class ExperimentWithConstructorEx3
{
        public static void main(String args[])
        {
                System.out.println("*** Experiment with constructors  ***");
                System.out.println("***Question: Can construcors be final?  ***");
                MyClassEx3 myOb=new  MyClassEx3();
        }
}
```

Output

Compilation error.



Students ask:

Sir, why we are encountering errors when we try to use `final` keywords with constructors?
Teacher says: Think from a general point of view: The keyword `final` is used to prevent overriding but constructors cannot be overridden at all as per the language specification.

Students ask:

Sir, can we override the `main()` method?
Teacher says: No. It is because static methods cannot be overridden. We'll see the detailed discussions and complete implementations of static methods later in the chapter of "Use of static keyword".

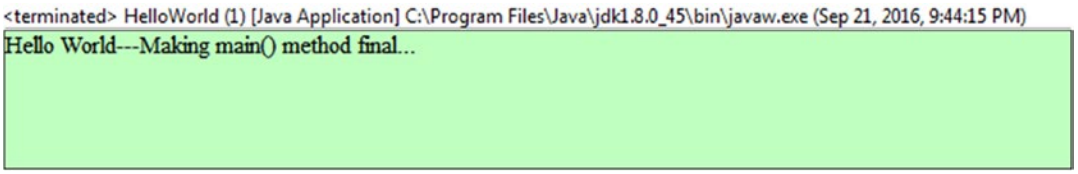
Students ask:

Sir, can we make the `main()` method final?
Teacher says: In Eclipse neon, we tried the following program and found no compile time or runtime error. But I do not seeing any significant benefit by making this change to our conventional `main()`.

Demonstration-8

```
package overriding.examples;

class HelloWorld
{
        public static final void main(String args[])
        {
                System.out.println("Hello World---Making main() method final...");
        }
}
```

Output



Students ask:

Sir, can we override an overloaded method?
Teacher says: Sure. Consider the below program and corresponding output.

Demonstration-9

```
package overriding.examples;

class ParentOverloadedClass
{
        public void showMe()
        {
                System.out.println("I am in Parent class");
        }
        public void showMe(int x)
        {
                System.out.println("Overloaded method in Parent. x is " +x);
        }
}
class ChildOverridedClass extends ParentOverloadedClass
{
        public void showMe()
        {
                System.out.println("I am in Child class");
        }
}
class  OverloadingWithOverridingEx
{
        public static void main(String args[])
        {
                System.out.println("***Method Overriding with overloading Demo***");
                ChildOverridedClass childOb=new ChildOverridedClass();
                childOb.showMe();
                childOb.showMe(25);
        }
}
```

Output

```
<terminated> OverloadingWithOverridingEx [Java Application] C:\Program Files\Java\jdk1.8.0_45\bin\javaw.exe (Sep 21, 2016, 9:54:07 PM)
***Method Overriding with overloading Demo***
I am in Child class
Overloaded method in Parent. x is 25
```

Students ask:

Sir, give us some pointer, so that we can easily distinguish between method overloading and method overriding.

Teacher says: Following points can help you to brush up your knowledge:

- In method overloading –all *methods may reside inside the same class* (you must note the word *may* here-because we have already implemented an example before where both method overloading and method overriding are implemented and the concept of method overloading spans in 2 classes-a parent and its child).In method overriding-inheritance hierarchy of a parent class and a child class is involved, means that at least a parent class and its child class(i.e. minimum 2 classes) are involved in case of overriding.
- Method overloading-signatures are different. In method overriding-method signatures are same (No need to consider covariant return type at this point).
- We can achieve compile time (static) polymorphism through method overloading but we can achieve runtime (dynamic) polymorphism through method overriding.

Covariant return type

■ **Note** You are suggested to come back this topic once you complete other fundamental topics first to avoid confusions.

Teacher continues:
Consider the below program and output carefully:

Demonstration-10

```
//Primary Version

package overriding.examples;

class ParentCov
{
        int i;
        int getMultipliedNumber(int x )
        {
                System.out.println("Inside Parent");
                this.i=x;
                return (int) (i*1.75);
        }
}
class ChildCov extends ParentCov
{
        int getMultipliedNumber(int x )
        //error:Return type is incompatible
        //double getMultipliedNumber(int x )
        {
                System.out.println("Inside Child");
                this.i=x;
                return  i*50;
        }

}
```

```
public class CovarianceEx {
        public static void main(String args[])

        {System.out.println("***Datatype :primitive(int)***\n");
                ParentCov pOb=new ParentCov();
                int result1=pOb.getMultipliedNumber(10);
                System.out.println("Multiplied result="+result1);

                pOb=new ChildCov();
                result1=pOb.getMultipliedNumber(10);
                System.out.println("Multiplied result="+result1);
        }
}
```

Output

```
<terminated> CovarianceEx [Java Application] C:\Program Files\Java\jdk1.8.0_45\bin\javaw.exe (Sep 21, 2016, 4:54:12 PM)
***Datatype :primitive(int)***

Inside Parent
Multiplied result=17
Inside Child
Multiplied result=500
```

Analysis

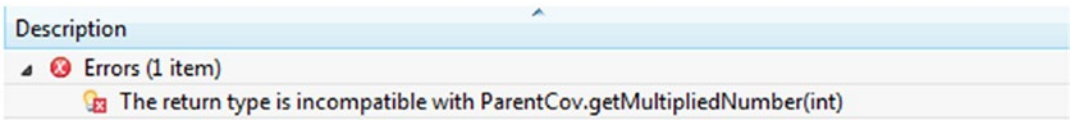
You must notice that 2 lines are commented in this section:

```
int getMultipliedNumber(int x )
//error:Return type is incompatible
//double getMultipliedNumber(int x )
```

If you change the return type int with `double` like this:

```
// int getMultipliedNumber(int x )
//error:Return type is incompatible
double getMultipliedNumber(int x )
```

You'll get the compilation error (the return type is incompatible…):



Now suppose you are dealing with methods that return class names as their return types. In this case, you'll not receive the similar kind of error if you vary the return type in the direction of subclass. So, let's go through the modified program (to show you the difference between these two, we have kept previous parts untouched and added new codes to it).

Demonstration-11

```
//Modified Version

package overriding.examples;

class ParentCov
{
        int i;
        int getMultipliedNumber(int x )
        {
                System.out.println("Inside Parent");
                this.i=x;
                return (int) (i*1.75);
        }
        ParentCov getMultipliedNumber(int x,int y )
        {
                System.out.println("Inside Parent-overloaded version");
                this.i=x*y;
                return this;
        }
}
class ChildCov extends ParentCov
{
        int getMultipliedNumber(int x )
        //error:Return type is incompatible
        //double getMultipliedNumber(int x )
        {
                System.out.println("Inside Child");
                this.i=x;
                return  i*50;
        }
        ChildCov getMultipliedNumber(int x,int y )
        {
                System.out.println("Inside Child- overloaded version");
                this.i=x*y*25;
                return this;
        }
}

public class CovarianceEx {
        public static void main(String args[])
        {
                System.out.println("*** Testing the covariance return type in Java ***");
                System.out.println("***Datatype :primitive(int)***\n");
                ParentCov pOb=new ParentCov();
                int result1=pOb.getMultipliedNumber(10);
                System.out.println("Multiplied result="+result1);

                pOb=new ChildCov();
                result1=pOb.getMultipliedNumber(10);
                System.out.println("Multiplied result="+result1);
```

```
                System.out.println("\n***Testing covariance now.Datatype: non-primitive***\n");

                ParentCov pOb2=new ParentCov();
                pOb2=pOb2.getMultipliedNumber(10,20);
                System.out.println("Multiplied result="+pOb2.i);

                ParentCov pOb3=new ChildCov();
                pOb3=pOb3.getMultipliedNumber(10,20);
                System.out.println("Multiplied result="+pOb3.i);
        }
}
```

Output

```
<terminated> CovarianceEx [Java Application] C:\Program Files\Java\jdk1.8.0_45\bin\javaw.exe (Sep 21, 2016, 5:10:27 PM)
*** Testing the covariance return type in Java ***
***Datatype :primitive(int)***

Inside Parent
Multiplied result=17
Inside Child
Multiplied result=500

***Testing covariance now.Datatype: non-primitive***

Inside Parent-overloaded version
Multiplied result=200
Inside Child- overloaded version
Multiplied result=5000
```

Analysis

Notice the return type of the method:

```
ChildCov getMultipliedNumber(int x,int y )
```

You can see that this time instead of `ParentCov`, we have used `ChildCov` as the return type but the compiler did not complain about this though it complained for primitive data types.

This technique is known as covariant return type in Java.

Students ask:

Sir, why Java started supporting this concept?

Teacher says: If you notice that subtype also belongs to a subclass of a parent type e.g. go back to our earlier example: A `Cricketer` is a `Player` but the reverse is not true i.e. we can substitute a cricketer as a player. So, in some real world scenario also, it makes sense that we can use the subtype as a parent type in appropriate places.

CHAPTER 6

Abstract Class

Sometimes we start doing some work with an expectation that our incomplete work will be carried out by someone else. A real life example can be seen in case of properties purchases and modeling those. It is very common that many of our grandparents may bought some properties earlier, then our parents made a small house in that property and ultimately we give the house a larger shape or we redecorate the house. So basic idea is same: we want someone to continue and complete the incomplete work first. We give them freedom that upon completion, they can remodel as per their needs. The concepts of abstract class suits best in such type of scenarios in the programming world.

These are incomplete classes and we cannot instantiate objects from this type of classes. The child of those classes must complete them first and then they can redefine some of the methods (by overriding).

In general, if a class contains at least one incomplete/abstract method, the class itself is an abstract class. By the term "abstract method"- we mean that the method has the declaration (or signature) but no implementation.

The technique is useful when the super class can define a generalized form (that will be shared by its subclasses) and passes the responsibilities to fill the details to its subclasses.

Here is the implementation:

Demonstration-1

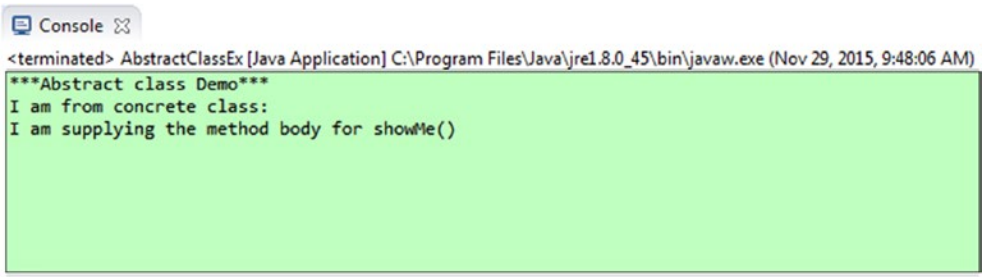
A simple abstract class demo

```
package abstractclasses.examples;

abstract class MyAbstractClass
{
        public  abstract void showMe();
}
class MyConcreteClass extends MyAbstractClass
{
        @Override
        public void showMe()
        {
                System.out.println("I am from concrete class:");
                System.out.println("I am supplying the method body for showMe()");

        }
}
```


V. Sarcar, *Interactive Object Oriented Programming in Java*, DOI 10.1007/978-1-4842-2544-8_6

```
class AbstractClassEx1
{
        public static void main(String Args[])
        {
                System.out.println("***Abstract class Demo***");
                //Illegal:Cannot instantiate
                //MyAbstractClass abstractOb=new MyAbstractClass();
                MyConcreteClass concreteOb=new MyConcreteClass();
                concreteOb.showMe();
        }
}
```

Output



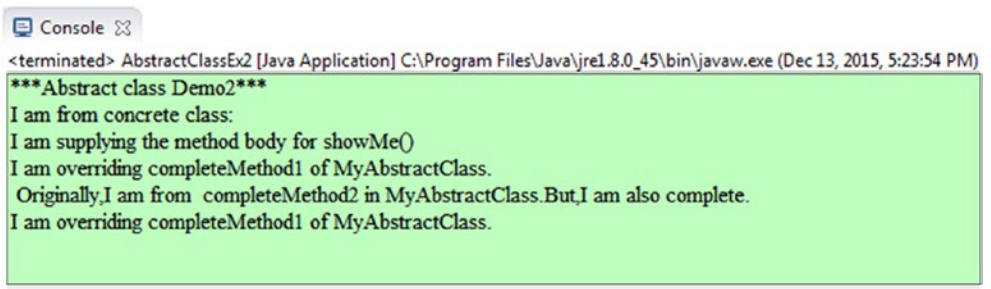
An abstract class can contain concrete methods also. The child class may or may not override those methods. Here is another implementation.

Demonstration-2

```
package abstractclasses.examples;
abstract class AbstractClass
{
        public  abstract void showMe();
        public void completeMethod1()
        {
                System.out.println(" Originally,I am from completeMethod1 in
                  MyAbstractClass.But,I am complete.");
        }
        public void completeMethod2()
        {
                System.out.println(" Originally,I am from  completeMethod2 in
                  MyAbstractClass.But,I am also complete.");
        }
}
class ConcreteClass extends AbstractClass
{
        @Override
        public void showMe()
        {
```

```
                System.out.println("I am from concrete class:");
                System.out.println("I am supplying the method body for showMe()");

        }
        //It wants to override completeMethod1() in MyAbstractClass
        public void completeMethod1()
        {
                System.out.println("I am overriding completeMethod1 of MyAbstractClass.");
        }
}
class AbstractClassEx2
{
        public static void main(String Args[])
        {
                System.out.println("***Abstract class Demo2***");
                ConcreteClass concreteOb=new ConcreteClass();
                concreteOb.showMe();
                //It will show that completeMethod1 is redefined in MyConcreteClass.
                concreteOb.completeMethod1();
                //It will show the details of completeMethod2 defined in MyAbstractClass.
                concreteOb.completeMethod2();
                //Following declaration will be fine
                AbstractClass abstractRef=new ConcreteClass();
                abstractRef.completeMethod1();
        }
}
```

Output



Students ask:

Can we implement the concept of dynamic method dispatch here?
Yes. Following declaration will be perfectly fine and it can call `CompleteMethod1` of the `ConcreteClass`.

through the below codes:

```
AbstractClass abstractRef=new ConcreteClass();
abstractRef.completeMethod1();
```

Students ask:

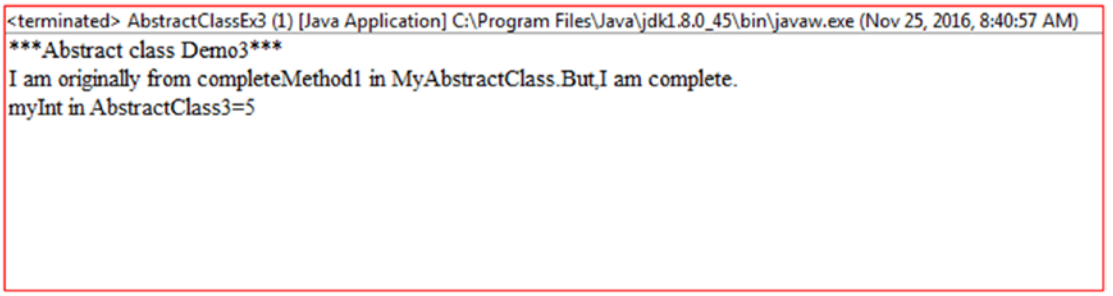
Can an abstract class contain fields?
Yes.

Following example will demonstrate how we can use the concept of dynamic method dispatch here. Also, the program will show that an abstract class contain fields.

Demonstration-3

```
package abstractclasses.examples;
abstract class AbstractClass3
{
        public int myInt=5;
        public  abstract void showMe();
        public void completeMethod1()
        {
                System.out.println("I am originally from completeMethod1 in MyAbstractClass.
                But,I am complete.");
        }
}
class ConcreteClass3 extends AbstractClass3
{
        @Override
        public void showMe()
        {
                System.out.println("I am from concrete class:");
                System.out.println("I am supplying the method body for showMe()");

        }
}

class AbstractClassEx3
{
        public static void main(String Args[])
        {
                System.out.println("***Abstract class Demo3***");
                AbstractClass3 abstractRef=new ConcreteClass3();
                abstractRef.completeMethod1();
                System.out.println("myInt in AbstractClass3="+abstractRef.myInt);
        }
}
```

Output



Students ask:

In the above example, the access modifier is public. Is it mandatory?
Teacher says: Not at all. We can use non-public access modifiers also. Later you'll learn that it is one of the key differences with interfaces.

Students ask:

Suppose in a class we have 10+ methods and out of that only one is an abstract method. Still we need to mark the class as abstract?
Teacher says: Yes. If a class contains at least one abstract method, the class itself is abstract. You can think from a general point of view-an abstract keyword is used in a sense to represent the incompleteness. So, if your class contains one incomplete method, your class itself is incomplete and hence need to mark by the keyword `abstract`.

■ **Note** So, the simple formula is: whenever your class has at least an abstract method, your class itself is an abstract class.

Teacher asks:

Now consider a reverse scenario. Suppose, you have marked your class abstract but there is no abstract method in it like this:

```
abstract class AbstractClass
{
        public void completeMethod1()
        {
                System.out.println("A complete method");
        }
        public void completeMethod2()
        {
                System.out.println("Another complete method.");
        }
}
```

Can we compile the program?
Teacher says: Yes. Still it will compile but till this point, you cannot create object for this class.

Students ask:

So sir, how can we create objects from an abstract class?
Teacher says: We mentioned already that we cannot create objects from an abstract class.

Students ask:

Sir, it appears to me that an abstract class has virtually no use if it is not extended. Is the understanding correct?
Yes.

Students ask:

If a class extends an abstract class, it has to implement all the abstract methods.Is the understanding correct?

Teacher says: It may or may not implement all the abstract methods in the parent class. The simple formula is that if you want to create objects of a class, the class needs to be completed i.e. it should not contain any abstract methods. So, if the child class cannot provide implementation (i.e. body) of all the abstract methods, it should be marked again with the keyword `abstract` like the below example.

```
abstract class AbstractClass
{
        public abstract void inCompleteMethod1();
        public abstract void inCompleteMethod2();
}
abstract class child1 extends AbstractClass
{
        //Here our child class is implementing only one of the abstract methods.
        //So, the class is abstract again.
        @Override
        public void inCompleteMethod1()
        {
                System.out.println("Implementing the inCompleteMethod1()");

        }
}
```

Students ask:

A concrete class is a class which is not abstract-is the understanding correct?
Teacher says: Yes.

Students ask:

Can we tag a method with both abstract and final?
Teacher says: No. Just think, by declaring abstract, you want overriding and by declaring final, you want to prevent overriding. i.e. you cannot do both at the same time.

Quiz

Teacher asks:

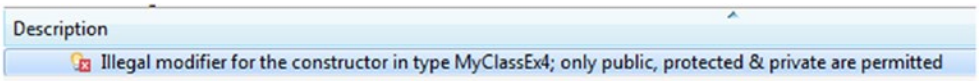
Can you predict the output?

```
package abstractclasses.examples;

class MyClassEx4
{
        //Constructors cannot be final/abstract/static
        abstract MyClassEx4()
        {
                System.out.println("I am a no argument constructor");
        }
}
class ExperimentWithConstructorEx4
{
        public static void main(String args[])
        {
```

```
                System.out.println("*** Experiment with constructors  ***");
                System.out.println("***Question:Can construcors be abstract?  ***");
                MyClassEx4 myOb=new  MyClassEx4();
        }
}
```

Output

Compilation error.



Students ask:

Sir, why constructors cannot be abstract?

Teacher says: Think from a general point of view. The keyword `abstract` with a method is used to mean that the method will be overriding somewhere in a child class. But constructors cannot be overridden as per the language specification.

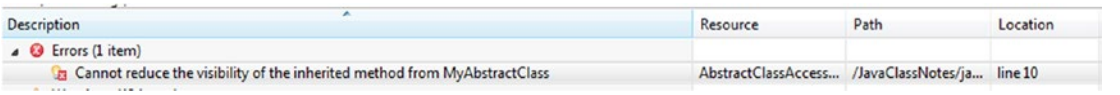
Quiz

Teacher asks:

Can you predict the output?

```
package abstractclasses.examples;

abstract class MyAbstractClass
{
        public abstract void showMe();
}
class MyConcreteClass extends MyAbstractClass
{
        @Override
        protected void showMe()
        {
                System.out.println("I am from concrete class:");
                System.out.println("I am supplying the method body for showMe()");

        }
}

public class AbstractClassAccessModifierEx {
        public static void main(String args[])
        {
                MyAbstractClass myref=new MyConcreteClass();
                myref.showMe();
        }
}
```

Output

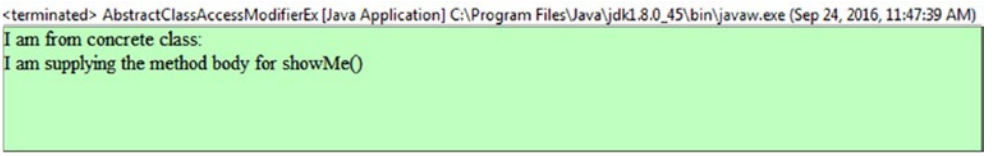


■ **Note** We have 2 solutions to remove the problem. *Either in the parent class, we use the modifier protected or in the child class, we can use the modifier public*. Just think, if you are implementing dynamic method dispatch and suddenly at runtime you discover that you do not have enough visibility-Then what will happen? This will be a definite source of problem then.

Students ask:

Sir, what will happen if we do the reverse (i.e. using protected in parent class and public in child class in the above scenario)?

Teacher says: In this case, you are basically increasing the visibility, so there can be no issue in runtime like above. So, compiler will allow you to do this change and you will receive the following output:



CHAPTER 7

Interface

With the interface, we declare what we are going to implement but we are not specifying how we are going to do that. These are similar to classes but with no instance variables and all of their methods are declared without a body (i.e. methods are actually abstract).

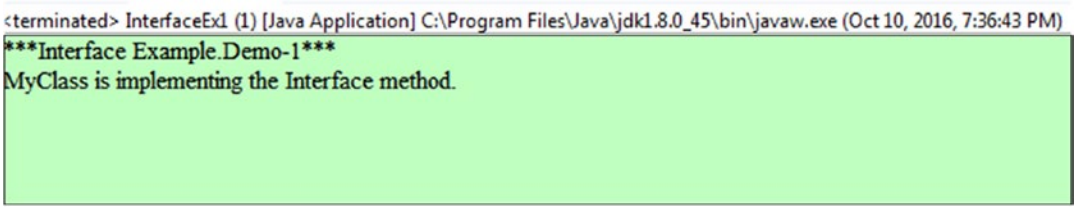
We can support dynamic method resolution during run time with the help of interfaces. Once defined, a class can implement any number of interfaces.

Demonstration-1

```
package interfaces.examples;
interface MyInterface
{
        void show();
}
class MyClass implements MyInterface
{
        @Override
        public void show()
        {
                System.out.println("MyClass is implementing the Interface method.");
        }
}

public class InterfaceEx1
{
        public static void main(String args[])
        {
                System.out.println("***Interface Example.Demo-1***");
                MyClass myClassOb=new MyClass();
                myClassOb.show();
        }

}
```


V. Sarcar, *Interactive Object Oriented Programming in Java*, DOI 10.1007/978-1-4842-2544-8_7

Output



Students ask:

Sir, if the methods are incomplete then a class who is using the interface needs to implement all the methods in the interfaces. Is the understanding correct?

Teacher says: Exactly. And if the class cannot implement all of them, it will announce its incompleteness by marking itself abstract. Following example will help you to understand better.

Here is implementation 2:

The interface has two methods. But a class is implementing only one. Then the class itself becomes abstract.

Demonstration-2

```
package interfaces.examples;

interface MyInterface2
{
        void show1();
        void show2();
}
abstract class MyClass2 implements MyInterface2
{
        @Override
        public void show1()
        {
                System.out.println("MyClass2 is implementing the show1() method.");
        }
}
```

■ **Note** So the formula is: A class needs to implement all the methods defined in the interface. Otherwise, it will be an abstract class.

Students ask:

Sir, you said earlier that interfaces can help us to implement the concept of multiple inheritance. Then can our class implement two or more interfaces?

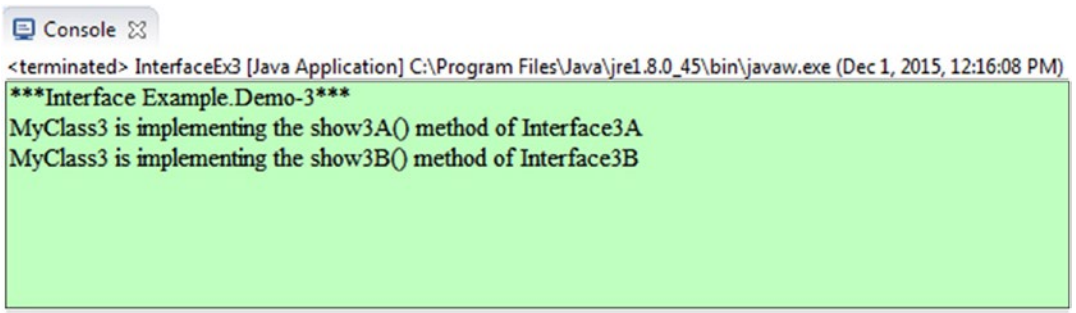
Teacher says: Yes. Following example will show you how to do that.

In Demonstration 3, a class is implementing multiple interfaces.

Demonstration-3

```
package interfaces.examples;
interface MyInterface3A
{
        void show3A();
}
interface MyInterface3B
{
        void show3B();
}
class MyClass3 implements MyInterface3A,MyInterface3B
{
        @Override
        public void show3A()
        {
                System.out.println("MyClass3 is implementing the show3A() method of Interface3A");
        }
        @Override
        public void show3B() {
                System.out.println("MyClass3 is implementing the show3B() method of Interface3B");

        }
}
public class InterfaceEx3 {
        public static void main(String args[])
        {
                System.out.println("***Interface Example.Demo-3***");
                MyClass3 myClassOb=new MyClass3();
                myClassOb.show3A();
                myClassOb.show3B();
        }
}
```

Output



Students ask:

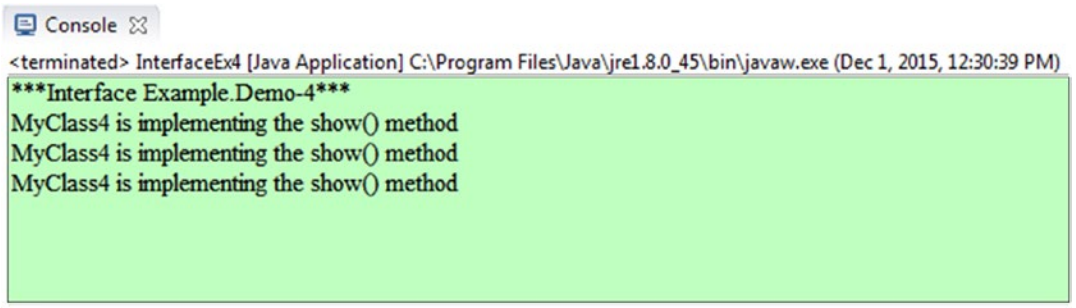
In the above program, method names were different in interfaces. But if both of the interfaces contain the same method name, can we implement them?

Teacher says: Very good question. Yes we can but in that case, the class needs to implement its own implementation for the same named method:

Demonstration-4

```
package interfaces.examples;

//Both of the interface have the same method name "show()".
interface MyInterface4A
{
        void show();
}
interface MyInterface4B
{
        void show();
}
class MyClass4 implements MyInterface4A,MyInterface4B
{
        @Override
        public void show()
        {
                System.out.println("MyClass4 is implementing the show() method ");
        }

}
public class InterfaceEx4 {
        public static void main(String args[])
        {
                System.out.println("***Interface Example.Demo-4***");

                //All the 3 callings are legal.
                MyClass4 myClassOb=new MyClass4();
                myClassOb.show();

                MyInterface4A inter4A=myClassOb;
                inter4A.show();

                MyInterface4B inter4B=myClassOb;
                inter4B.show();

        }
}
```

Output



Students ask:

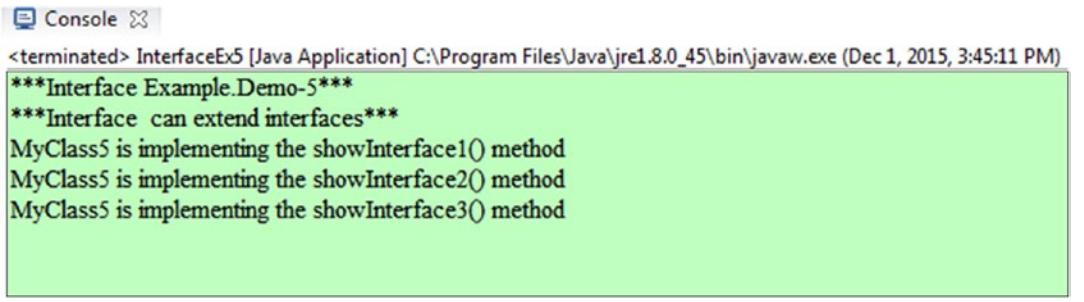
Can an interface extend/implement another interface?
Teacher says: It can extend but not implement (see Demonstration 5).

Demonstration-5

```
package interfaces.examples;

interface Interface1
{
        void showInterface1Method();
}
interface Interface2
{
        void showInterface2Method();
}
//Interface extending another interfaces
interface Interface3 extends Interface1,Interface2
{
        void showInterface3Method();
}
class MyClass5 implements Interface3
{
        //Now MyClass5 needs to implement methods from Interface1,Interface2 and Interface3
        @Override
        public void showInterface1Method() {
                System.out.println("MyClass5 is implementing the showInterface1() method ");

        }
        @Override
        public void showInterface2Method() {
                System.out.println("MyClass5 is implementing the showInterface2() method ");
        }
```

```
        @Override
        public void showInterface3Method() {
                System.out.println("MyClass5 is implementing the showInterface3() method ");

        }
}
public class InterfaceEx5 {
        public static void main(String args[])
        {
                System.out.println("***Interface Example.Demo-5***");
                System.out.println("***Interface  can extend interfaces***");
                MyClass5 myClassOb=new MyClass5();
                myClassOb.showInterface1Method();
                myClassOb.showInterface2Method();
                myClassOb.showInterface3Method();
        }
}
```

Output



Tagging Interface

Teacher asks:

What is a tag/tagging Interface?
Answer: An interface which is empty is termed as a tag/tagging interface.

```
//tagging interface example
interface ITaggingInterface
{
}
```

Teacher asks:

Why we need a tagging interface?

- We can create a common parent.

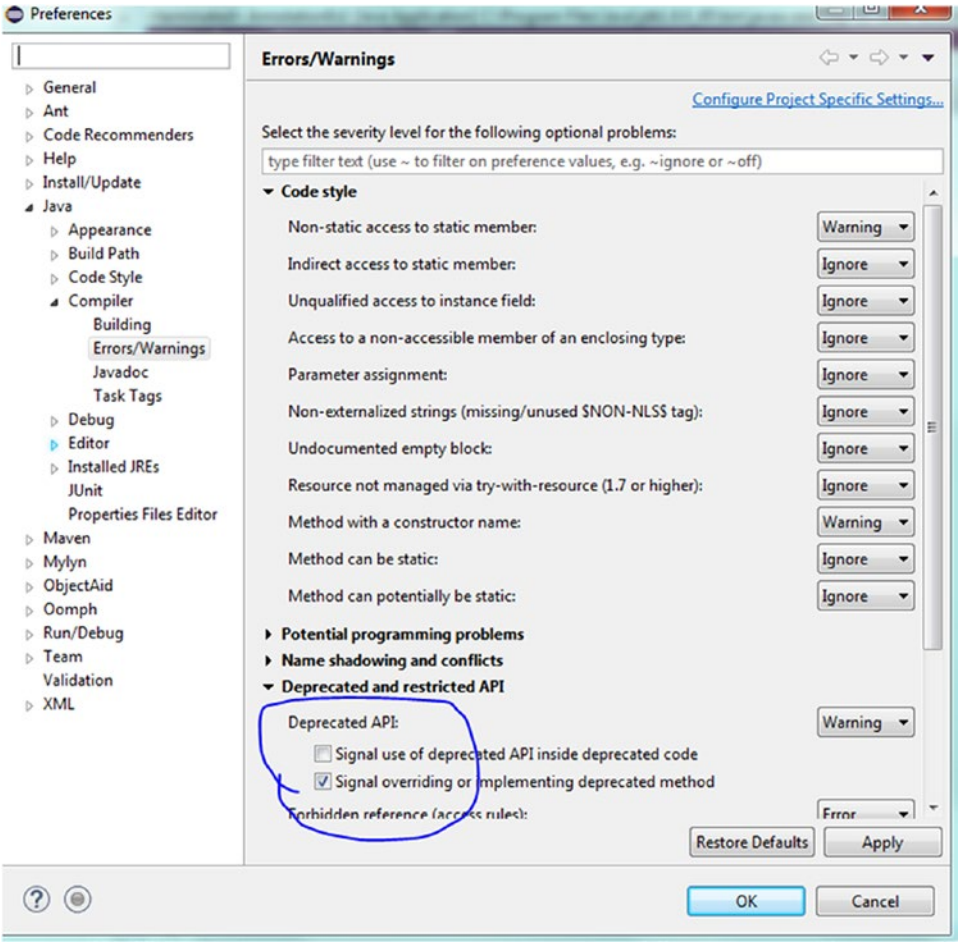
- A class can claim membership in the set e.g. if our class implements the `Serializable` interface, it becomes serializable. So, our class actually becomes an interface type through polymorphism. Even a class that is implementing a tagging interface, need not define any new method because the interface itself does not have any such method.
- Later you'll know that we can implement thread safety through marker interfaces.

The concept of annotation is more popular than marker interfaces JDK5 onwards we get the concept of annotations. We have also used built in annotation `@Override` in some of our programs.

Basically we can add some metadata information with our source code with this technique.e.g.to mark a deprecated method we can use `@Deprecated` as below. To show you the visual behavior in Eclipse IDE, we have taken a snapshot of the program:

```
1  package javaclassnotes.programs;
2
3  interface IAnnotationDemo
4  {
5     /**
6      * @deprecated
7      * Please use our new method myNewMethod() instead of this.
8      */
9     @Deprecated
10    public void myOldMethod();
11    /**
12     * This is our new method.
13     */
14    public void myNewMethod();
15
16 }
17 class AnnotationDemo implements IAnnotationDemo
18 {
19    public void myOldMethod(){
20       System.out.println("I am an old method.I am retired");
21    }
22    public void myNewMethod(){
23       System.out.println("I am the modified method for myOldMethod().Use me now onwards.");
24    }
25 }
26 public class AnnotationEx1 {
27    public static void main(String args[])
28    {
29       System.out.println("***Simple use of Annotation in Java***");
30       AnnotationDemo anOb=new AnnotationDemo();
31       anOb.myOldMethod();
32       anOb.myNewMethod();
33    }
34 }
```

We can see the warning messages in the above program due to use of the deprecated method. If you can not see the message, just you check whether the following checkbox is checked for you:



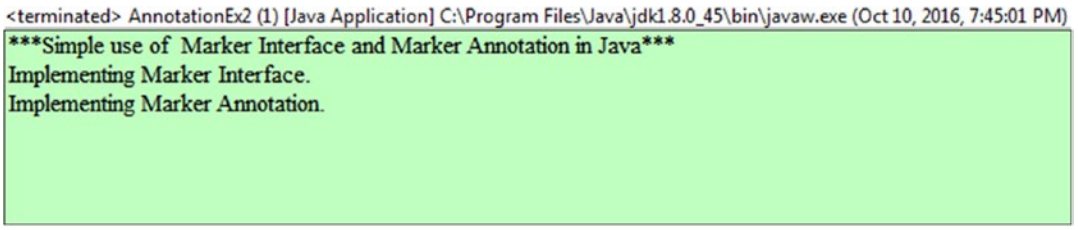
Following program is presented for your ready reference and comparison of uses of a simple marker interface and a marker annotation:

Demonstration-Marker Interface and Annotation

```
package interfaces.examples;
//Marker interface
interface IMarkerInterfaceDemo{}

//Marker Annotation
@interface IMarkerAnnotation{}

class Annotation2 implements IMarkerInterfaceDemo
{
        public void myInterfaceMethod(){
                System.out.println("Implementing Marker Interface.");
        }
}
```

```
class Annotation3
{
        @IMarkerAnnotation public void myInterfaceMethod(){
                System.out.println("Implementing Marker Annotation.");
        }
}
public class AnnotationEx2 {
        public static void main(String args[])
        {
                System.out.println("***Simple use of  Marker Interface and Marker Annotation
                  in Java***");
                Annotation2 anOb2=new Annotation2();
                Annotation3 anOb3=new Annotation3();
                anOb2.myInterfaceMethod();
                anOb3.myInterfaceMethod();
        }
}
```

Output



■ **Note** We must note that `@Override`, `@Deprecated` are also examples of marker annotations. The detail discussions of annotations is out of scope of this book.

Teacher asks:

Can you tell me: What is the difference between an abstract class and an interface?

- An abstract class can have concrete methods in it but an interface cannot have that. We'll come to this point later. Now in Java 8, we have a keyword called `default`. We can use this keyword in an interface to provide some default implementation, see below in our implementation 6.
- An abstract class can have only one parent class (can extend from another abstract class or concrete class), an interface can have multiple parent interfaces. An interface can extend from other interface/s only.
- Members of an interface is by default public. An abstract class can have other flavors e.g. private, protected etc.
- Variables in an interface is by default static final. An abstract class can have non-final as well as final variables.

Students ask:

Sir, then how we decide-whether we should use an abstract class or an interface?

Good question. I believe that if we want to have some centralized or default behavior/s, abstract class is a better choice .Because here we can provide some default implementation(in case of abstract class). On the other hand, interface implementation starts from a scratch. They indicate some kind of rules-what to be done (e.g. you must implement the method) but they will not enforce you how to be done. Also interfaces are preferred when we are trying to implement the concept of multiple inheritance.

But at the same time we also remember that if we need to add a new method in an interface, then we need to track down all the implementation/s of that interface and we need to put the concrete implementation for that method in all those places. An abstract class is ahead here-we can add a new method in an abstract class with a default implementation and our existing code can run smoothly.

So, now Java has taken special care to this point and Java 8 has introduced the use of `default` keyword. In Java 8, we can prefix the word `default` before our intended method signature and can provide a default implementation. Interface methods are public by default, so, we do not need to mark it by the keyword `public`.

The Java documentation also briefly summarizes as below:

We should give preferences to abstract for these scenarios:

- We want our code sharing among multiple closely related classes.
- We expect that classes that extend our abstract class can have many common methods or fields, or they require non-public access modifiers inside them.
- We want to use non-static or/and non-final fields which enables us to define methods that can access and modify the state of the object to which they belong.

On the other hand, we should give preferences to interfaces for these scenarios:

- You expect that several unrelated classes are going to implement your interface e.g. Comparable interface can be implemented by many unrelated classes.
- We want to specify the behavior of a particular data type, but not concerned about the implementer.
- We want to use the concept of multiple inheritance of type in the application.

Students ask:

Sir, can you show us an example of the usage of `default` keyword inside an interface?

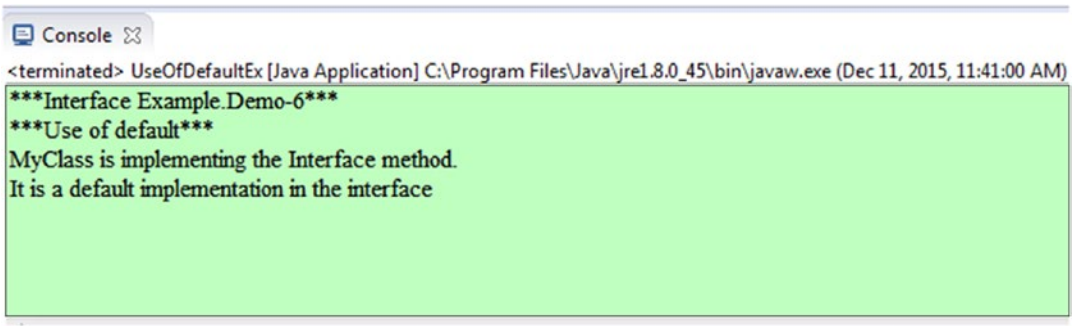
Teacher says: Consider the below example (Demonstration 6)

Demonstration-6

```
package interfaces.examples;
interface MyDefaultInterface
{
        void show();
        default void defaultMethod()
        {
                System.out.println("It is a default implementation in the interface");
        }
}
```

```
class MyClass6 implements  MyDefaultInterface
{
        @Override
        public void show()
        {
                System.out.println("MyClass is implementing the Interface method.");
        }
}

public class UseOfDefaultEx
{
        public static void main(String args[])
        {
        System.out.println("***Interface Example.Demo-6***");
                System.out.println("***Use of default***");
                MyDefaultInterface interfaceOb=new MyClass6();
                interfaceOb.show();
                interfaceOb.defaultMethod();
        }
}
```

Output



Now we can see that `MyClass6` has implemented only `show()` method but still the program can run without a compilation error.

Students ask:

Sir, can we override the default method in an interface?
Teacher says: Yes. We can do that.
Consider the below example (Demonstration 7) and corresponding output. Here we have added a method inside `MyClass7` to override the default method in the interface `MyDefaultInterface2`.

Demonstration-7

```
package interfaces.examples;

interface MyDefaultInterface2
{
        void show();
        default void defaultMethod()
        {
                System.out.println("It is a default implementation in the interface");
        }
}
class MyClass7 implements  MyDefaultInterface2
{
        @Override
        public void show()
        {
                System.out.println("MyClass is implementing the Interface method.");
        }
        @Override
        public void defaultMethod()
        {
                System.out.println("MyClass is overriding the default Interface method.");
        }
}

public class UseOfDefaultEx2
{
        public static void main(String args[])
        {
                System.out.println("***Interface Example.Demo-7***");
                System.out.println("***Use of default.Ex-2***");
                MyDefaultInterface2 interfaceOb=new MyClass7();
                interfaceOb.show();
                interfaceOb.defaultMethod();
        }
}
```

Output

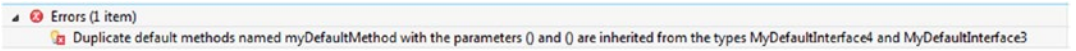
```
***Interface Example.Demo-7***
***Use of default***
MyClass is implementing the Interface method.
MyClass is overriding the default Interface method.
```

Students ask:

Sir, with the use of default, are we not going back to the diamond problem with multiple inheritance?

Teacher says: No. Here is the trick. Java puts a restriction that if a class is implementing from 2 (or more) interfaces where each interface has its own default implementation with same method name, the class needs to implement its own implementation for the same named method, otherwise we'll receive a compilation error.

Consider the below example (implementation 8). Each of the interfaces has a default method namely `myDefaultMethod()`. Now `MyClass8` is implementing both of them. So, to avoid the conflict, it must provide its own implementation for `myDefaultMethod()`, otherwise we'll receive following compilation error:



Demonstration-8

```
package interfaces.examples;

interface MyDefaultInterface3
{
        void show();
        default void myDefaultMethod()
        {
                System.out.println("Default implementation for interface3");
        }
}
interface MyDefaultInterface4
{
        void show();
        default void myDefaultMethod()
        {
                System.out.println("Default implementation for interface4");
        }
}
class MyClass8 implements  MyDefaultInterface3,MyDefaultInterface4
{
        @Override
        public void show()
        {
         System.out.println("MyClass is implementing the Interface method.");
        }

        @Override
        public void myDefaultMethod()
            {
          System.out.println("MyClass8 needs to implement this method");
        }
}
```

```
public class UseOfDefaultEx3
{
        public static void main(String args[])
        {
                System.out.println("***Interface Example.Demo-8***");
                System.out.println("***Use of default.Ex-3***");
                MyDefaultInterface3 interfaceOb3=new MyClass8();
                interfaceOb3.show();
                interfaceOb3.myDefaultMethod();
        }
}
```

Output

```
<terminated> UseOfDefaultEx3 (1) [Java Application] C:\Program Files\Java\jdk1.8.0_45\bin\javaw.exe (Oct 10, 2016, 7:51:42 PM)
***Interface Example.Demo-8***
***Use of default.Ex-3***
MyClass is implementing the Interface method.
MyClass8 needs to implement this method
```

Students ask:

Sir, looks like the defaults methods in the interfaces are not used at all in the above program. Is there any way to call those methods?

Teacher says: Definitely. You can modify the `myDefaultMethod()` for `MyClass8` as below to call those interfaces method

```
@Override
public void myDefaultMethod()
{
        System.out.println("MyClass8 needs to implement this method");
        //Calling default method of MyDefaultInterface3
        MyDefaultInterface3.super.myDefaultMethod();
        //Calling default method of MyDefaultInterface4
        MyDefaultInterface4.super.myDefaultMethod();
}
```

Now you'll get output like this:

```
***Interface Example.Demo-7***
***Use of default.Ex-3***
MyClass is implementing the Interface method.
MyClass8 needs to implement this method
Default implementation for interface3
Default implementation for interface4
```

Students ask:

Sir, can we make the interface final?

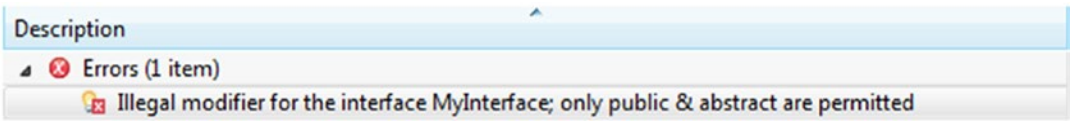
Teacher says: If you make the interface final, then who will implement the incomplete methods of that interface? We must remember that before Java 8, static methods (we'll discuss later about them) were not supported in interfaces. So, basically, there was no point to make an interface final.

For the below declaration, Eclipse IDE also raises the error:

Demonstration-9

```
final interface MyInterface

{
        void show();
}
```

Output



Students ask:

Sir, can we use the keyword abstract before the interface method?

Teacher says: Actually there is no need to do that. Because by default, they are abstract. But compiler will not raise any issue here.

```
interface MyInterface
{
        //void show();
        //no need to mention abstract
        abstract void show();
}
```

Students ask:

Sir, can we use constants inside interfaces?

Teacher says: Yes. They are by default public, static and final. So, we can omit these modifiers.

Assignment

You have two classes- A and B. Class A contains an abstract method `showA()`.You also have an interface called `Inter`. In `Inter`, you have a method `showInter()`. Now write a simple program where B will implement the methods defined in A and `Inter`.

CHAPTER 8

Package

Consider a simple scenario. Can you use the same class name twice in a Java file? No. Compiler will raise the issue and it will point towards this naming collision. So we need to choose unique naming conventions each time whenever we are going to define a class. But we must remember that in real world programming, class name should be meaningful enough and so there is a possibility that two different programmer in a project are going to choose the same name for their class. Then how we can deal with those situations? Package will rescue us in those scenarios.

We can bundle our classes/interfaces etc. inside our own packages. Packages help us to avoid naming conflicts and/or to control the visibility. We can control the visibility inside a package in such a way that our particular class may or may not be exposed to outside world (both inside and outside packages).

Packages are reflected as directories. Creating a package in Eclipse is quite easy. We do not even think about how Java runtime is going to find the proper packages or classes inside it. Otherwise, we need to put special attention to the `CLASSPATH` environment variable.

We must remember about the following points:

- `package` statement should be on top of our source file. If we do not explicitly define this statement, then all the classes/interfaces etc. will be in the current default package.
- When one class refers another class inside the same package, `package` statement need not to be included. Otherwise we need to use fully qualified class name like `packagename.classname` or we need to use `import` statement.
- Whole package can be imported like:

```
import packagename.*;
```

Or, if we want to import only a particular class from a package, use:

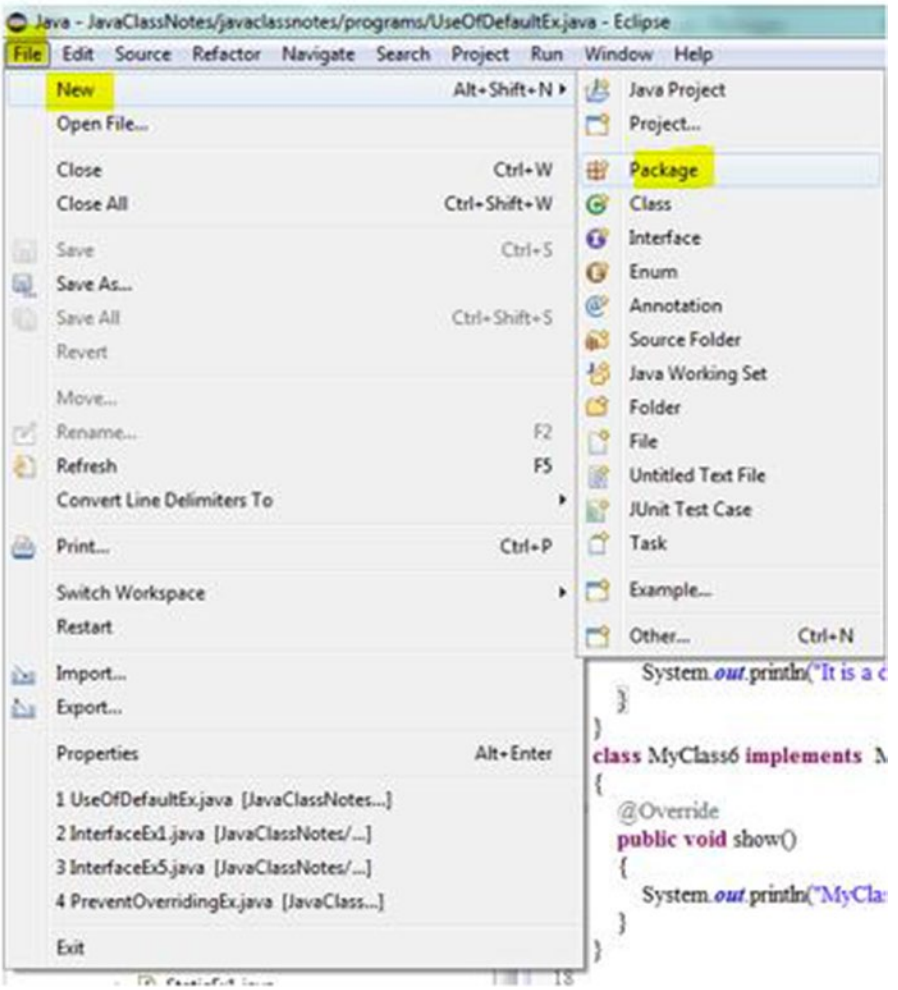
```
import packagename.classname;
```

- The name of the package must follow the directory structure.

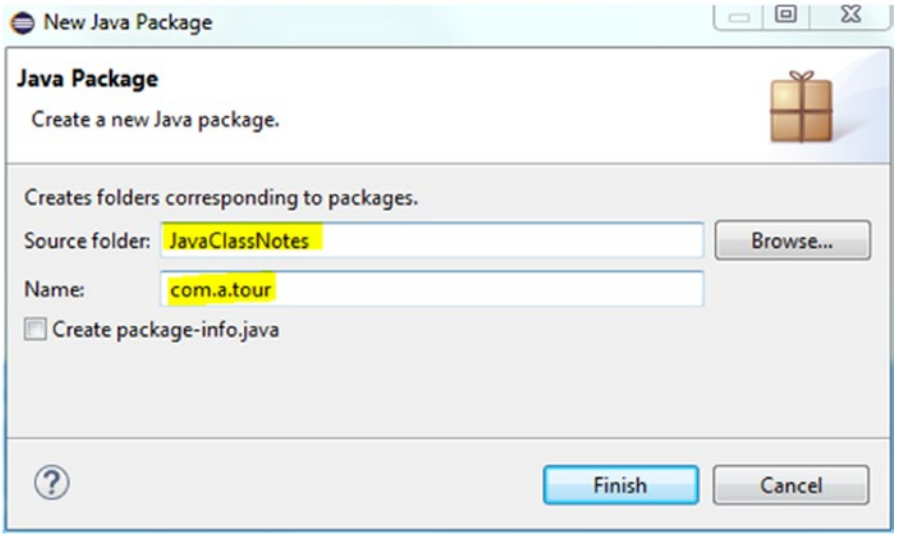
V. Sarcar, *Interactive Object Oriented Programming in Java*, DOI 10.1007/978-1-4842-2544-8_8

Here is an example on how to create a package in Eclipse IDE:

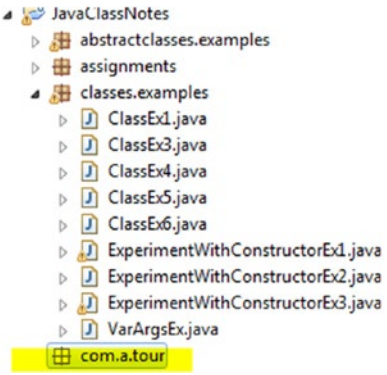
1. Click File menu ➤ New ➤ Package



2. Supply the required information and click finish.



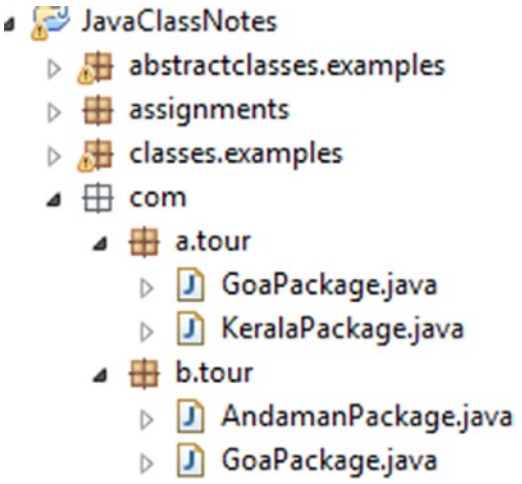
3. Now you'll see the package in your package explorer view e.g. it may look like as below:



■ **Note** The newly created package is empty. But other packages in the snapshots already have some classes inside them. Those packages were created earlier.

---

4. Now right-click the package name ➤ New ➤ Class/Package etc to put classes/sub packages etc. inside the created package. It may have the following structure:



■ **Note** here in this package, already there are some classes-which we made earlier.

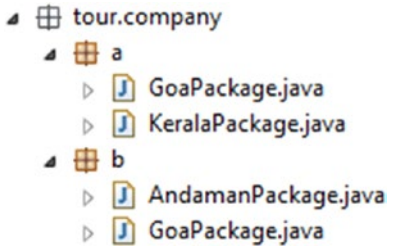
Demonstration-1

Now let us go through an example. Consider two travel companies-a and b. Company a conducts tours for Goa and Kerala. Company b conducts tours for Goa and Andaman. Any tourist can seek information from them for any particular tour package.

Here we have covered both of the following scenarios:

- Less challenging situation: Only Company a conducts tour for Kerala and Only company b conducts tour for Andaman.
- More challenging situation: Notice that both of the companies are providing tour for Goa. And we need to get the information through the GoaPackage Class. See both the packages are using the same class name.

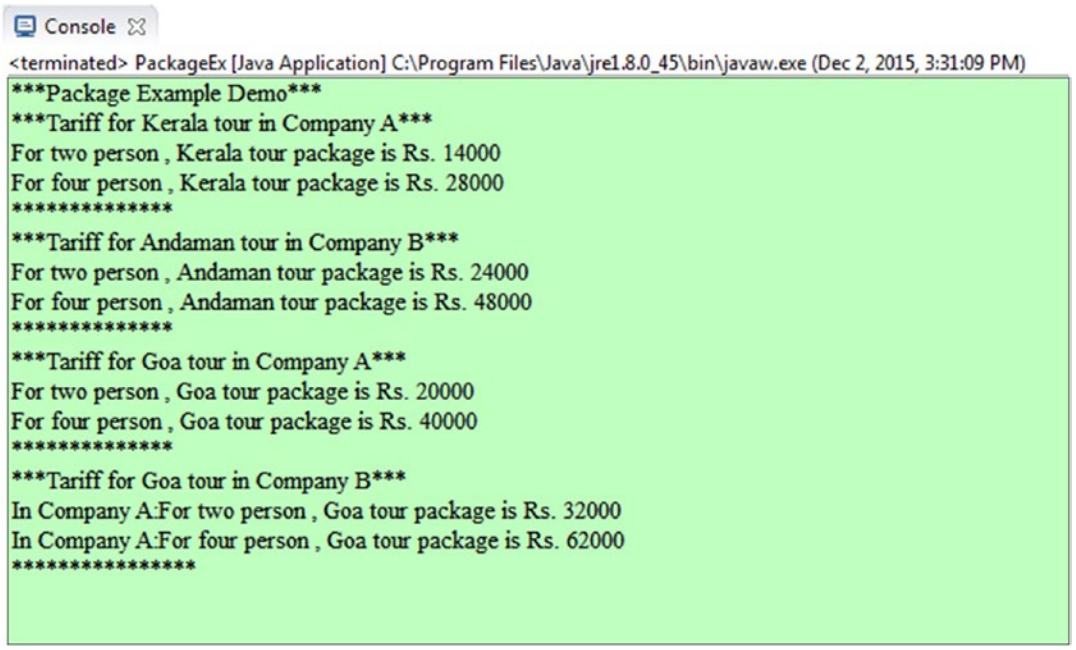
Here is the Eclipse package explorer view



```
// GoaPackage.java [For Company A, in com.a.tour package]
package com.a.tour;
public class GoaPackage
{
        int basic_price=10000;
        public void ShowPrice()
        {
                System.out.println("***Tariff for Goa tour in Company A***" );
                System.out.println("For two person , Goa tour package is Rs. "+ basic_price*2 );
                System.out.println("For four person , Goa tour package is Rs. "+ basic_price*4 );
                System.out.println("**************" );
        }
}

// KeralaPackage.java [For Company A, in com.a.tour package]
package com.a.tour;

public class KeralaPackage
{
        int basic_price=7000;
        public void ShowPrice()
        {
                System.out.println("***Tariff for Kerala tour in Company A***" );
                System.out.println("For two person , Kerala tour package is Rs. "+ basic_price*2 );
                System.out.println("For four person, Kerala tour package is Rs. "+ basic_price*4 );
                System.out.println("**************" );
        }
}
// AndamanPackage.java [For Company B, in com.b.tour package]
package com.b.tour;
public class AndamanPackage
{
        int basic_price=12000;
        public void ShowTariff()
        {
                System.out.println("***Tariff for Andaman tour in Company B***" );
                System.out.println("For two person , Andaman tour package is Rs. "+ basic_price*2 );
                System.out.println("For four person , Andaman tour package is Rs. "+ basic_price*4 );
                System.out.println("**************" );
        }
}
// GoaPackage.java [For Company B, in com.b.tour package]
package tour.company.b;
public class GoaPackage
{
        int basic_price=15000;
        int serviceTax=2000;
        public void ShowTariff()
```

```
        {
                int forTwoPerson=basic_price*2 +serviceTax;
                int forFourPerson=basic_price*4 +serviceTax;
                System.out.println("***Tariff for Goa tour in Company B***" );
                System.out.println("In Company A:For two person , Goa tour package is Rs. "+
                forTwoPerson);
                System.out.println("In Company A:For four person , Goa tour package is Rs.
                "+ forFourPerson );
                System.out.println("****************" );
        }
}
//Our main
package packages.examples;

//For company a packages
/*
import com.a.tour.GoaPackage;
import com.a.tour.KeralaPackage;
//or, simply use the following statement*/
import com.a.tour.*;
//For company b packages
import com.b.tour.*;

public class PackageEx {
        public static void main(String args[])
        {
                System.out.println("***Package Example Demo***");
                // Only CompanyA has KeralaPackage
                KeralaPackage keralaPackageInA=new KeralaPackage();
                keralaPackageInA.ShowPrice();
                //Only CompanyB has AndamanPackage
                AndamanPackage companyBAndamanPackage=new AndamanPackage();
                companyBAndamanPackage.ShowTariff();
                //Company A and B both have package for Goa.
                com.a.tour.GoaPackage companyAGoaPackage=new com.a.tour.GoaPackage();
                companyAGoaPackage.ShowPrice();

                com.b.tour.GoaPackage companyBGoaPackage=new com.b.tour.GoaPackage();
                companyBGoaPackage.ShowTariff();
        }
}
```

Output



- All classes in `java.lang` package are imported by default.
    1. Every class in Java resides inside a Package.
    2. If you want to rename your package, first rename the directory in which your classes are stored.
    3. Package naming convention should be followed carefully e.g. if we use statement like package a.b.c; we mean to say that directory c is inside b which is in inside a.
- You must remember the visibility control mechanism with the following table:

	public	protected	private	Default/No modifier
Same class	Yes	Yes	Yes	Yes
Subclass in same package	Yes	Yes	No	Yes
Non-subclass in same package	Yes	Yes	No	Yes
Subclass in different package	Yes	Yes	No	No
Non-subclass in different package	Yes	No	No	No

Students ask:

Sir, if every class stays in some package, then how could we use `System.out.print()` till now without importing any package?

*Teacher says:*Remember that in Java, all classes in `java.lang` package, are imported by default. This is why, you can use `System.out.println()` because `System` class also resides inside the default `java.lang` package.

Students ask:

Sir, can you please explain about the default access specifier?

Teacher says: always look into the table mentioned above. From the table, it is obvious that if you do not mention any specific access modifiers like public, private etc. to any of the member, it will be considered to have a default modifier and then your particular member is visible inside the same packages only i.e. all other classes inside the package can see and use them.

By the same way, you can give the visibility to outside classes with a restriction that only those outside classes that are in the same inheritance hierarchy (i.e. subclass) can see the intended member, then you can use the modifier protected. And if you do not want to put any restriction at all, simply use the public modifier and to provide maximum restriction, use the private modifier.

Students ask:

Sir, what is the purpose of `import` statements?

Teacher says: We bring all classes (or packages) from a specified location to our intended location with the use of `import` statements. Otherwise, we need to use the fully qualified name e.g. suppose we have class named `MyClass` with some methods (for simplicity, consider we have used public modifiers only). This class is stored inside a package (or directory) `b` which, in turn, is placed inside another directory a. Now we want to reuse those methods/class from a different location. So, ideally, we need to refer the class as `a.b.Myclass`. So, we can see that it becomes tedious and looks ugly to type the long dot separated package name first for the classes we need to use. So, in short, we can save a lot of typing and increase the readability of our program.

Students ask:

Sir, then technically we can avoid the `import` statements. Is the understanding correct?

Teacher says: Yes but you have to pay a lot in terms of typing and readability in a real life programming situation. So, particularly, I'll not encourage you to do that.

Students ask:

Sir, suppose we have used same class name inside two packages. And then in some other program, we have imported both of the packages. Now will we face any compiler issue? or, now how can we access any particular class?

Teacher says: First of all there is no compiler issue. If you have same named class in two or more packages, just use their fully qualified name to avoid the conflict. Go through the program that is presented earlier in the chapter. You have noticed that both of the packages have the class `GoaPackage` and we have used their fully qualified name inside our `main()`.

Students ask:

Sir, in some example, we see `import` statement is the first statement.But you also tell us the `package` statement is the first statement.Then if we have both, which one should come first?

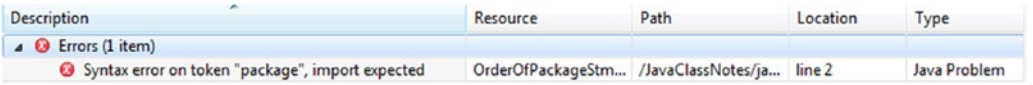
Teacher says: We must remember that `package` statements should be the first statements. Then `import` statements should be placed.Consider the below program:

Demonstration-2

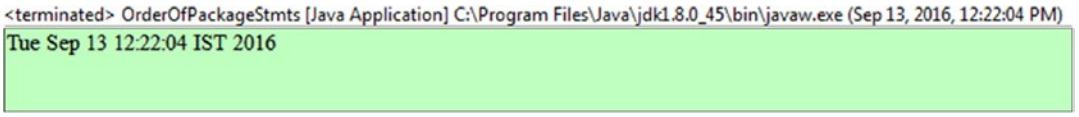
```
import java.util.Date;// error
package javaclassnotes.testprograms;
//import java.util.Date;
public class OrderOfPackageStmts {
        public static void main(String args[])
        {
                Date currentTime = new Date();
              System.out.println(currentTime.toString());
        }
}
```

Output

Compilation error.



Once you change the order of package and the `import` statement, you will get the expected output:



Students ask:

Sir, why we support this kind of design-package statements must come before import statements?
Teacher says: I see it from this angle-you need to fix a location first before writing the code. Then we may decide which classes are needed for our application .If our class is inside the same package (i.e. in same location), we can refer it immediately and import do not come into picture. Otherwise, we need to bring that class in our intended location and import takes its place. So, its like first deciding a place first to build a house. We can never build a house and then change the location. Likewise, if you notice carefully, you will find that package naming conventions follow a directory structure of the corresponding bytecode i.e. your intention is to fix a location first and then you proceed.

Students ask:

Sir, how to deal with multiple package statements in a source file?
Teacher says: You can have only one package statement in a source file.

Students ask:

Sir, in many cases, why we do not see any package statement at all in a source file?
Teacher says: It means that you are using the current default package.

Students ask:

Sir, till now we have got some idea about the usage of packages. It'll be helpful if you can summarize the overall usefulness of the packages.

Teacher says: If you notice carefully, you can see that packages are covering 3 major scenarios:

- They provide an organized structure which is very much useful to understand the program and debug.
- We are avoiding naming collisions using package statements.
- With different access modifiers inside packages, we can provide a level of security which is very much required in the real development process of a software.

CHAPTER 9

OOPs Concepts Revisited

The fundamental features of object oriented programming is as below:

- Class and object
- Polymorphism
- Abstraction
- Encapsulation
- Inheritance
- Message passing
- Dynamic binding

Teacher asks:

Can you tell me how we have covered these topics in Java?

- Class and object-Almost in every example, we have used classes and objects.
- Polymorphism -Method overloading (compile-time polymorphism) and method overriding (run-time polymorphism).
- Abstraction-Abstract classes and interfaces.
- Encapsulation-Each class can be an example. A more effective example can be a class with a private member and getter-setter.
- Inheritance-Examples in inheritance.
- Message passing- Mostly observed in a multithreaded environment. But also the dynamic method dispatch example can be treated in this category.
- Dynamic binding –Through the example in the dynamic method dispatch (method overriding).

Students ask:

Can you give us a simple example of getter-setter?

V. Sarcar, *Interactive Object Oriented Programming in Java*, DOI 10.1007/978-1-4842-2544-8_9

Teacher says: Go through the following program and analysis.

Demonstration-1

```
package oopsconcepts.examples;

class GetterSetter
{
        private int aVariable=5;
        public int getaVariable() {
                return aVariable;
        }
        public void setaVariable(int aVariable) {
                this.aVariable = aVariable;
        }

}
public class GetterSetterEx1
{
        public static void main(String args[])
        {
                System.out.println("*** Simple getter-setter example ***");
                GetterSetter myOb=new  GetterSetter();
                //Following line will cause error:
                //We cannot access the private member
                //System.out.println(myOb.aVariable);
                System.out.println("aVariable is now : "+myOb.getaVariable());//5
                //Setting the variable
                myOb.setaVariable(25);
                //Check the modification
                System.out.println("aVariable is now : "+myOb.getaVariable());//25
        }
}
```

Output

```
<terminated> GetterSetterEx1 [Java Application] C:\Program Files\Java\jdk1.8.0_45\bin\javaw.exe (Sep 9, 2016, 9:52:57 AM)
*** Simple getter-setter example ***
aVariable is now : 5
aVariable is now : 25
```

Analysis

We can see from the above example that we cannot access the private member of a class (as expected) through the object. But we can achieve this functionality through getter-setters which are basically two public methods inside the class and have access to this private member.

We must also note that we can have either a getter or a setter (or both- as in the example above).When we have a getter only, we can see the private member but we cannot change that private member .So the variable becomes read-only.

And in a similar fashion when we have only setter, the variable becomes write-only.

Students ask:

Sir, can you please summarize the difference between abstraction and encapsulation?

Teacher says: The process of binding/wrapping the data and codes into a single entity is termed as encapsulation. It helps us to prevent the arbitrary and unsecured accesses from outside the wrapper.

In an abstraction, we show the essential features and hides the detailed implementation from user e.g. when we use a remote control to switch on a television, we do not think about the internal circuits or digital instructions. We are simply happy if we can see the images coming out from the television after the button press.

Students ask:

What are the different types of polymorphism?

Teacher says:

- Compile time polymorphism-Which method needs to call is resolved by the compiler. So, it is also known as early binding. Since the call is resolved early, it is faster in general.
- Run time polymorphism (Or, Dynamic Polymorphism) –Which method needs to call will be decided during runtime. That is why, it is also known as late binding and it is slower compared to early binding.

Students ask:

Does Java support pointers like C/C++?

Teacher says: We already mentioned that Java does not support pointers. One of the main reason is with pointers we can access beyond our intended data boundary which is really dangerous. Apart from this, if we support pointers, memory management will become tedious, because, in many cases, they are error-prone. We believe, as long as we are in the Java execution environment, we'll never feel the need of using a pointer. Java developers also believe that the major use of pointers come into pictures due to the use of structures, processing of strings and manipulation of arrays. But Java does not use structures .And in Java, strings and arrays are treated as objects- so these entire domain of usage can be replaced without the pointers.

Students ask:

Sir, you told us earlier that inheritance may not provide the best solution always. Can you please elaborate it?

Teacher says: In some cases, composition can provide a better solution. But to understand the composition, you need to be aware of following concepts first:

- Association
- Aggregation

Association: It can be one way or 2-ways.Suppose we are seeing this kind of UML diagram:

It means `ClassA` knows about `ClassB` but reverse is not true.

And when we see this kind of diagrams, we conclude it as a 2-ways association i.e. they know each other.

Aggregation: It is a stronger type of association and widely represented as:

E.g. a professor belongs to a particular department. If in future, the department closes, the professor still can exist. But the reverse is also true i.e. if a professor leaves the college and he joins in a new institution, his old department still exists.

■ **Note** So, in general, we say that a department has a professor. For this reason, an association relationship is also termed as a **has-a** relationship. (Remember, in case of inheritance, we discussed about is-a relationships).

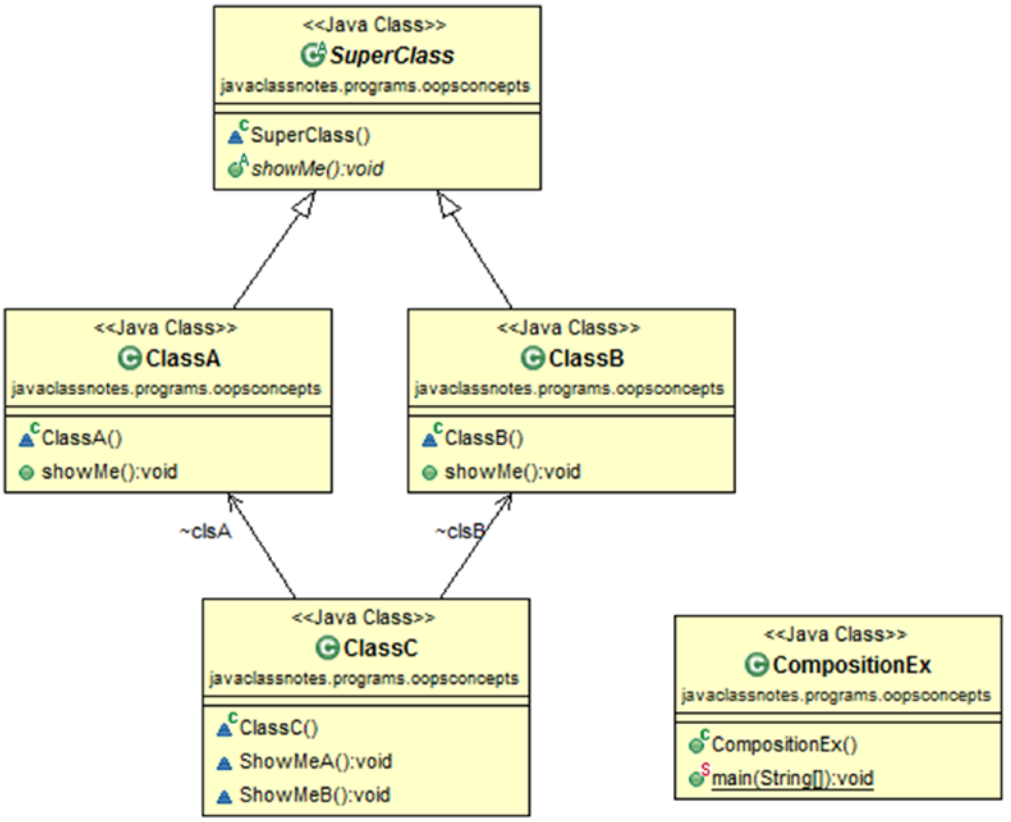
Composition: It is a stronger form of aggregation and this time we have a filled diamond in place.

A particular department in a college cannot exist without the college. We also note the fact that the college is only responsible to create and destroy (close) any of its department.(You can argue that if there is no department at all, a college also cannot exist, but we do not need to complicate the things by considering these type of corner cases to understand these concepts).

Now to show the power of composition, *recall the diamond problem mentioned in the chapter of inheritance.* And then go through the following program with the analysis:

Composition

Demonstration-2




```
package oopsconcepts.examples;

abstract class SuperClass
{
        public abstract  void showMe();
}
//Both ClassA and ClassB are overriding their parent method.
class ClassA extends SuperClass
{
        @Override
        public void showMe()
        {
                System.out.println("I am in Class A");
        }
}
class ClassB extends SuperClass
{
        @Override
        public void showMe()
```

```
        {
                System.out.println("I am in Class B");
        }
}
//The concept of composition to avoid the diamond problem
class ClassC
{
        ClassA clsA;
        ClassB clsB;
        ClassC()
        {
          this.clsA=new ClassA();
          this.clsB=new ClassB();
        }
        void ShowMeA()
        {
                clsA.showMe();
        }
        void ShowMeB()
        {
                clsB.showMe();
        }
}
public class CompositionEx {

        public static void main(String args[])
        {
                System.out.println("*** Example of composition to avoid multiple inheritance
                  in Java***");
                CompositionEx ob=new CompositionEx();
                ClassC clsC=new ClassC();
                clsC.ShowMeA();
                clsC.ShowMeB();
        }
}
```

Output

```
<terminated> CompositionEx [Java Application] C:\Program Files\Java\jdk1.8.0_45\bin\javaw.exe (Sep 25, 2016, 5:02:34 PM)
*** Example of composition to avoid multiple inheritance in Java***
I am in Class A
I am in Class B
```

You can see that both classes- `ClassA` and `ClassB` have overridden their parent method `showMe()`. And `ClassC` doesn't have its own `showMe()` method. But still we can call those class specific methods through an object of `ClassC`.

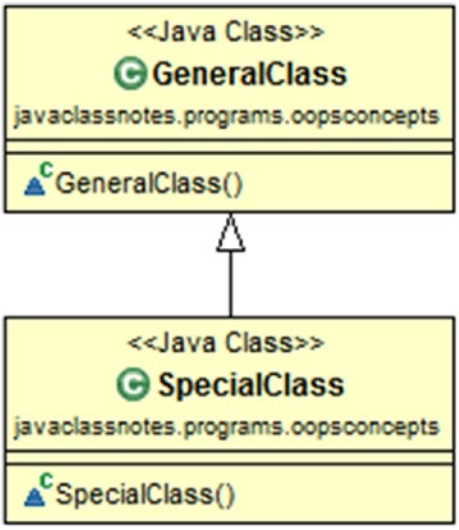
You must notice that `ClassC` is holding the references of both the classes-`ClassA` and `ClassB` and inside its constructor only, we have created `ClassA` and `ClassB` objects. So, when `ClassC` objects will not be present in our application (e.g. garbage collected), there will be no `ClassA` or `ClassB` objects inside the system. You can also put some restrictions to users so that they will not be able to create objects for `ClassA` and `ClassB` directly inside the application but for simplicity, we have ignored that part.

You must be aware of some other common terms e.g. generalization/specialization and realization. We have already used these concepts in our applications. Simply, in our programs, when our class extends another class (i.e. inheritance), we used the concepts of generalization/specialization e.g. a `Cricketer` is a special kind (specialization) of a `Player`. Or we can say that both the `Footballer` and `Cricketer` are `Players` (generalization).

And when our class implemented an interface, we used the concept of realization. For your ready reference, go through the following structures:

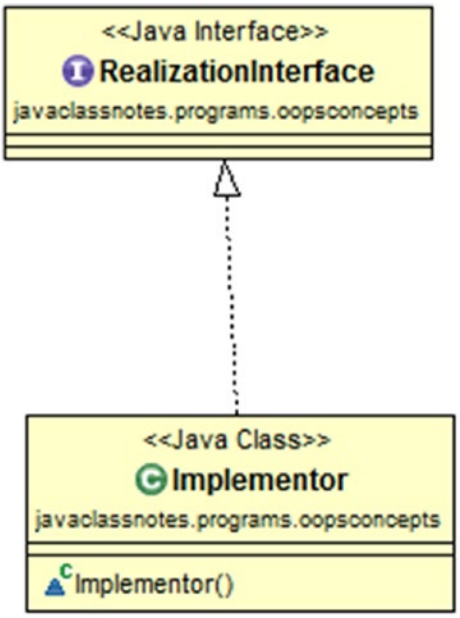
Generalization demo

```
package oopsconcepts.examples;
class GeneralClass
{
 //Some code
}
class SpecialClass extends GeneralClass
{
//Some code
}
```



Realization demo

```
package oopsconcepts.examples;

interface  RealizationInterface
{
 //Some code
}
class Implementor implements RealizationInterface
{
 //Some code
}
```



Apart from these there are concepts of reflexive associations and multiplicities (We will see them in the solution of assignment for the chapter "*Use of static keyword*").

Students ask:

What are challenges/drawbacks of OOP?

Teacher says: Some of the experts believe that in general, size of the object oriented programs are larger. As size of the programs are larger, we can assume that it will take more storage (But, I believe that in modern days, in most of cases, these issues hardly matters).

Some developers find challenges to design, code and debug in this type of programming style. As a result, maintenance of the software becomes tricky for them.

Some of the real world situations cannot be modeled properly with object oriented style. But at the end, I personally like OOPs because its merit side is much heavier than the demerit side.

CHAPTER 10

Use of static keyword

Sometimes we need variables that can be used without creating any object of that class. To serve that purpose, we tag the member/s with the keyword `static`. When a member is preceded with the keyword `static`, the member can be accessed before any object of that class is created i.e. we do not need to reference any object in this context.

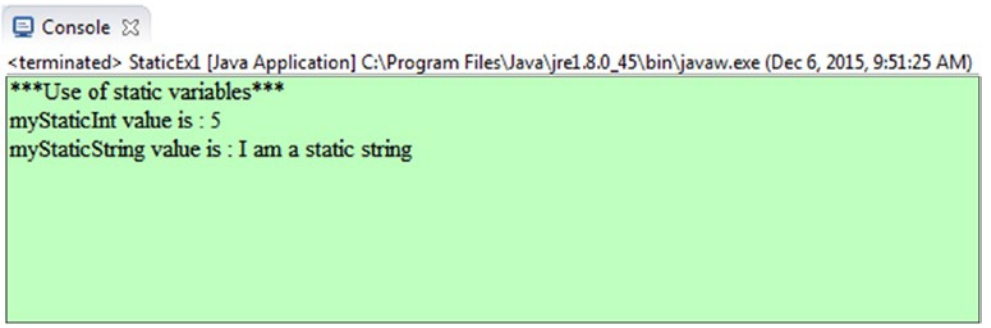
Consider the below example:

Demonstration-1

```
package useofstatickeyword.examples;

class StaticDemo1
{
        //static members
        static int myStaticInt=5;
        static String myStaticString="I am a static string";
        //Non static members
        int myNonStaticInt=25;
}
public class StaticEx1
{
        public static void main(String args[])
        {
                System.out.println("***Use of static variables***");
                //We can call static members with the class name itself
                //No need to create objects
                System.out.println("myStaticInt value is : "+StaticDemo1.myStaticInt);
                System.out.println("myStaticString value is : "+StaticDemo1.myStaticString);
                //Error:We cannot call instance variable with class name.
                //System.out.println("myNonStaticInt value is : "+StaticDemo1.myNonStaticInt);

        }
}
```


V. Sarcar, *Interactive Object Oriented Programming in Java*, DOI 10.1007/978-1-4842-2544-8_10

Output



Analysis

We can see that we are accessing the static members with `classname.membername` i.e. we do not need to create an object to access those variables.

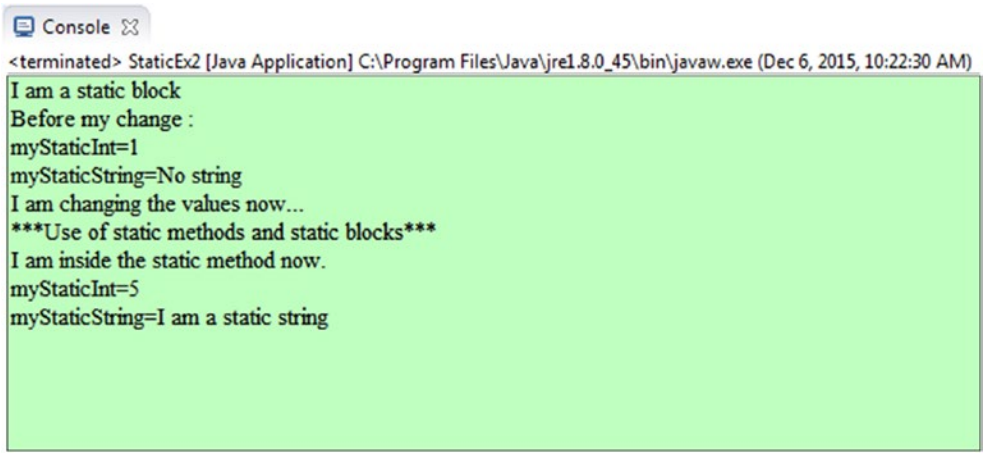
Demonstration-2

Let us go through another example. Here we'll test static variables initialization with a static method and a static block.

```
package useofstatickeyword.examples;

class StaticEx2
{
        //static members
        static int myStaticInt=1;
        static String myStaticString="No string";
             //instance variable
        int nonStaticInt=25;
        //static method
        static void setValuesToStaticMembers()
        {
                System.out.println("I am inside the static method now.");
                System.out.println("myStaticInt="+ myStaticInt);
                System.out.println("myStaticString="+ myStaticString);
                          //error:Can access only static fields from here
                //System.out.println("myNonStaticInt="+ myNonStaticInt);
        }
        //static block
        static
        {
                System.out.println("I am a static block");
                System.out.println("Before my change :");
                System.out.println("myStaticInt="+ myStaticInt);
                System.out.println("myStaticString="+ myStaticString);
```

```
                System.out.println("I am changing the values now...");
                myStaticInt=5;
                myStaticString="I am a static string";
        }
        public static void main(String args[])
        {
                System.out.println("***Use of static methods and static blocks***");
                StaticEx2.setValuesToStaticMembers();
        }
}
```

Output



Analysis

Look at the output carefully. Look at the order of the output. You can see the statements in the static block printed on the top of output. Even before the execution of the static block, the static variables were initialized. Later static block changed the values (which are reflected clearly when we call the static method).

It is because as soon as a static class loaded, all static statements run.

You should notice another important characteristic also: Static methods can access only static members. Also note that our static method is nested here.

Rules of thumb:

- Static methods can only other static methods.
- Static methods can access only static fields.
- Static methods cannot refer this or super.

Students ask:

Can we create static class?

Yes. But there is a constraint in Java. The static class should be inside of another class i.e. it must be nested. Java does not allow us to create top level static class.

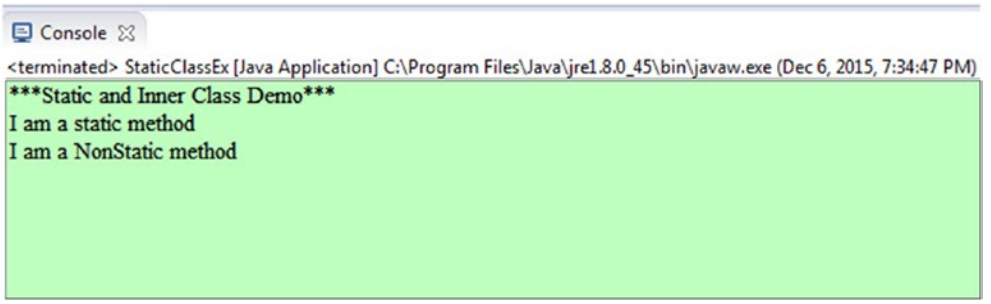
The class which contains the static class is termed as an outer class.

Consider the below example. Here we have shown how to create and use a nested static class and a nested non-static (inner class).

Demonstration-3

```
package useofstatickeyword.examples;

//Java doesn't allow us to create top-level static
//classes, it must be  nested.

class OuterClass
{
        //static class
        static class MyStaticClass
        {
                public static void showStaticMethod()
                {
                        System.out.println("I am a static method");
                }
        }
        //non static inner class
        public class MyNonStaticClass
        {
                public void showNonStaticMethod()
                {
                        System.out.println("I am a NonStatic method");
                }
        }
}

class StaticClassEx
{
        public static void main(String args[])
        {
                System.out.println("***Static and Inner Class Demo***");
                //Call Static method OuterClass.MyStaticClass.showStaticMethod();
                //CallNonStatic method
                OuterClass.MyNonStaticClass obNonStatic=(new OuterClass()).new MyNonStaticClass();
                obNonStatic.showNonStaticMethod();
        }
}
```

Output



Students ask:

What is an inner class?

As described above, a non-static nested class is termed as an inner class. It can access all the variables and methods of the outer class.

Students ask:

From static class, can we access the variables of outer class?

Answer is yes if and only if those variables are static. Consider the below example. Following code snippet is fine:

```
class OuterClass
{
        static int outer_int=25;
        //static class
        static class MyStaticClass
        {
                public static void showStaticMethod()
                {
                        System.out.println("I am a static method");
                        System.out.println("Outer_int =" +outer_int);
                }
        }
}
```

But if `outer_int` is non-static, compiler will raise an issue.

Teacher asks:

Can you predict the output?

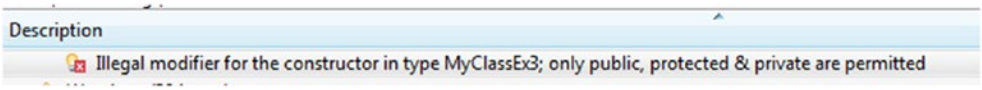
Quiz

```
package useofstatickeyword.examples;

class MyClassEx3
{
        //Constructors cannot be final/abstract/static
        static MyClassEx3()
```

```
        {
                System.out.println("I am a no argument constructor");
        }
}
class ExperimentWithConstructorEx3
{
        public static void main(String args[])
        {
                System.out.println("*** Experiment with constructors  ***");
                System.out.println("***Question:Can construcors be static?  ***");
                MyClassEx3 myOb=new  MyClassEx3();
        }
}
```

Output

Compilation error.



Students ask:

Sir, why constructors cannot be static?

Teacher says: Think from a general point of view. The keyword `static` is used to represent a method or variable as a class variable means it is not specific to any object. But constructors are used to initialize a particular object. And we must remember that to access class variables we use class name not objects. So, there is no point of making static constructors.

Now consider the below program and corresponding output carefully:

Demonstration-4

```
package useofstatickeyword.examples;

class StaticDemo3
{
        static void aStaticmethod()
        {
                System.out.println("I am a static method");
        }
        void aNonStaticmethod()
        {
                System.out.println("A non static method");
        }
}
```

```
class ChildStaticDemo3 extends StaticDemo3
{
        static void aStaticmethod()
        {
                System.out.println("I am a overriding the static method");
        }
        void aNonStaticmethod()
        {
                System.out.println("Overriding a non static method");
        }
}
public class StaticEx3
{
        public static void main(String args[])
        {
                System.out.println("***Testing the static methods***");
                StaticDemo3.aStaticmethod();
                ChildStaticDemo3.aStaticmethod();
                //Checking dynamic method dispatch
                StaticDemo3 parent=new ChildStaticDemo3();
                parent.aStaticmethod();
                parent.aNonStaticmethod();

        }
}
```

Output

```
<terminated> StaticEx3 [Java Application] C:\Program Files\Java\jdk1.8.0_45\bin\javaw.exe (Sep 17, 2016, 8:39:39 PM)
***Testing the static methods***
I am a static method
I am a overriding the static method
I am a static method
Overriding a non static method
```

Teacher asks:

Can you see any important behavior with the above program?
Students reply: Yes Sir. All results are expected except the result from the call `parent.aStaticmethod()`. We are seeing instead of calling the child class method, it is calling the static method from base class.
Teacher says: Yes. Here is the difference and we must remember that static method cannot be overridden.

Students ask:

Why static methods cannot be overridden?
Teacher says: We must know that for static methods (or class methods) , method calls are decided at compile time only i.e. it is not dependent on which object we are pointing at runtime. But for non static methods, method calls are decided at runtime (which object we are pointing at that moment).

Students ask:

If static methods cannot be overridden, then why we did not receive the compiler error in the above example?

Teacher says: There the derived class hides the static method in the parent class. If you add following lines inside the `main()` above:

```
//Will call the child method
ChildStaticDemo3 child=new ChildStaticDemo3();
child.aStaticmethod();
```

Then you will get the corresponding method calls from the child class:

```
<terminated> StaticEx3 [Java Application] C:\Program Files\Java\jdk1.8.0_45\bin\javaw.exe (Sep 26, 2016, 9:21:02 PM)
I am a static method
I am a overriding the static method
I am a static method
Overriding a non static method
I am a overriding the static method
```

Students ask:

Can we overload static methods?

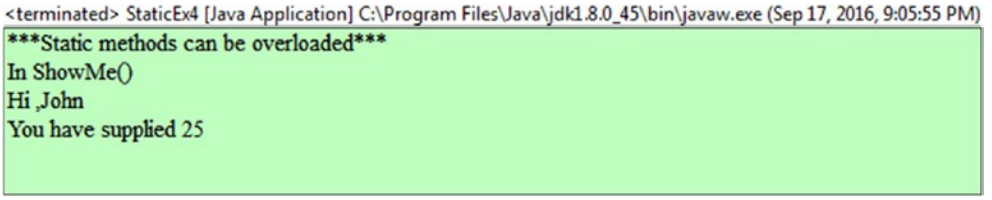
Teacher says: Yes. Consider the below program and output:

Demonstration-5

```
package useofstatickeyword.examples;

class StaticDemo4
{
        static void showMe()
        {
                System.out.println("In ShowMe()");
        }
        static void showMe(String s)
        {
                System.out.println("Hi ," +s);
        }
        static void showMe(int i)
        {
                System.out.println("You have supplied " +i);
        }
}
class StaticEx4
{
        public static void main(String[] args)
        {
                System.out.println("***Static methods can be overloaded***");
                StaticDemo4.showMe();
```

```
                StaticDemo4.showMe("John");
                StaticDemo4.showMe(25);
        }
}
```

Output



Teacher asks:

Will the code compile?

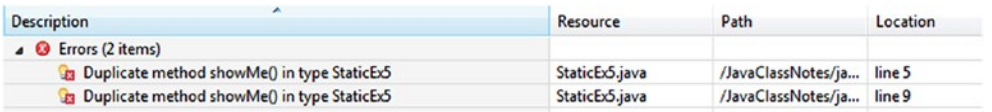
Quiz

```
package useofstatickeyword.examples;
class StaticEx5
{
        static void showMe()
        {
                System.out.println("Static method");
        }
        void showMe()
        {
                System.out.println("Non Static method");
        }
}
```

Answer: No. The overloading is to be allowed if the method signature is different. In the above case, inclusion of a static keyword before a method name is not considered as a different signature. (What are elements that constitute a method signature, we already discussed earlier)

In this case, compiler will raise the following error:

Output



Actually I see it from another point of view.In Java, we can invoke the static methods through objects also.Consider the below program and output.

Demonstration-6

```
package useofstatickeyword.examples;

class StaticDemo6
{
        static void showMe()
        {
                System.out.println("Static method");
        }
}

public class StaticEx6
{
        public static void main(String[] args)
        {
                System.out.println("***Static methods can be invoked through objects also in
                Java***");
                       //Using classname to call the static method
                StaticDemo6.showMe();
                //Using object to invoke the static method
                StaticDemo6 myOb=new StaticDemo6();
                myOb.showMe();
        }
}
```

Output

```
<terminated> StaticEx6 [Java Application] C:\Program Files\Java\jdk1.8.0_45\bin\javaw.exe (Sep 17, 2016, 9:39:55 PM)
***Static methods can be invoked through objects also in Java***
Static method
Static method
```

Analysis

So, we can see that Java allows us to call a static method through objects also. Now if we have another non static method with the same signature, Java compiler will be confused-which one to be called. That is another way of approving the fact that only the inclusion of the keyword `static` cannot be considered a overloaded method.

Students ask:

Then why Java allows us to invoke static methods through objects?

Teacher says: There is no straight forward answer to this question. But some developers find it useful when backward compatibility comes into picture. We can just use the modifier static to an older method and then we can invoke the method through the object.But at the same time, there are other languages who will not allow you to do this.

Teacher asks:

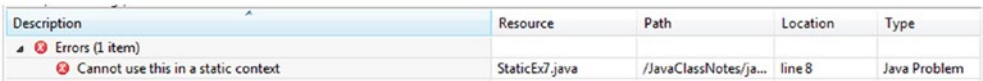
Can you explain the output of the below program?

Quiz

```
package javaclassnotes.programs;

public class StaticEx7
{
        int i;
        static void showMe()
        {
                this.i=7;
                System.out.println("Static method");
        }
}
```

Output

Compiler error.



Explanation

The keyword **`this`** is used in the context of the current object. But static methods can be called with class name (and it is the true intention of the keyword static) .There is no need to create an object to call a class method (or a static method).

Assignment

Suppose you have formed a cricket team. Now your team is going to play against an opponent team. You must be aware of the fact that which team will bat (or bowl) first will be decided through the toss and you need to send your captain for that. So, at first, you must elect a captain. At the same time, you must be aware that you can select one and only one captain. So, if you do not have any such captain, you will select one and send him for toss. Otherwise, you simply send the already nominated captain for the toss. Can you design this?

CHAPTER 11

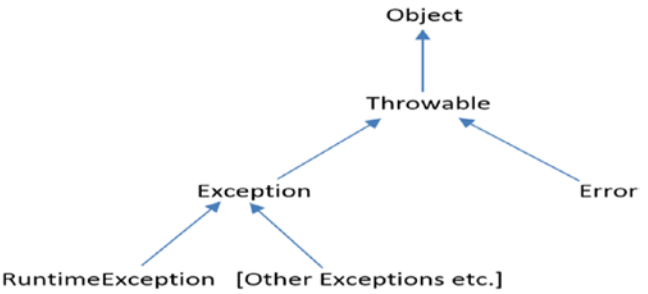
Exceptions

Here we deal with some unwanted situations. These situations can occur due to some careless mistakes or wrong logic written in the program. And we often term these unwanted situations as exceptions.

By definition, an exception is an event which breaks the normal execution/instruction flow. An object is created to describe the exception. When these exceptional situations arise, an exception object is created and thrown in the methods which created those. Then that method may or may not handle the situation (in programming term: the exception).If it cannot handle the exception, it will pass the responsibility to other. (Like our everyday situation, when situation reach beyond our control, we seek advice from experts).But ultimately they are handled and processed.

Java run time system can create these exceptions or we can create our own.

Before going forward, we'll introduce following class hierarchy to get some idea about our upcoming topic:



So, we can see that both `Exception` class and `Error` class are the subclasses of `Throwable` class which in turn derives from `Object` (in `java.lang` package).By other exceptions we mean classes like `IOException` (already defined in Java), our own custom exception classe/s (not defined in Java).

Here we'll primarily focus on `RuntimeExceptions`. Errors, in general, causes from some catastrophic failures like JVM out of memory, stack overflow etc. We can hardly do anything with them. Java run time environment itself needs to take care of these severe situations.

When we create our custom exception classes, in general, we'll subclass from the `Exception` class.

The following five keywords are used to deal with Java exceptions:

- `try`
- `catch`
- `throw`
- `throws`
- `finally`

V. Sarcar, *Interactive Object Oriented Programming in Java*, DOI 10.1007/978-1-4842-2544-8_11

Demonstration-1

Consider the below program:

I have taken snap shots of the code from eclipse editor here for some of the programs below -so that you can see the exact line number in the editor. This line numbers are useful when we analyze the outputs.

ExceptionEx1.java

```
1  package javaclassnotes.programs.exceptions;
2
3  public class ExceptionEx1 {
4     public static void main(String args[])
5     {
6        System.out.println("***Exception Example -1***");
7        int a=5;
8        int b=0;
9        int c=a/b;
10       System.out.println("I am at line number 10 now");
11       System.out.println("c="+c);
12    }
13 }
14
```

Output

```
<terminated> ExceptionEx1 [Java Application] C:\Program Files\Java\jdk1.8.0_45\bin\javaw.exe (Nov 27, 2016, 11:32:38 AM)
***Exception Example -1***Exception in thread "main"
java.lang.ArithmeticException: / by zero
        at javaclassnotes.programs.exceptions.ExceptionEx1.main(ExceptionEx1.java:9)
```

From the output window, we can have below observations:

- We have encountered an exception in the program.
- This exceptional situation arises in the `main` method of `ExceptionEx1.java`
- Line number of this faulty code is 9.

- / by zero (divide by zero) was the operation which raises this exception. The name of the exception is `AirthmeticException`. We have not defined this. That means, Java already defined this type of exception for us and this exception is defined in `java.lang` package(which is the default package).
- Once the program encountered this exception at line number 9, control comes out and it did not print the below statements (e.g I am in line number 10 now).

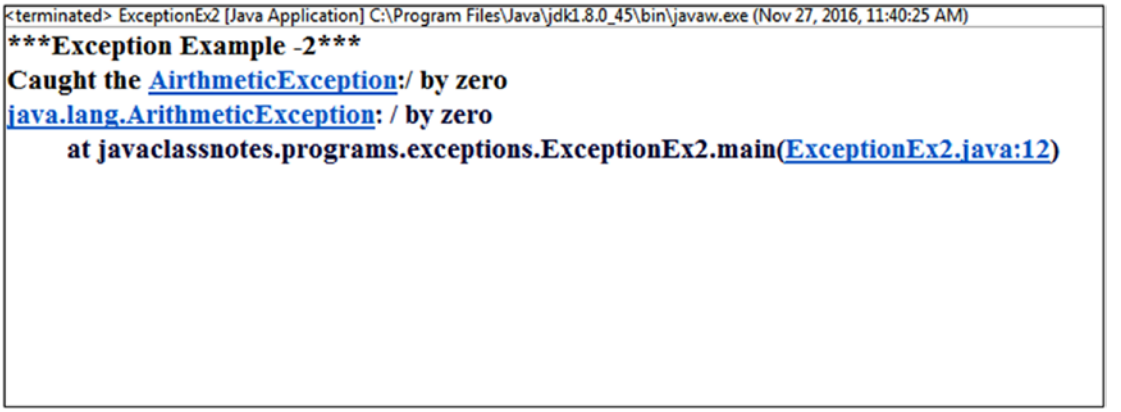
Demonstration-2

Now consider the below example. Now we try to catch the exception and handle it properly. The statement/s which may cause an exception are placed inside a `try` block. A `catch` block is placed just after the `try` block. If you notice carefully, you will find that this `catch` block will handle the `ArithmeticException` (if it occurs) only, no other type of exception cannot be handled by this `catch` block.

ExceptionEx2.java

```
1  package javaclassnotes.programs.exceptions;
2
3  public class ExceptionEx2{
4    public static void main(String args[])
5    {
6       System.out.println("***Exception Example -2***");
7       int a=5;
8       int b=0;
9       int c=0;
10      try
11      {
12        c=a/b;
13      System.out.println("c="+c);
14      }
15      catch(ArithmeticException ex)
16      {
17         System.out.println("Caught the AirthmeticException :"+ ex.getMessage());
18          ex.printStackTrace();
19      }
20   }
21 }
```

Output



Students ask:

Sir, what should be the behavior/output if we encounter a different type of exception in our code which we do not handle inside the catch block?

Teacher says: It will be handled by Java's run time error management (with a default handler) just like Demonstration 1.Consider the below example:

Demonstration-2A

ExceptionEx2A.java

```
1  package javaclassnotes.programs.exceptions;
2  public class ExceptionEx2A {
3     public static void main(String args[])
4     {
5        System.out.println("***Exception Example -2A***");
6        try
7        {
8           int a=5;
9           int b=0;
10          //AirthmeticException is Occured but we are not handling it by catching it
11          int c=a/b;
12          System.out.println("I am at line number 10 now");
13          System.out.println("c="+c);
14       }
15       catch(ArrayIndexOutOfBoundsException ex)
16       {
17          System.out.println("Caught the ArrayIndexOutOfBoundsException :"+ ex.getMessage());
18          ex.printStackTrace();
19       }
20    }
21 }
```

Output

We can see from the above example that if we do not catch the `ArrayIndexOutOfBoundException`, a default handler will take the charge and it will handle the scenario.

Students ask:

Sir, we are seeing that when we encounter an exception inside a method (here inside `main`) remaining lines are not printed in the console. But we want those lines to be printed. e.g. we want to see the statement I am at line number….Is there any way?

Teacher says: You can place the required lines in the `finally` block. Any code inside a `finally` block must be executed.

Demonstration-2B

*Exception2B.java

```
1  package javaclassnotes.programs.exceptions;
2  public class Exception2B {
3     public static void main(String args[])
4     {
5        System.out.println("***Exception Example -2B***");
6        try
7        {
8           int a=5;
9           int b=0;
10          //AirthmeticException is Occured but we are not handling it by catching it
11          int c=a/b;
12          System.out.println("I am at line number 10 now");
13          System.out.println("c="+c);
14       }
15       //We are not catching the AirthmeticException
16       catch(ArrayIndexOutOfBoundsException ex)
17       {
18          System.out.println("Caught the ArrayIndexOutOfBoundsException :"+ ex.getMessage());
19          ex.printStackTrace();
20       }
21       //finally block- always be executed.
22       finally
23       {
24          System.out.println("I am in finally.So I'll be printed always.");
25          System.out.println("I am at line number 25 now");
26       }
27    }
28 }
```

Output

```
<terminated> ExceptionEx2B [Java Application] C:\Program Files\Java\jdk1.8.0_45\bin\javaw.exe (Nov 27, 2016, 12:03:22 PM)
***Exception Example -2B***
Exception in thread "main" I am in finally.So I'll be printed always.
I am at line number 25 now
java.lang.ArithmeticException: / by zero
        at javaclassnotes.programs.exceptions.ExceptionEx2B.main(ExceptionEx2B.java:11)
```

Students ask:

Sir, then why we need `catch` block at all?
Teacher says: To handle the exception in some specified manner.

Students ask:

Sir, Can we use only `try` and `finally` like below?

```
try
{
   //Some code
 }
finally
{
   //Some code
}
```

Teacher says: Yes.

Students ask:

Sir, what will happen if we encounter exception inside a `finally` block?

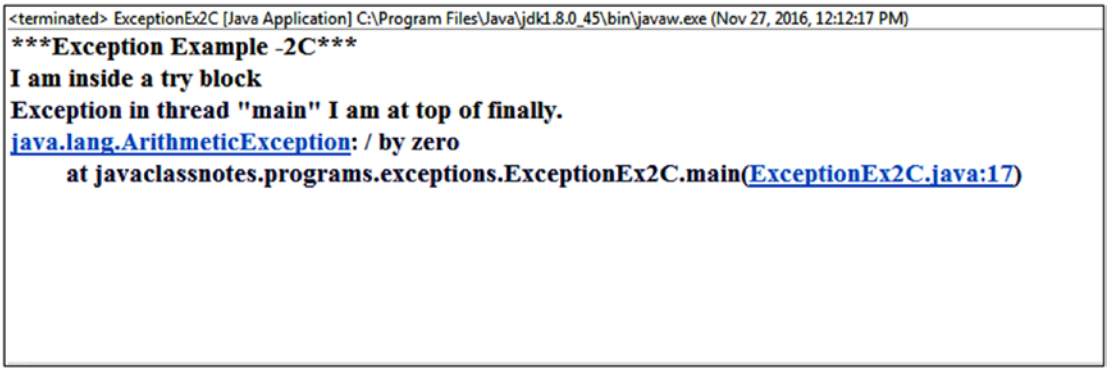
Teacher says: Good question. We should not forget the purpose of `finally`. The purpose of `finally` basically is to close files, release occupied resources etc. If we do not put those codes inside `finally`, there may be situations where we encounter an exception and an opened file is not closed properly or some resources are not released properly (which in turn can cause memory leaks inside the system). But yes, if you put your erroneous logic in the following manner, you'll get corresponding output like this (i.e. once we receive exceptions, below lines will not be executed in the block).

Demonstration-2C

*ExceptionEx2C.java

```
1  package javaclassnotes.programs.exceptions;
2  public class ExceptionEx2C {
3     public static void main(String args[])
4     {
5        System.out.println("***Exception Example -2C***");
6        try
7        {
8           System.out.println("I am inside a try block");
9        }
10       //Wrong way of writing :Exception occurs inside finally
11       finally
12       {
13          System.out.println("I am at top of finally.");
14          int a=5;
15          int b=0;
16          //AirthmeticException is Occured but we are not handling it by catching it
17          int c=a/b;
18          //Following lines will not be printed
19          System.out.println("c="+c);
20          System.out.println("I am at the bottom of finally");
21       }
22    }
23 }
```

Output



Demonstration-3

Now consider the below program. Here we have multiple `catch` blocks with a `try` block. In the try block we may encounter `ArithmeticException`, `ArrayIndexOutOfBoundsException` or `NullPointerException`. We are generating a random number between 0 and 2 and based on the random number generated, we'll encounter different type of exceptions and then handle those e.g. when `b=0`, we'll encounter `ArithmeticException`, if not then we'll proceed further. Now we'll check whether b is an even number or an odd number. If b is a non-zero even number, we encounter an `ArrayIndexOutOfBoundsException`. If b is a non-zero odd number, we'll encounter a `NullPointerException`.

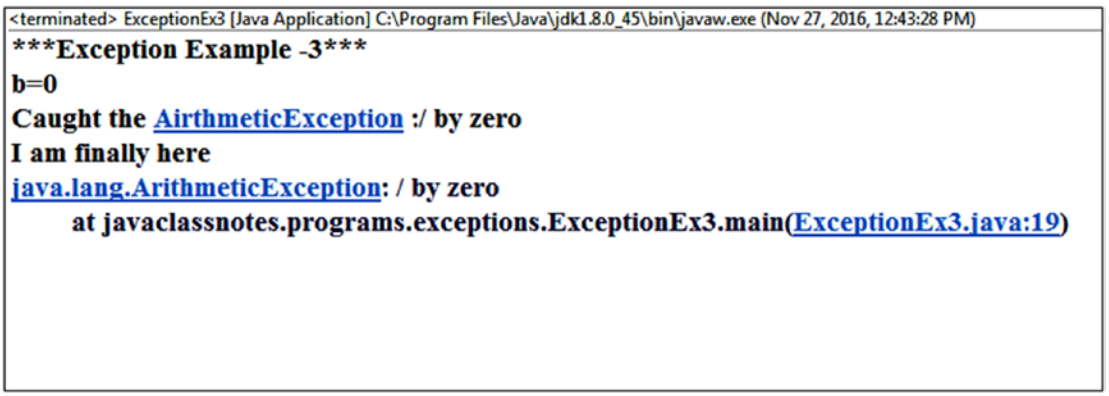
I have shown all possible outputs in different runs. You may get a different order because the value of b is generated at random.

```
package javaclassnotes.programs.exceptions;

import java.util.Random;

public class ExceptionEx3 {

        public static void main(String args[])
        {
                System.out.println("***Exception Example -3***");
                int a=5;
                Random randomGenerator=new Random();
                //Will generate 0 to 2.
                int b=randomGenerator.nextInt(3);
                System.out.println("b="+b);
                int c=0;
                try
                {
                //Case-1:it will encounter ArithmeticException
                //if b=0
                  c=a/b;
                System.out.println("c="+c);
                int[] arr=new int[2];
                arr[0]=0;
                arr[1]=c+1;
```

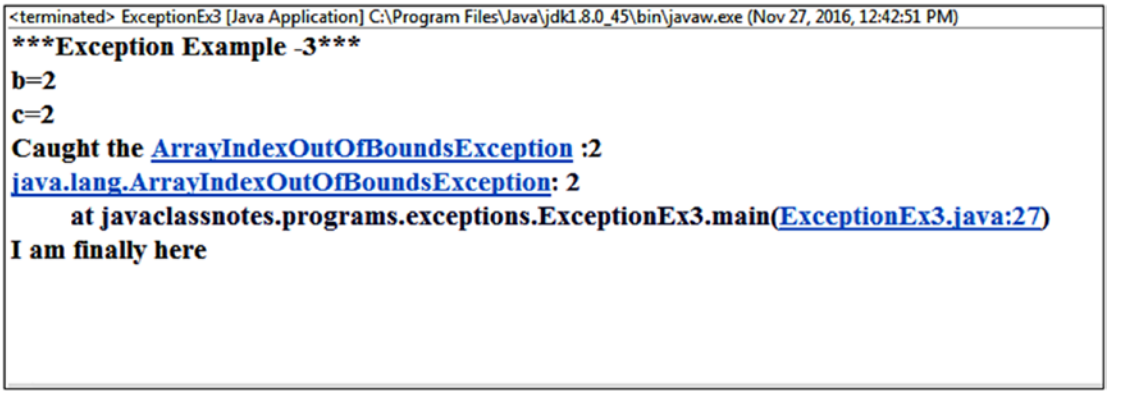
```
                if(b%2==0)
                {
                //Case-2: it will encounter //ArrayIndexOutOfBoundsException
                arr[2]=c+2;
                }
                else
                 {
                        Object myObject=null;
                        //case-3: It will encounter //NullPointerException
                        int hashcode=myObject.hashCode();
                 }
                }
                catch(ArithmeticException ex)
                {
                        System.out.println("Caught the AirthmeticException :"+ ex.getMessage());
                         ex.printStackTrace();
                }
                catch(ArrayIndexOutOfBoundsException ex)
                {
                        System.out.println("Caught the ArrayIndexOutOfBoundsException :"+
                          ex.getMessage());
                         ex.printStackTrace();
                }
                catch(Exception ex)
                {
                        System.out.println("Caught the Exception :"+ ex.getMessage());
                         ex.printStackTrace();
                }
                finally{
                System.out.println("I am finally here");
        }
        }
}
```

Output

Run 1:



Run 2:

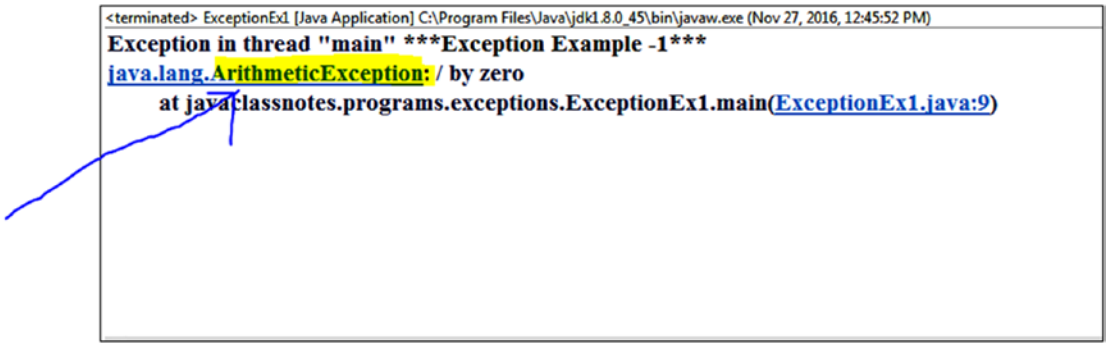


Run 3:

Students ask:

Sir, so far you have given examples like `ArrayIndexOutOfBoundsException`, `ArithmeticException` etc. How we'll remember these names?

Teacher says: These are built-in exceptions in Java. All of these are already defined in `java.lang` packages. Since this package is the default package, we'll get all these exceptions imported by default. Upon practice, you can remember their names. I personally take help from eclipse editor. Any IDE can help you in this context. Otherwise, you can see what the exception is-your default handler is throwing. From that report, you can get the name of the exception e.g. notice the output of our example 1:



From the output, you can see that the name of the exception name is AirthmeticException.

Demonstration-4

Consider the below program and corresponding output. So far, we were in receiving end, we were handling exceptions that were thrown by java run time system. But we have the freedom to throw an exception. This freedom is necessary when we make our own application and we want to control some situation by ourselves only.

The basic format is:

```
throw anObjectOfThrowable
```

where `anObjectOfThrowable` must be an instance of `Throwable` class or its subclass.

*ExceptionEx4.java

```
1  package javaclassnotes.programs.exceptions;
2
3  class DemoClass
4  {
5     void thowingException()
6     {
7        System.out.println("I always throw a NullPointerException");
8        throw new NullPointerException("Forceful throw");
9        //System.out.println("I will never print this line");
10    }
11 }
12 public class ExceptionEx4
13 {
14    public static void main(String args[])
15    {
16       System.out.println("***Exception Example -4: use of throw***\n");
17       DemoClass demo=new DemoClass();
18       try
19       {
20        demo.thowingException();
21       }
22       catch(Exception e)
23       {
24          System.out.println(e.getMessage());//Forceful throw
25          e.printStackTrace();
26       }
27    }
```

Output

```
<terminated> ExceptionEx4 [Java Application] C:\Program Files\Java\jdk1.8.0_45\bin\javaw.exe (Nov 27, 2016, 1:15:11 PM)
***Exception Example -4:use of throw***

I always throw a NullPointerException
Forceful throw
java.lang.NullPointerException: Forceful throw
        at javaclassnotes.programs.exceptions.DemoClass.thowingException(ExceptionEx4.java:8)
        at javaclassnotes.programs.exceptions.ExceptionEx4.main(ExceptionEx4.java:20)
```

Quiz

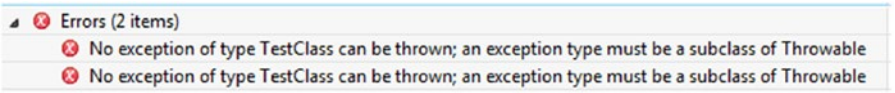
Teacher asks:

Tell me whether following block of code is valid or not?

```
class TestClass
{
   //some code
}
class DemoClass
{
        void thowingException()
        {
                System.out.println("I always throw an Exception");

                try
                {
                        throw new TestClass();
                }
                catch (TestClass e)
                {
                // TODO Auto-generated catch block

                }
        }
}
```

Answer: No. `TestClass` is not derived from `Throwable` or its subclass.

Output



Students ask:

How can we overcome this error?
Teacher says: By making it a subclass of `Throwable` (or its subclass) like below:

```
class TestClass extends Throwable
{
        //some code
}
class DemoClass
{
        void thowingException()
        {
                System.out.println("I always throw an Exception");
                try
```

```
                {
                        throw new TestClass();
                }
                catch (TestClass e)
                {
                 // TODO Auto-generated catch block

                }
        }
}
```

Students ask:

What are the different ways –by which we can obtain a throwable object/instance?
Teacher says:

- A common way is already shown in the above example- using a new operator like above.
- Another way is to use a parameter inside a catch clause.

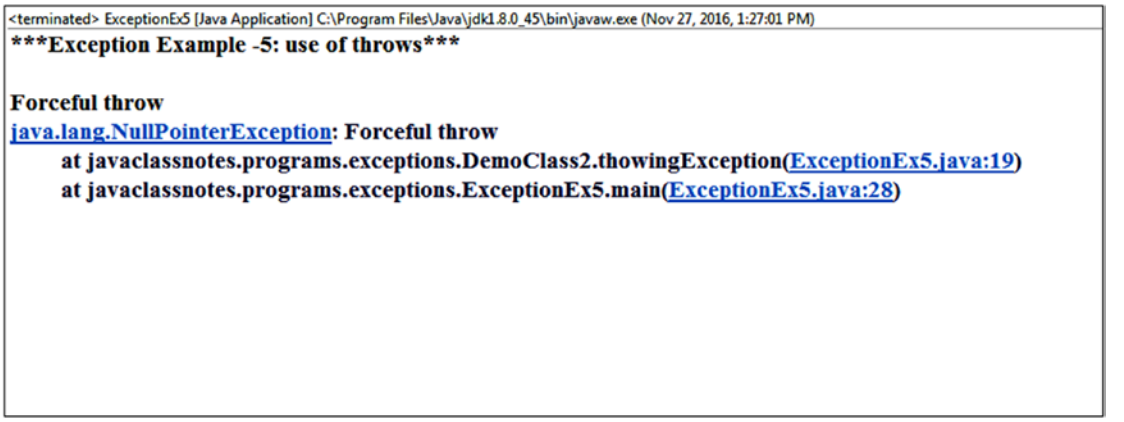
Demonstration-5

Consider the below program and corresponding output. Compare this program from the previous one (Demonstration 4) carefully. Notice at the highlighted areas in the below program- we have used the keyword `throws`. You can see that inside `DemoClass2`, the method `throwingException()` is throwing an exception but we have not used `try/catch` block around this code. Instead of this, we have added throws statements after the method names. It means that this method has the capability of throwing `TestClass2` exception. We have used a constructor (which can accept a `String` message) in the `TestClass2`, to provide a meaningful message when we intend to throw such kind of exceptions.

ExceptionEx5.java

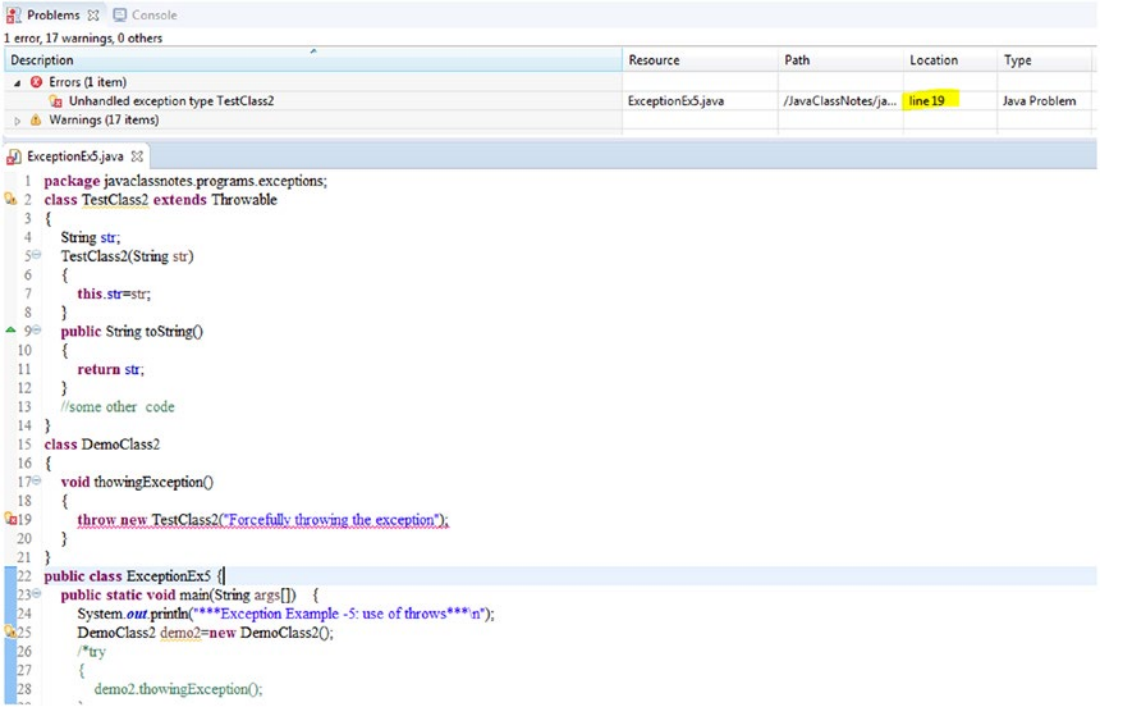
```
1  package javaclassnotes.programs.exceptions;
2  class TestClass2 extends Throwable
3  {
4     String str;
5     TestClass2(String str)
6     {
7        this.str=str;
8     }
9     public String toString()
10    {
11       return str;
12    }
13    //some other  code
14 }
15 class DemoClass2
16 {
17    void thowingException() throws TestClass2
18    {
19       throw new TestClass2("Forcefully throwing the exception");
20    }
21 }
22 public class ExceptionEx5 {
23    public static void main(String args[]) throws TestClass2  {
24       System.out.println("***Exception Example -5: use of throws***\n");
25       DemoClass2 demo2=new DemoClass2();
26       try
27       {
28          demo2.thowingException();
29       }
30       catch(TestClass2 e)
31       {
32          System.out.println(e);
33          e.printStackTrace();
34       }
35    }
36 }
```

Output



Points to remember:

- A `throws` clause will be needed to indicate all the exceptions that a method can throw. Otherwise, we'll encounter compile-time errors (except for the next point) e.g. in the above example, if we do not include throws clause with `throwingException()` (line no 17), we'll get the compile time error:



- The above rule will not be applicable for `Error` or `RuntimeException` or any of their subclasses.
- We must remember that checked exceptions must be included in a method's throws list.

Students ask:

You have just mentioned a term "Checked exception". What are the checked and unchecked exceptions?

Teacher says: Good question. We basically deal with two types of exceptions- checked and unchecked.

Checked exceptions-They can raise issue at compile time itself. That's why they are also known as compile time exceptions. We must need to take care to handle these type of exceptions.

To understand the difference between checked exceptions and unchecked exceptions, you must go through this example and the above example repeatedly.

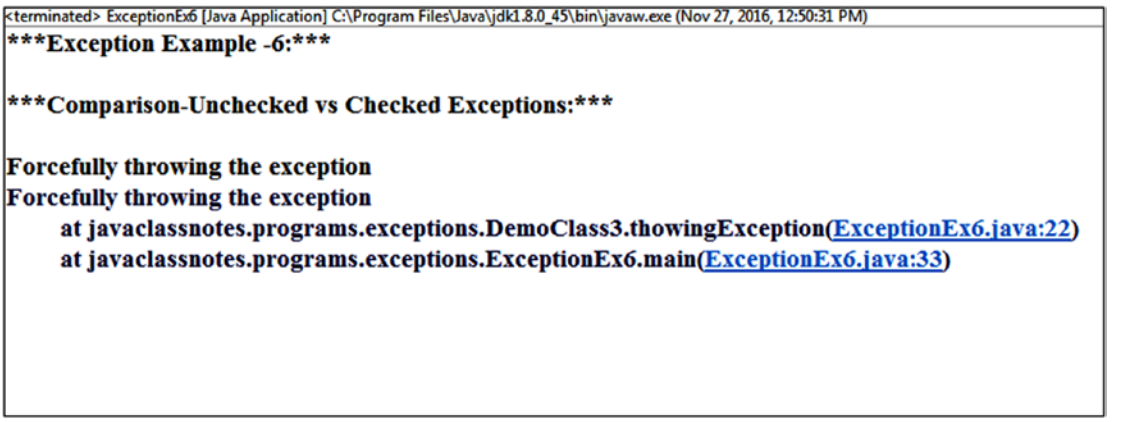
Demonstration-6

In the below example, our `TestClass3` inherits from `RuntimeException`. And now Java compiler will not check whether the method throws an exception or handles the exception. So, it becomes unchecked exception. So, no compile time error will be raised due to exclusion of throws clause in method declarations. But on the other hand, in our previous example (`ExceptionEx5.java`), our `TestClass2` inherits from `Throwable`, so we need to exclusively put the `throws` clause/s with method declarations.

```
package javaclassnotes.programs.exceptions;
//Check the difference with ExceptionEx5

class TestClass3 extends RuntimeException
{
        String str=null;
        TestClass3(String str)
        {
                this.str=str;
        }
        public String toString()
        {
                return str;
        }
        //some other  code
}
class DemoClass3
{
        void thowingException() //throws clause not necessary now
        {
                throw new TestClass3("Forcefully throwing the exception");
        }
}
public class ExceptionEx6 {
        public static void main(String args[])
        {
                System.out.println("***Exception Example -6:***\n");
                System.out.println("***Comparison-Unchecked vs Checked Exceptions:***\n");
                DemoClass3 demo3=new DemoClass3();
```

```
                try
                {
                        demo3.thowingException();
                }
                catch(TestClass3 e)
                {
                        System.out.println(e);
                        e.printStackTrace();
                }
        }
}
```

Output



Students ask:

Tell us about some built in checked and unchecked exceptions.

Checked exceptions examples:

- `ClassNotFoundException`
- `NoSuchMethodException`
- `NoSuchFieldException`
- `InstantiationException`
- `CloneNotSupportedException`
- `IllegalAccessException`

Unchecked exceptions examples:

- `ArithmeticException`
- `ArrayIndexOutOfBoundsException`
- `IndexOutOfBoundsException`
- `SecurityException`
- `NullPointerException etc.`

Discussion on Chained Exception

Sometimes we can receive an exception which may be caused by some other exception. So, we may be interested to know the original cause. The concept of chained exception comes into picture in such a scenario.

Consider a very simple scenario of `ArithmeticException` when we try to divide an integer by 0 (we showed the case in our demonstration).Though we are receiving this exception, the original cause is an I/O error which created this zero.

Chained exceptions can help us to know about such exceptions and also in which layer those exist.

Demonstration-7

Consider the below example and output:

```
package javaclassnotes.programs.exceptions;

class OuterException extends RuntimeException
{
        String str=null;
        OuterException(String str)
        {
                this.str=str;
        }
        public String toString()
        {
                return str;
        }
}
class InnerException extends RuntimeException
{
        String str=null;
        InnerException(String str)
        {
                this.str=str;
        }
        public String toString()
        {
                return str;
        }
}

class DemoClass4
{
        void thowingException() //throws clause not necessary now
        {
                OuterException outer=new OuterException("OuterException");
                InnerException inner=new InnerException("InnerException");
                outer.initCause(inner);
                throw outer;
        }
}
```

```
public class ExceptionEx7 {
        public static void main(String args[])
        {
                System.out.println("***Exception Example -7:***\n");
                System.out.println("***Demo:Chained Exception:***\n");
                DemoClass4 demo4=new DemoClass4();
                try
                {
                        demo4.thowingException();
                }
                catch(OuterException e)
                {
                        System.out.println("Caught : "+ e);
                        System.out.println("It is caused by :"+e.getCause());
                        //e.printStackTrace();
                }
        }
```

Output

Students ask:
Sir, in the above example, can the inner exception be further caused by another exception?

Yes. We can carry out to the depth we want. But it is always recommended that we must not end up by making a very long chain because that can lead to a poor design.

Demonstration-8

Consider the below example: we have modified the above program (Demonstration 7) to increase the depth of the exception by one level. `printStackTrace()` method is used to back trace and see the entire stack.

To show the exact line number in the output window, Eclipse snapshots are included in two parts here.

```
1  package javaclassnotes.programs.exceptions;
2
3  class OuterException extends RuntimeException
4  {
5     String str=null;
6     OuterException(String str)
7     {
8        this.str=str;
9     }
10    public String toString()
11    {
12       return str;
13    }
14 }
15 class InnerException extends RuntimeException
16 {
17    String str=null;
18    InnerException(String str)
19    {
20       this.str=str;
21    }
22    public String toString()
23    {
24       return str;
25    }
26 }
27 class SubInnerException extends RuntimeException
28 {
29    String str=null;
30    SubInnerException(String str)
31    {
32       this.str=str;
33    }
34    public String toString()
35    {
36       return str;
37    }
38 }
```

```
39 class DemoClass4
40 {
41   void thowingException() //throws clause not necessary now
42   {
43      OuterException outer=new OuterException("OuterException");
44      InnerException inner=new InnerException("InnerException");
45      SubInnerException subInner=new SubInnerException("SubInnerException");
46      outer.initCause(inner);
47      inner.initCause(subInner);
48      throw outer;
49   }
50 }
51 public class ExceptionEx7 {
52   public static void main(String args[])
53   {
54      System.out.println("***Exception Example -7.Modified:***\n");
55      System.out.println("***Demo:Chained Exception:***\n");
56      DemoClass4 demo4=new DemoClass4();
57      try
58      {
59         demo4.thowingException();
60      }
61      catch(OuterException e)
62      {
63         System.out.println("Caught : "+ e);
64         System.out.println("It is caused by :"+e.getCause());
65         e.printStackTrace();
66      }
67   }
68 }
```

Output

```
<terminated> ExceptionEx7 [Java Application] C:\Program Files\Java\jdk1.8.0_45\bin\javaw.exe (Nov 27, 2016, 1:00:28 PM)
***Exception Example -7.Modified***

***Demo:Chained Exception:***

Caught : OuterException
It is caused by :InnerException
OuterException
        at javaclassnotes.programs.exceptions.DemoClass4.thowingException(ExceptionEx7.java:43)
        at javaclassnotes.programs.exceptions.ExceptionEx7.main(ExceptionEx7.java:59)
Caused by: InnerException
        at javaclassnotes.programs.exceptions.DemoClass4.thowingException(ExceptionEx7.java:44)
        ... 1 more
Caused by: SubInnerException
        at javaclassnotes.programs.exceptions.DemoClass4.thowingException(ExceptionEx7.java:45)
        ... 1 more
```

To allow chained exceptions we have the following methods:

- `Throwable getCause() and`
- `Throwable initCause(Throwable cause)`

And the following constructors:

- `Throwable(Throwable cause)`
- `Throwable(String msg, Throwable cause)`

Students ask:

Sir, it appears to us that we can also suppress errors with exceptions.

Teacher says: Yes. But it is never intended. Consider the below example. Here, when we get 0 (randomly generated) inside b, instead of reporting the true issue, we can suppress the error by printing `c=7`-which is a total misuse of this feature.

```
package javaclassnotes.programs.exceptions;
import java.util.Random;

public class ExceptionEx8{
        public static void main(String args[])
        {
                System.out.println("***Exception Example -8***");
                System.out.println("***Wrong use of the concepts of Exception ***");
                int a=10;
                Random randomGenerator=new Random();
                //Will generate 0 to 2.
                int b=randomGenerator.nextInt(3);
                int c=0;
                try
                {
                  c=a/b;
                System.out.println("c="+c);
                }
                catch(ArithmeticException ex)
                {
                        //printing c=7, after catching the exception
                        System.out.println("b=" +b);
                        System.out.println("c=" +7);
                }
        }
}
```

Output

Students ask:

Sir, Java does not support pointers but it supports `NullPointerException`. Why?

Teacher's reply: You need to understand the scenario-When we try to do some illegal operations like invoking a method or try to access fields etc from a null object, we encounter this exception. This exception generally indicates that you are treating a null object as an actual object –so your intended operation is illegal.Yes, some developers believe that something like `NullReferenceException` could be a better naming for this type of exceptions.

But we must remember the fact that Java developers always had the believe that use of pointers by programmers are one of the primary sources of injecting bugs into the application. So they do not support any pointer datatype.

Assignment

Create a custom `Exception` class. You need to consider two integer inputs which must be supplied by the user. You will display the sum of the integers if and only if the sum is less than 100. If it is not less than 100, throw your custom exception.

CHAPTER 12

An introduction to design patterns

During different phases of a software development, one the most common query was: Is there any standards to this development process? The question was obvious because a software team consists of many engineers and they all involve in the development process. But different people have different mindsets and different level of understanding to deal with a similar kind of situation. This issue was also a big concern for a new member (experienced or unexperienced does not matter) who later joined in the team and was assigned to do something from scratch or to modify something in the existing product. As already mentioned, since earlier days, there were no standards, to become familiar with the existing design of the system, he/she needed to put some additional efforts though he/she may have handled situation like this earlier. Design Patterns simply addresses this kind of issues and make a common platform for all developers. We shall remember that these patterns were intended to be applied in object oriented designs with the intention of reuse to reduce duplicate efforts.

In 1994-95, four authors Erich Gamma, Richard Helm, Ralph Johnson and John Vlissides came with their famous book titled Design Patterns - Elements of Reusable Object-Oriented Software in which they initiated the concept of Design Pattern in Software development (*Design Patterns: Elements of Reusable Object-Oriented Software* (Addison-Wesley Pub Co, 1995)). These authors became popular with the name Gang of Four (GoF). They introduced 23 patterns which were developed by experienced software developers over a very long period of time. As a result, now if any new member joins in a development team and he knows that the new system is following some specific design patterns, immediately he can get some idea with that design and as a result, he can actively participate in the development process with the other members of the team within a very short period of time.

The building architect Christopher Alexander can be considered the father of these concepts. He found that throughout his life time, he encountered some common problems. He mastered himself to deal with those problems and tried to resolve those in some unified manner. People believe that software industry grasped those concepts because software engineers can also relate their product applications with these building applications.

Each pattern describes a problem which occurs over and over again in our environment, and then describes the core of the solution to that problem, in such a way that you can use this solution a million times over, without ever doing it the same way twice.

—Christopher Alexander

GoF assures us that though those patterns were described for building and towns, the same concepts can be applied with object oriented design methodology also. They found that we can substitute the original concepts of walls and doors with objects and interfaces in the designs of our software models. The common thing in both is: at core, both type of patterns are solution to problems in some context.

V. Sarcar, *Interactive Object Oriented Programming in Java*, DOI 10.1007/978-1-4842-2544-8_12

In 1995 the original concepts were discussed with C++. Sun Microsystems released the first public implementation as Java 1.0 in 1995 and it has gone through various changes. So, the key point is: Java was relatively new at that time. Later Java became popular and we were interested to implement the concepts in Java.So, in this book, we'll introduce 3 of those 23 design patterns. These 23 patterns were broadly partitioned into 3 categories.

- Creational patterns
- Structural patterns
- Behavioral patterns

Creational patterns

These patterns are used to abstract the instantiation processes. Applications are built in such a way that they become independent from how their objects are composed or created. Following five patterns come into this category.

- Singleton Pattern
- Prototype Pattern
- Factory Method Pattern
- Builder Pattern
- Abstract Factory Pattern

Structural Patterns

Here we deal with relatively large structures and we focus on how classes and objects can be composed. The concepts of inheritance are widely used here. Following seven patterns fall in this category.

- Proxy pattern
- Flyweight Pattern
- Composite Pattern
- Bridge Pattern
- Facade Pattern
- Decorator Pattern
- Adapter Pattern

Behavioral Patterns

Here we concentrate on algorithms and the assignment of responsibilities among objects. We need to focus on the communication between them. We also give a detailed look on the way by which those objects are interconnected. Following eleven patterns fall in this category.

- Observer Pattern
- Strategy Pattern
- Template Method Pattern
- Command Pattern
- Iterator Pattern
- Memento Pattern
- State Pattern
- Mediator Pattern
- Chain of Responsibility Pattern
- Visitor Pattern
- Interpreter Pattern

We'll pick one from each category for this introductory discussions on design patterns.

Observer Pattern

GoF Definition: Define a one-to-many dependency between objects so that when one object changes state, all its dependents are notified and updated automatically.

Concept

In this pattern, there are many observers (objects) which are observing a particular subject (object). Observers are basically interested and want to be notified when there is a change made inside that subject. So, they register themselves to that subject. The subject must have the freedom-which request he/she will accept. The subject can also discard any of the observers at any time. Sometimes this model is also referred as Publisher-Subscriber model.

Real life Example

We can think about a celebrity who can have many fans in Facebook or any of social sites. Everyday he/she gets friend requests from his fans to include them in his social world. Each of these fans want to get all the latest updates from this celebrity. So, they send their requests to the celebrity to accept them as one of his followers. But the celebrity can decide which request he will accept or not. Our celebrity also has the freedom to discard any of his followers at any time. We can think a fan as an Observer and the celebrity as a Subject in the observer pattern.

Computer world Example

In the world of computer science, consider a simple UI based example-Where this UI is connected with some database (or business logic).A user can execute some query through that UI and after searching the database, the result is reflected back in the UI. In most of the cases we segregate the UI with the database. If a change occurs in the database, the UI should be notified -so that it can update its display forms according the change.

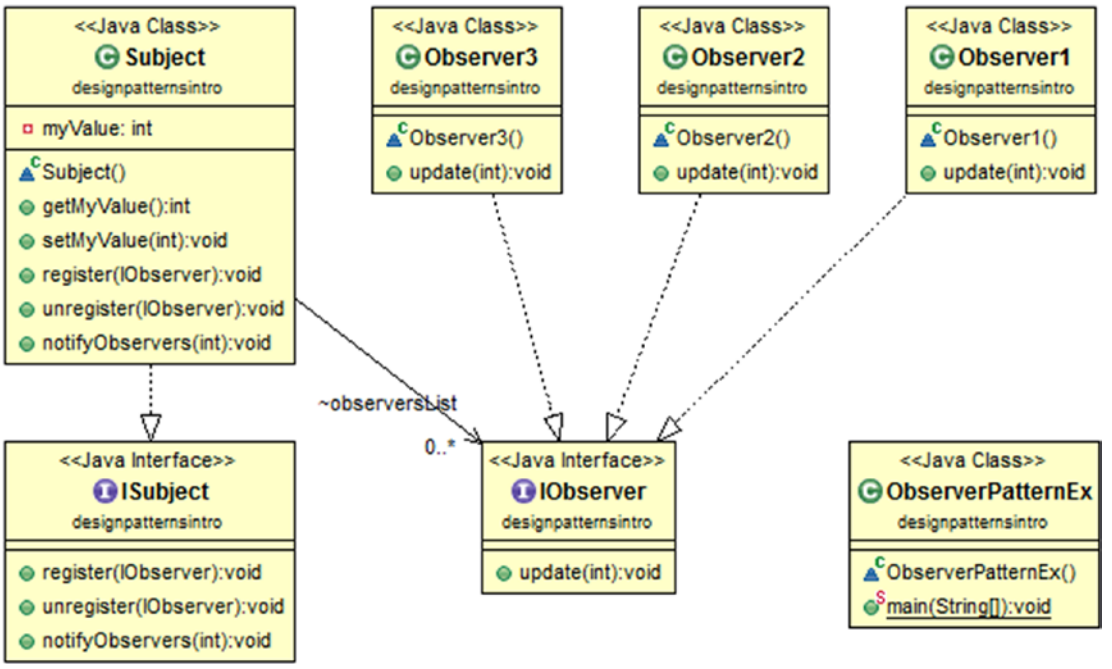
Illustration

In the example below, for simplicity, we have created only three Observers (followers) and one Subject (celebrity). (Though you can create any number of observers and subjects). Subject maintains a list for all of its observer/s .Our observers want to get notification when the flag value changes in the subject (Like when the celebrity change his status).

With the output, we can see that `Observer1` and `Observer2` were getting those notifications initially because the Subject registered them (means granted their request to be his followers).

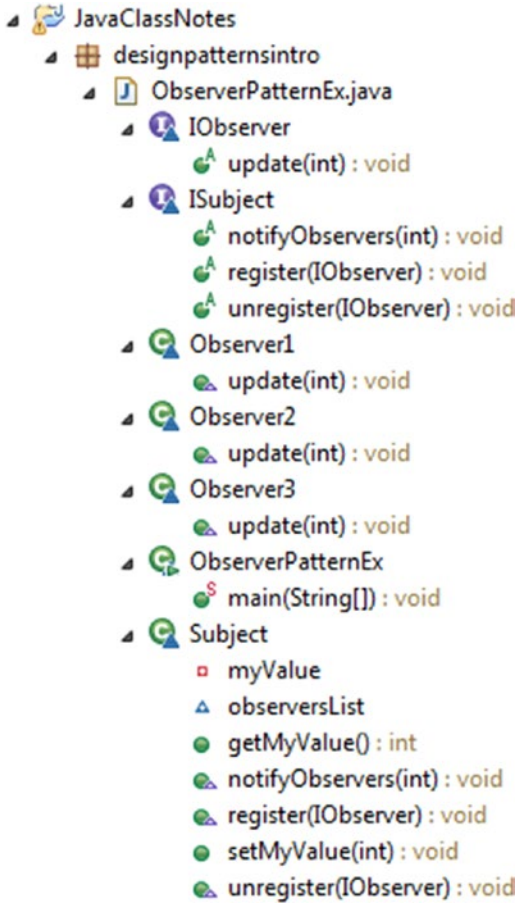
Later the Subject (the celebrity) discard one of his followers from his social world and then only `Observer2` was getting the notification.

Finally our celebrity (Subject) grants one more observer-`Observer3` as one of his followers. So, now both `Observer2` and `Observer3` are getting those alerts for updates/changes in the Subject.


<<Java Class>>
Subject
designpatternsintro
myValue: int
Subject()
getMyValue():int
setMyValue(int):void
register(IObserver):void
unregister(IObserver):void
notifyObservers(int):void
<<Java Class>>
Observer3
designpatternsintro
Observer3()
update(int):void
<<Java Class>>
Observer2
designpatternsintro
Observer2()
update(int):void
<<Java Class>>
Observer1
designpatternsintro
Observer1()
update(int):void
~observersList
0..*
<<Java Interface>>
ISubject
designpatternsintro
register(IObserver):void
unregister(IObserver):void
notifyObservers(int):void
<<Java Interface>>
IObserver
designpatternsintro
update(int):void
<<Java Class>>
ObserverPatternEx
designpatternsintro
ObserverPatternEx()
main(String[]):void

Package Explorer view

High level structure of the parts of the program is as follows:



Implementation

Here is the implementation:

```
package designpatternsintro;
import java.util.*;

interface IObserver
{
        void update(int i);
}
class Observer1 implements IObserver
{
```

```
        @Override
        public void update(int i)
        {
                System.out.println("Observer1 is seeing a change in Subject: "+i);
        }
}
class Observer2 implements IObserver
{
        @Override
        public void update(int i)
        {
                System.out.println("Observer2 get notification from Subject :"+i);
        }
}
class Observer3 implements IObserver
{
        @Override
        public void update(int i)
        {
                System.out.println("Observer3 is seeing the change in Subject :"+i);
        }
}

interface ISubject
{
        void register(IObserver o);
        void unregister(IObserver o);
        void notifyObservers(int i);
}

class Subject implements ISubject
{
        private int myValue;

        public int getMyValue() {
                return myValue;
        }

        public void setMyValue(int myValue) {
                this.myValue = myValue;
                //Notify observers
                notifyObservers(myValue);
        }

        List<IObserver> observersList=new ArrayList<IObserver>();

        @Override
        public void register(IObserver o)
        {
                observersList.add(o);
        }
```

```
        @Override
        public void unregister(IObserver o)
        {
                observersList.remove(o);
        }
        @Override
        public void notifyObservers(int updatedValue)
        {
                for(int i=0;i<observersList.size();i++)
                {
                        observersList.get(i).update(updatedValue);
                }
        }
}

class ObserverPatternEx
{
        public static void main(String[] args)
        {
                System.out.println("*** Observer Pattern Example***\n");
                Subject sub = new Subject();
                Observer1 ob1 = new Observer1();
                Observer2 ob2 = new Observer2();
                Observer3 ob3 = new Observer3();
                //Only Observer1 and Observer2 is registered
                sub.register(ob1);
                sub.register(ob2);

                sub.setMyValue(10);
                System.out.println();
                sub.setMyValue(100);
                System.out.println();

                //unregister Observer1 only
                sub.unregister(ob1);
                //Now only Observer2 will observe the change
                sub.setMyValue(200);
                System.out.println();
                //Take a new follower and register Observer3 now
                sub.register(ob3);
                //Set a new change in subject
                //Now observer 2 and observer 3 will see the change
                sub.setMyValue(500);
        }
}
```

Output

```
<terminated> ObserverPatternEx (1) [Java Application] C:\Program Files\Java\jdk1.8.0_45\bin\javaw.exe (Nov 25, 2016, 9:46:27 AM)
*** Observer Pattern Example***

Observer1 is seeing a change in Subject: 10
Observer2 get notification from Subject :10

Observer1 is seeing a change in Subject: 100
Observer2 get notification from Subject :100

Observer2 get notification from Subject :200

Observer2 get notification from Subject :500
Observer3 is seeing the change in Subject :500
```

Students ask:

In the above example, we are seeing that all observers are being notified at the same time. But in some cases, we may need to send notification to only one observer and not to others. Then how that scenario can be handled by this pattern?

Teacher says: You must understand that each of the pattern has its own intention. The scenario you explained can be handled easily with another design pattern named-Chain of responsibility. There we can process a series of objects one by one i.e. in a sequential manner. With that pattern, a source can initiate that processing. Then we'll constitute a chain where each of the processing object can have some logic to handle a particular type of command object and after one's processing is done, if anything is still pending-they can be forwarded to the next object in the chain. We can also add new objects anytime at the end of the chain.

Consider a real world situation. In an organization, there are some customer care executives who collects feedbacks from customers and forward those customer issues/escalations to appropriate departments in the organization. Not all departments will start fixing an issue. The department who seems to be responsible will take a look first and if they believe that issue should be forwarded to another department, they will do that.

But our observer pattern works better in situations where we want to notify all observers at the same time with a centralized announcement. It does not care how many objects will follow its announcement .So, this pattern should be used to deal with similar models only.

Students ask:

From the above example, it appears to us that we need to create separate classes for each of the observer. Is the understanding correct?

Teacher says: Not at all. We could simply create one observer class and different objects of that. And then we can follow the similar registration and unregistration process.Similarly we could vary the update methods in each of them-they do not need to be similar.

Prototype Pattern

GoF Definition: Specify the kinds of objects to create using a prototypical instance, and create new objects by copying this prototype.

Concept

It provides an alternative method for instantiating new objects. In real world, to create a brand a new instance every time is normally treated as an expensive operation. This pattern helps us to reduce that expense by making a clone (or copy) of an actual instance. So, this cloning operations plays the vital role here.

Real life Example

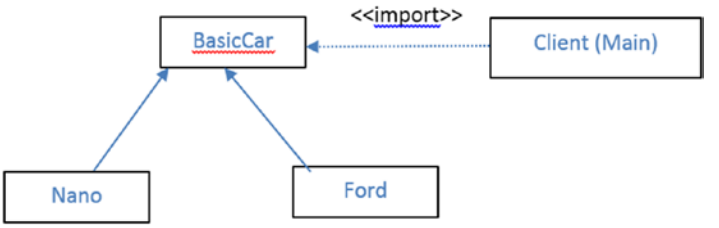
Suppose we have a master copy of a valuable document. Now, we want to experiment some changes on it. In most of the cases we make a replica first by making a photocopy of this .Then we try to incorporate our changes into that document.

Computer world Example

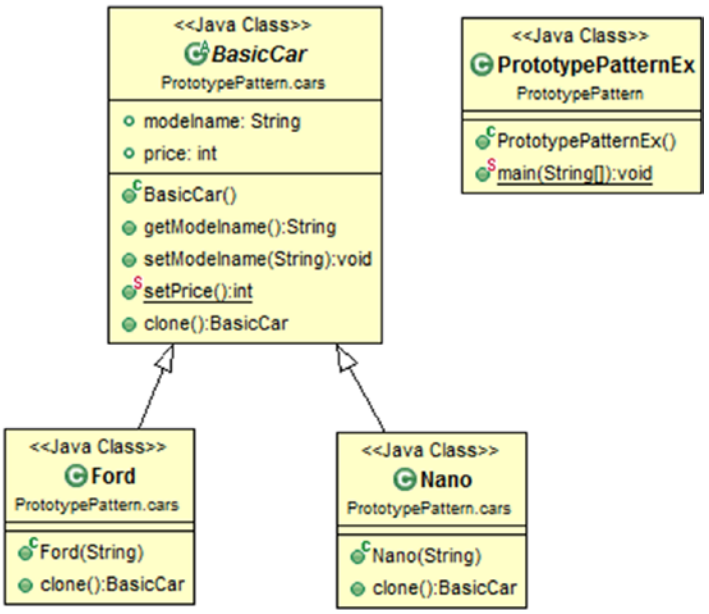
Suppose we have made an application for an inexpensive mobile device which does not support touch screen. Next time we want the device to support the touchscreen mechanism to attract more customers. So, you must notice that basic functionalities of the device must be the same, we are basically adding some more features in it. Then we must start with a copy from our master copy application and try building the new features on top of it. Definitely, we'll not start from the scratch.

Illustration

In the below example, we are going to follow this structure:

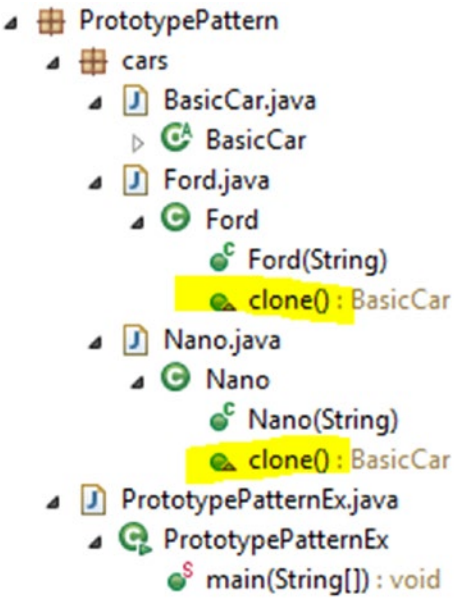


Here `BasicCar` is our prototype. `Nano` and `Ford` are our concrete prototypes and they need to implement the `clone()` method defined in `BasicCar`. We can notice that a `BasicCar` model is created with some default price. Later we have modified that price as per the model. Please also note that `PrototypePatternEx` is the client here. As usual, the related parts are separated by the packages for better readability.



Package Explorer view

High level structure of the parts of the program is as follows:



Implementation

Here is the implementation:

```
package PrototypePattern;
import PrototypePattern.cars.BasicCar;
import PrototypePattern.cars.Ford;
import PrototypePattern.cars.Nano;

public class PrototypePatternEx
{
    public static void main(String[] args) throws CloneNotSupportedException
    {
        System.out.println("***Prototype Pattern Demo***\n");
        BasicCar nano_base = new Nano("Green Nano") ;
        nano_base.price=200000;

        BasicCar ford_basic = new Ford("Ford Yellow");
        ford_basic.price=500000;

        BasicCar bc1;
        //Nano
        bc1 =nano_base.clone();
       //Price will be more than 200000 for sure
        bc1.price = nano_base.price+BasicCar.setPrice();
        System.out.println("Car is: "+ bc1.modelname+" and it's price is Rs."+bc1.price);

        //Ford
        bc1 =ford_basic.clone();
        //Price will be more than 500000 for sure
        bc1.price = ford_basic.price+BasicCar.setPrice();
        System.out.println("Car is: "+ bc1.modelname+" and it's price is Rs."+bc1.price);
    }
}
```

Output

```
<terminated> PrototypePatternEx (1) [Java Application] C:\Program Files\Java\jdk1.8.0_45\bin\javaw.exe (Oct 1, 2016, 4:59:06 PM)
***Prototype Pattern Demo***

Car is: Green Nano and it's price is Rs.205353
Car is: Ford Yellow and it's price is Rs.580157
```

Note that:

- When the system cares only to make a replica but do not care about the creational mechanism of the products-this pattern is very much helpful.
- We can use this pattern when we need to instantiate classes at runtime.
- In our example, we have used the default `clone()` method in Java-Which is a shallow copy. So, it is inexpensive compared to a deep copy.

Students ask:

Sir, with this pattern, we can implement runtime polymorphism. Is the understanding correct?
Teacher says: Yes. The above example depicts that clearly.

Students ask:

Sir, what are the advantages of the prototype patterns?
Teacher says:

- We can include or discard products at run time.
- We can create new instances with a cheaper cost.

Students ask:

Are there any disadvantages of the prototype pattern?
Teacher says:

- Each subclass must have to implement the cloning mechanism which is not always very easy e.g. implementing cloning mechanism can be challenging if the objects under consideration does not support copying or if there is any kind of circular references.

Bridge Pattern

GoF Definition: Decouple an abstraction from its implementation so that the two can vary independently.

Concept

In this pattern, abstract class is separated from the implementation class and we provide a bridge interface between them. This interface helps us to make concrete class functionalities independent from the interface implementer class. We can alter these different kind of classes structurally without affecting each other.

Real life Example

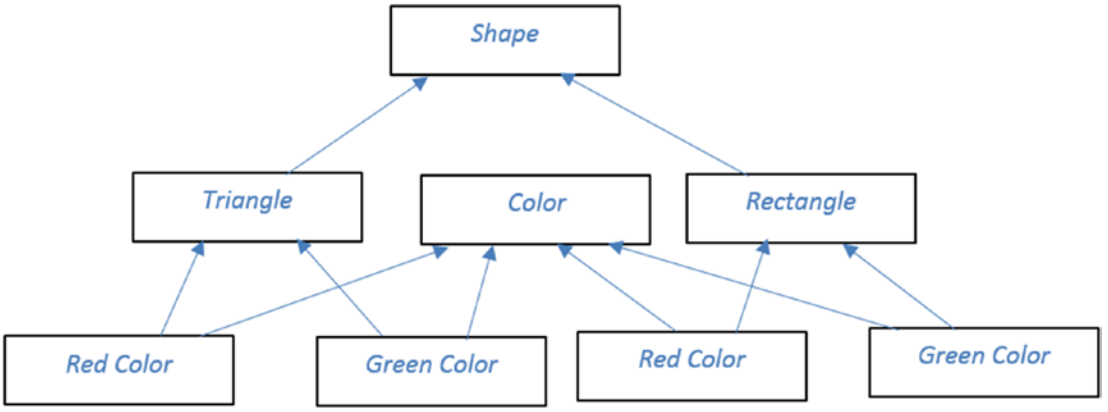
In a software product development company, development team and technical support team-both play the crucial role. The change in the operational strategy in any of the team should not have a direct impact on the other team. Here the technical support team plays the role of a bridge between the clients and the development team that implements the product.

Computer world Example

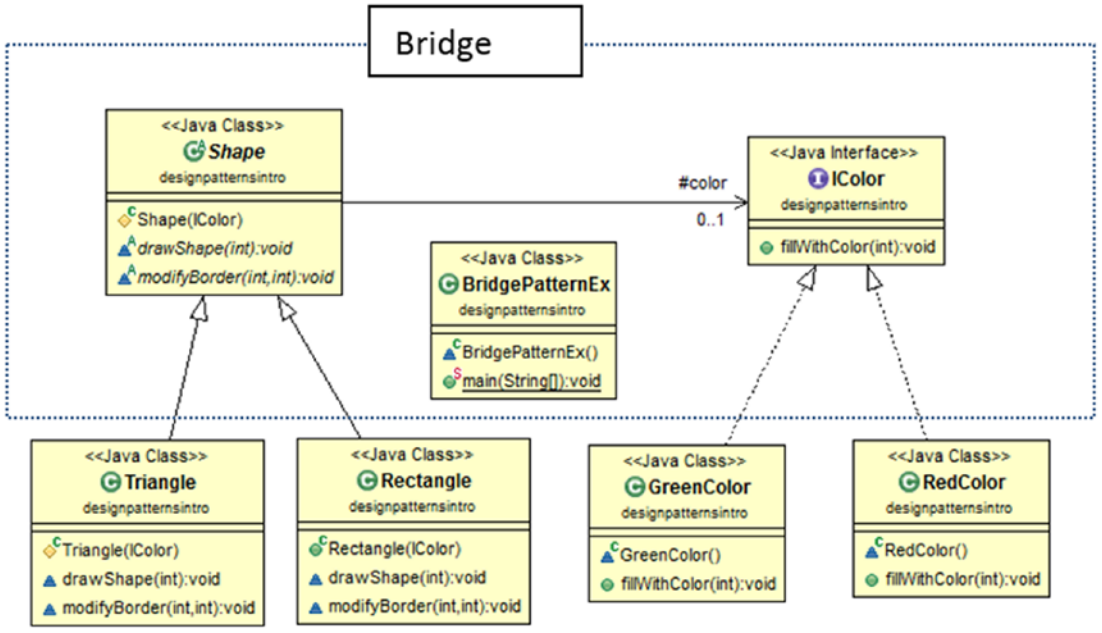
This pattern is used in GUI framework. It separates Window abstraction from Window implementation in Linux/Mac OS. Also the below illustration is one of the classical example in a software development field.

Illustration

Consider a situation like this:

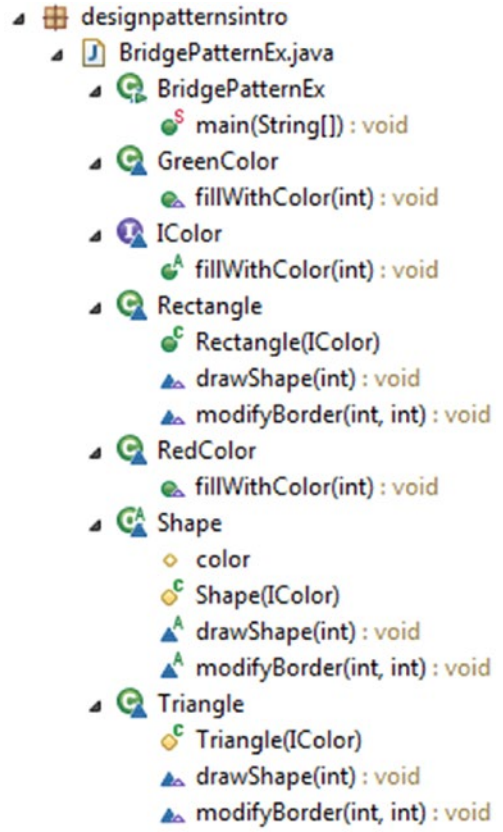


We'll use bridge pattern to decouple the interfaces in our example from the implementations. After our implementation it will have a cleaner look (Follow our UML diagram).



Package Explorer view

High level structure of the parts of the program is as follows:



Implementation

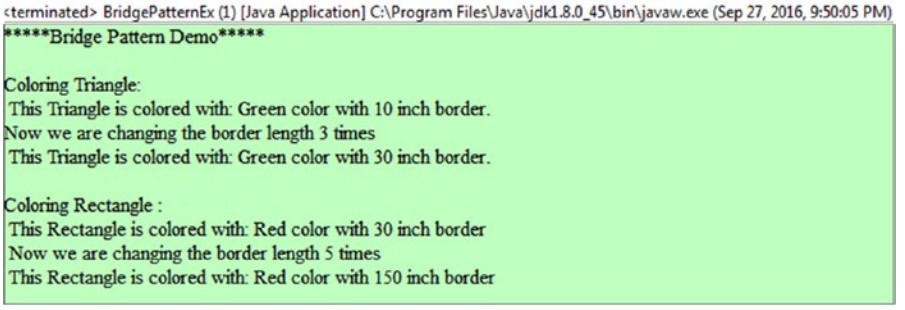
Here we have implemented both an abstraction specific and an implementer specific method to represent the power and usefulness of this pattern. We can draw `Triangle` and `Rectangle` with a particular color with the implementer specific method `drawShape()`. We can change the thickness of the border by the abstraction specific method `modifyBorder()`. Now go through the code.

```
package designpatternsintro;

//Colors-The Implementer

interface IColor
{
        void fillWithColor(int border);
}
```

```
class RedColor implements IColor
{
        @Override
        public void fillWithColor(int border)
        {
                System.out.print("Red color with " +border+" inch border");

        }
}
class GreenColor implements IColor
{
        @Override
        public void fillWithColor(int border)
        {
                System.out.print("Green color with " +border+" inch border.");

        }
}

//Shapes-The Abstraction

abstract class Shape
{
        //Composition
        protected IColor color;
        protected Shape(IColor c)
        {
                this.color = c;
        }
        abstract void drawShape(int border);
        abstract void modifyBorder(int border,int increment);
}

class Triangle extends Shape
{
        protected Triangle(IColor c)
        {
                super(c);
        }
        //Implementer specific method
        @Override
        void drawShape(int border) {
                System.out.print(" This Triangle is colored with: ");
                color.fillWithColor(border);
        }
        //Abstraction specific method
        @Override
        void modifyBorder(int border,int increment) {
                System.out.println("\nNow we are changing the border length "+increment+ " times");
                border=border*increment;
                drawShape(border);
        }
}
```

```
class Rectangle extends Shape
{
        public Rectangle(IColor c)
        {
                super(c);
        }
        //Implementer specific method
        @Override
        void drawShape(int border)
        {
                System.out.print(" This Rectangle is colored with: ");
                color.fillWithColor(border);
        }
        //Abstraction specific method
        @Override
        void modifyBorder(int border,int increment) {
                System.out.println("\n Now we are changing the border length "+increment+ "
                times");
                border=border*increment;
                drawShape(border);
        }
}

class BridgePatternEx
{
  public static void main(String[] args)
  {
          System.out.println("*****Bridge Pattern Demo*****");
                //Coloring Green to Triangle
                System.out.println("\nColoring Triangle:");
                IColor green = new GreenColor();
                Shape triangleShape = new Triangle(green);
                int initialBorder=10;
                triangleShape.drawShape(initialBorder);
                //Modifying the border length
                triangleShape.modifyBorder(initialBorder, 3);

                //Coloring Red to Rectangle
                System.out.println("\n\nColoring Rectangle :");
                IColor red = new RedColor();
                Shape rectangleShape = new Rectangle(red);
                //nitial border for Rectangle
                initialBorder=initialBorder*3;
                rectangleShape.drawShape(initialBorder);
                //Modifying the border length
                rectangleShape.modifyBorder(initialBorder,5);
  }
}
```

Output



Note that:

- The pattern is extremely helpful when our class and its associated functionalities may change in frequent intervals.
- Here we have removed the concrete binding between an abstraction and the corresponding implementation. As a result, both hierarchy (abstraction and its implementations) can grow independently. So, the benefit is: if we make any change in abstraction methods, they do not have impact on implementer methods (i.e. in `fillWithColor()`).

Students ask:

Sir, you have repeatedly referred here about the two hierarchies-abstraction and implementer. We could use interface to represent the abstraction"-is the statement correct?

Teacher says: Absolutely correct. We can use either an abstract class or an interface. And same rule applies for implementer also.

Probably you have noticed here that we have used the concept of composition here. This concept is very much useful to avoid the diamond problem in Java. (We must remember that Java does not support multiple inheritance through classes and diamond problem is one of key reason for that).

Teacher asks:

What is refined abstractions?
Answer: Children of an abstraction are termed as refined abstractions.

Teacher asks:

Who are concrete implementers?
Answer: Children of an implementer.

Students ask:

How can we differentiate an abstraction from its implementer?
Teacher says: In general, an abstraction contains the reference to its implementer.

Teacher asks:

Can you tell me: how can we implement the polymorphic behavior of the implementers?
Answer: By changing the reference in the abstraction.

APPENDIX A

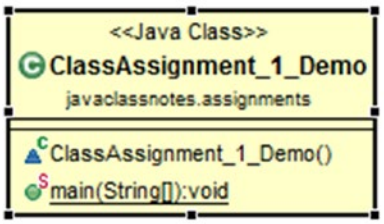
Solution to the Assignments

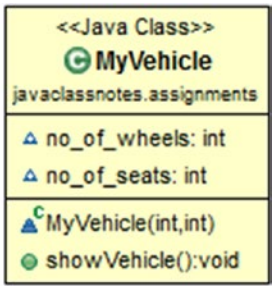
Class

Assignment 1

Create a class `Vehicle`. The class should have two fields-`no_of_seats` and `no_of_wheels`. Create two objects-`Motorcycle` and `Car` for this class. Your output should show the descriptions for `Car` and `Motorcycle`.

Uml class diagram:





Implementation

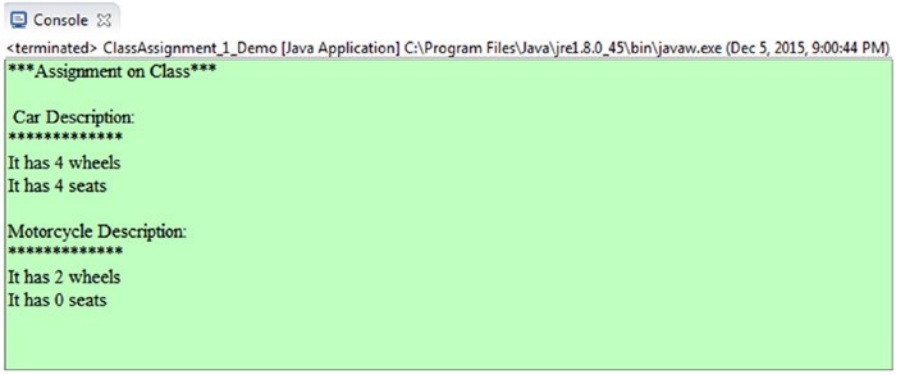
No need to include package statements. But once you understand the concept of package, we'll see that it is very much handy to make an organized structure for our programs.

```
package assignments;

class MyVehicle
{
        int no_of_wheels;
        int no_of_seats;
        MyVehicle(int wheels,int seats)
        {
                no_of_wheels=wheels;
                no_of_seats=seats;
        }
```


V. Sarcar, *Interactive Object Oriented Programming in Java*, DOI 10.1007/978-1-4842-2544-8

```
        public void showVehicle()
        {
                System.out.print("Description:");
                System.out.println("\n*************");
                System.out.println("It has "+ no_of_wheels+" wheels");
                System.out.println("It has "+ no_of_seats+" seats\n");
        }
}

class ClassAssignment_1_Demo
{
        public static void main(String args[])
        {
                System.out.print("***Assignment on Class***\n\n ");
                MyVehicle car=new MyVehicle(4,4);
                MyVehicle motorCycle=new MyVehicle(2,0);
                System.out.print("Car ");
                car.showVehicle();
                System.out.print("Motorcycle ");
                motorCycle.showVehicle();
        }

}
```

Output



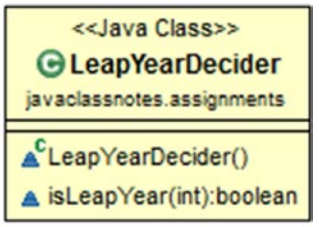
Assignment 2

Create a class with a method. The method has to decide whether a given year is a leap year or not.

Note- A year is a leap year if:

- It has an extra day i.e. 366 instead of 365.
- It occurs in every 4 year e.g. 2008, 2012 are leap years.
- For every 100 years a special rule applies-1900 is not a leap year but 2000 is a leap year. In those cases, we need to check whether it is divisible by 400 or not.

UML class diagram:



Implementation

Here is the impelmentation:

```
package assignments;

import java.io.BufferedReader;
import java.io.IOException;
import java.io.InputStreamReader;
//import java.util.Scanner;

class LeapYearDecider
{
        boolean isLeapYear(int yr)
        {
                if (yr%400 !=0)
                {
                        if((yr%4==0) && (yr%100 !=0))
                        {
                                return true;
                        }
                        else
                                return false;
                }
                return true;
        }
}

public class ClassAssignmentEx2
{
        public static void main(String args[]) throws IOException
        {
                System.out.print("***Assignment on Class. Ex-2***\n\n ");
                System.out.print("***Test -whether a given year is a leap year or not***\n ");
                System.out.print("Enter an year\n\n ");
                BufferedReader br = new BufferedReader( new InputStreamReader( System.in ));
                String str= br.readLine();
                             int input=0;
                try
                {
```

```
                input=Integer.parseInt(str);
                }
                catch(NumberFormatException e)
                {
                        System.out.print(" Caught invalid Input : " +e);
                        //A non zero status to indicate abnormal //termination
                        System.exit(-1);
                }
                /*Or we can use the scanner class to take input.Java 5   added this scanner class.
              Scanner in=new Scanner(System.in);
               int input=Integer.parseInt(in.nextLine());
                 */
                //LeapYearDeciderclass object
                LeapYearDecider ob=new LeapYearDecider();
                System.out.print( input +" is a leap year ? "+ob.isLeapYear(input));
        }
}
```

Output

Case 1: Positive case-2012 is a leap year.

```
<terminated> ClassAssignmentEx2 [Java Application] C:\Program Files\Java\jdk1.8.0_45\bin\javaw.exe (Aug 22, 2016, 10:09:50 PM)
***Assignment on Class. Ex-2***

***Test -whether a given year is a leap year or not***
Enter an year

2012
2012 is a leap year ? true
```

Case 2: Negative case-2005 is not a leap year.

```
<terminated> ClassAssignmentEx2 [Java Application] C:\Program Files\Java\jdk1.8.0_45\bin\javaw.exe (Aug 22, 2016, 10:11:33 PM)
***Assignment on Class. Ex-2***

***Test -whether a given year is a leap year or not***
Enter an year

2005
2005 is a leap year ? false
```

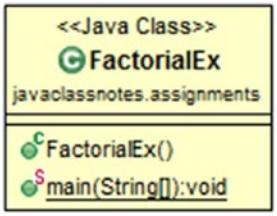
Case 3: Special case-1900 is not a leap year.

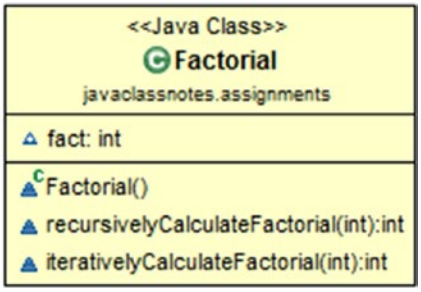
```
<terminated> ClassAssignmentEx2 [Java Application] C:\Program Files\Java\jdk1.8.0_45\bin\javaw.exe (Aug 22, 2016, 10:12:28 PM
***Assignment on Class. Ex-2***

***Test -whether a given year is a leap year or not***
Enter an year

1900
1900 is a leap year ? false
```

Assignment 3

Create a class with two functions-one recursive and one non recursive. Either of these function should be capable of calculating the factorial of a number.

UML class diagram:





Implementation

Here is the implementation

```
package assignments;

class Factorial
{
        int fact;
        //recursive version
        int recursivelyCalculateFactorial(int i)
        {
                fact=1;
                if (i==1 || i==0)
                        return 1;
                else
                {
                        fact=i*recursivelyCalculateFactorial(i-1);
                        return fact;
                }
        }
        //nonrecursive(iterative) version
        int iterativelyCalculateFactorial(int i)
        {
                if (i==1 || i==0)

                {
                        return 1;
                }
                else
                {
                        fact=1;
                        for(int j=1;j<=i; j++)
```

```
                        {
                                fact=fact*j;
                        }
                        return fact;
                }
        }
}

public class FactorialEx {
        public static void main(String args[])
        {
                System.out.println("*** Calculating factorial ***");
                Factorial factOb=new Factorial();
                System.out.println("*** By using recursive version ***");
                System.out.println("Factorial of 7 is :"+ factOb.recursivelyCalculateFactorial(7));
                System.out.println("*** By using nonrecursive(iterative) version ***");
                System.out.println("Factorial of 6 is :"+ factOb.iterativelyCalculateFactorial(6));
        }
}
```

Output

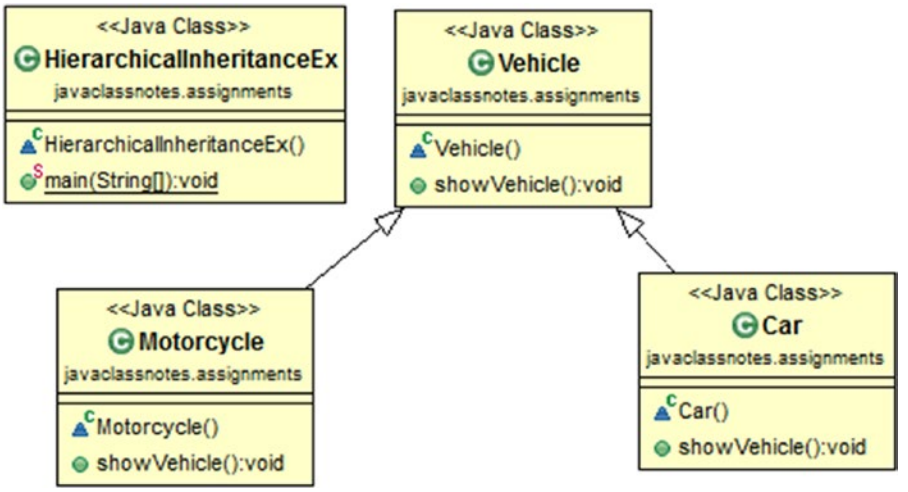
```
<terminated> FactorialEx [Java Application] C:\Program Files\Java\jdk1.8.0_45\bin\javaw.exe (Aug 24, 2016, 7:03:39 PM)
*** Calculating factorial ***
*** By using recursive version ***
Factorial of 7 is :5040
*** By using nonrecursive(iterative) version ***
Factorial of 6 is :720
```

Inheritance

Assignment 1

Write a simple program to implement hierarchical inheritance.

UML class diagram:



Implementation

Here is the implementation

```
package assignments;

class Vehicle
{
        public void showVehicle()
        {
                System.out.println("I am in Vehicle");
        }
}
class Car extends Vehicle
{
        public void showVehicle()
        {
                System.out.println("I am in Car");
        }
}
class Motorcycle extends Vehicle
{
        public void showVehicle()
        {
                System.out.println("I am in Motorcycle");
        }
}
```

```
class HierarchicalInheritanceEx
{
        public static void main(String args[])
        {
                System.out.println("***Hierarchical Inheritance Demo***");
                Car c=new Car();
                c.showVehicle();
                Motorcycle m=new Motorcycle();
                m.showVehicle();
        }
}
```

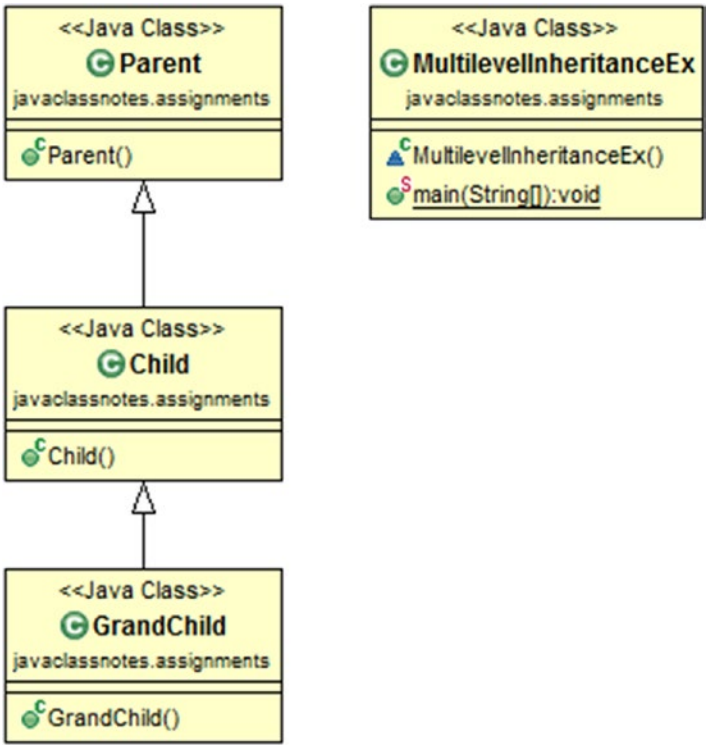
Output

Assignment 2

Write a simple program to implement multilevel inheritance.

We could write the program like the above program (See our code structure in the chapter on Inheritance).Just for a variety, we are using only constructors here. If you want to see the concrete methods implementation, just uncomment the codes in the below program.

UML class diagram:



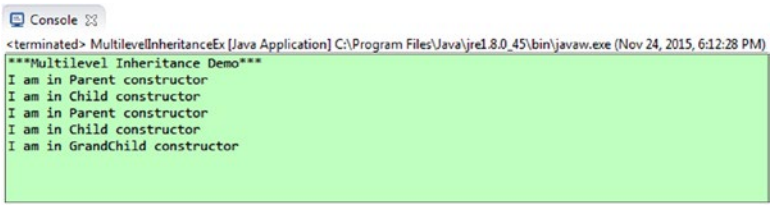
Implementation

Here is the implementation:

```
package assignments;

class Parent
{
        public Parent()
        {
                System.out.println("I am in Parent constructor");
        }
        /*public void showMe()
        {
                System.out.println("I am a Parent");
        }        */
}
class Child extends Parent
{
        public Child()
        {
                System.out.println("I am in Child constructor");
```

```
        }
        /*public void showMe()
        {
                System.out.println("I am a Child");
        }        */
}
class GrandChild extends Child
{
        public GrandChild()
        {
                System.out.println("I am in GrandChild constructor");
        }
        /*public void showMe()
        {
                System.out.println("I am a GrandChild");
        }*/
}

class MultilevelInheritanceEx
{
        public static void main(String args[])
        {
                System.out.println("***Multilevel Inheritance Demo***");
                //Parent p=new Parent();
                //p.showMe();
                Child c=new Child();
                //c.showMe();
                GrandChild g=new GrandChild();
                //g.showMe();
        }
}
```

Output



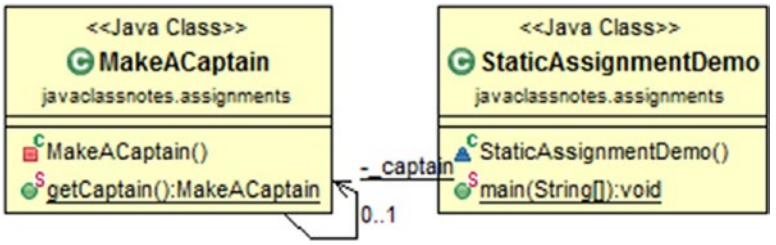
Note the output. Whenever we want to initiate a child class object, the parent class constructors called automatically first. This is why, `Child` class constructor called `Parent` class constructor first. Similarly, our `GrandChild` class constructor called its parent (i.e. `Child` class constructor) which in turn called its parent (i.e. `Parent` class constructor) constructor first.

Use of static keyword

Assignment

Suppose you have formed a cricket team. Now your team is going to play against an opponent team. You must be aware of the fact that which team will bat (or bowl) first will be decided through the toss and you need to send your captain for that. So, at first, you must elect a captain. At the same time, you must be aware that you can select one and only one captain. So, if you do not have any such captain, you will select one and send him for toss. Otherwise, you simply send the already nominated captain for the toss. Can you design this?

UML class diagram:



Implementation

Here is the implementation:

```
package assignments;


class NominateACaptain
{
        private static NominateACaptain _captain;
        //We make the constructor private to prevent the use of "new"
        private NominateACaptain() { }
        //To make the code thread safe using the synchronized version.
        // public static  NominateACaptain getCaptain()
        public static  synchronized NominateACaptain getCaptain()
        {
                // Lazy initialization
                if (_captain == null)
                { _captain = new NominateACaptain();
                System.out.println("We have selected the captain for our team");
                }
                else
                {
                        System.out.print(" We already have a Captain.");
                        System.out.println(" We'll send him for the toss.");
                }
                return _captain;
        }
}
```

```
class StaticAssignmentDemo
{
        public static void main(String[] args)
        {
                System.out.println("***Static Assignment Demo***\n");
                System.out.println("Trying to make a captain for our team");
                NominateACaptain c1 = NominateACaptain.getCaptain();
                System.out.println("Trying to make another captain for our team");
                NominateACaptain c2 = NominateACaptain.getCaptain();
                if (c1 == c2)
                {
                        System.out.println("c1 and c2 are same instance");
                }
        }
}
```

Output

Note that:

- We have used the term “Lazy initialization” because, the singleton instance will not be created until the `getCaptain()` method is called .
- To make the implementation thread safe we have used the keyword `synchronized`. Because, there may be a situation when two or more threads come into picture and they try to create more than one objects of the singleton class.
- But if we use the synchronized keyword in our program, we need to pay for the performance cost associated with this synchronization method also.

4. So, to implement the thread safety, developers came up with different other solutions also. But there is always pros and cons. But I want to highlight some of them.

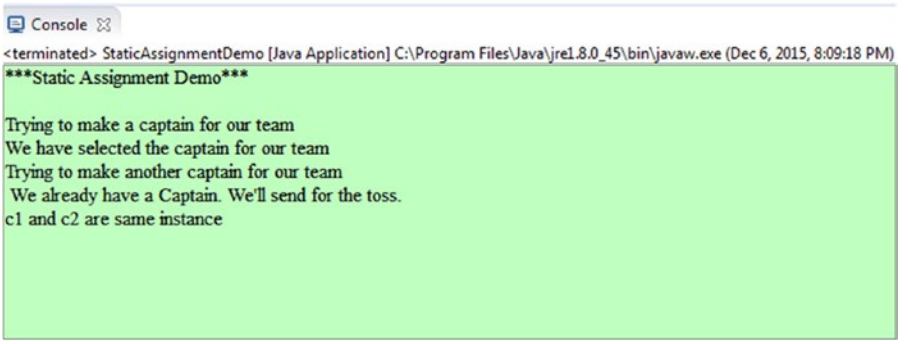
There is another method called *"Eager Initialization"* (opposite of "Lazy initialization" mentioned in our original code) to achieve thread safety.

```
class MakeACaptain
      {
          //Early initialization
          private static NominateACaptain _captain = new NominateACaptain();
          //We make the constructor private to prevent the use of "new"
          private NominateACaptain () { }

          // Global point of access //NominateAcaptain.getCaptain() is a public static //method
          public static NominateACaptain getCaptain()
          {
              return _captain;
          }
      }
```

In the above solution an object of the singleton class is always instantiated. To deal with this kind of situation, ***Bill Pugh*** came up with a different approach:

```
class NominateACaptain
{
        private static NominateACaptain _captain;
        private NominateACaptain () { }

      //Bill Pugh solution
      private static class SingletonHelper{
      //Nested class is referenced after getCaptain() is called

          private static final NominateACaptain _captain = new NominateACaptain ();
      }

      public static NominateACaptain getCaptain()
      {
          return SingletonHelper._captain;
      }

  }
```

This method does not need to use synchronization technique and eager initialization. It is also regarded as one of the standard method to implement singletons in Java.

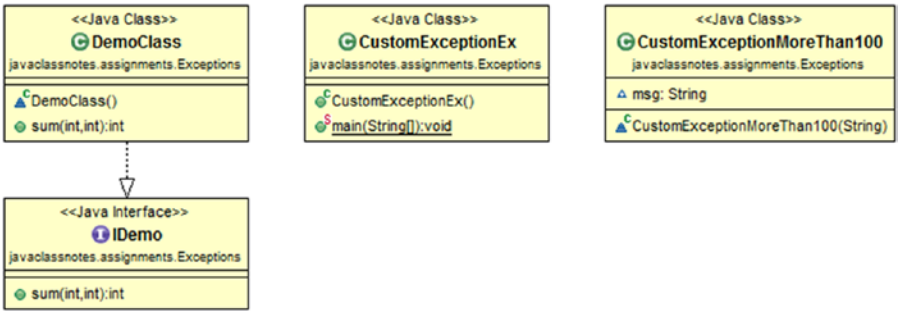


Exceptions

Assignment

Create a custom `Exception` class. You need to consider two integer inputs which must be supplied by the user. You will display the sum of the integers if and only if the sum is less than 100. If it is not less than 100, throw your custom exception.

UML class diagram:



Implementation

Here is the implementation:

```
package assignments;
import java.util.Scanner;//To take user input here
class SumGreaterThan100Exception  extends Exception
{
        String msg;
        SumGreaterThan100Exception(String msg)
        {
                this.msg=msg;
        }
}
interface IDemo
{
        int sum(int x,int y) throws SumGreaterThan100Exception;
}

class DemoClass implements IDemo
{
        public int sum(int x,int y) throws SumGreaterThan100Exception
        {
                int sumofIntegers=x+y;

                if(sumofIntegers<=100)
                {
                        return sumofIntegers;
                }
                else
                {
                        throw new SumGreaterThan100Exception ("Sum is greater than 100");
                }
        }
}
```

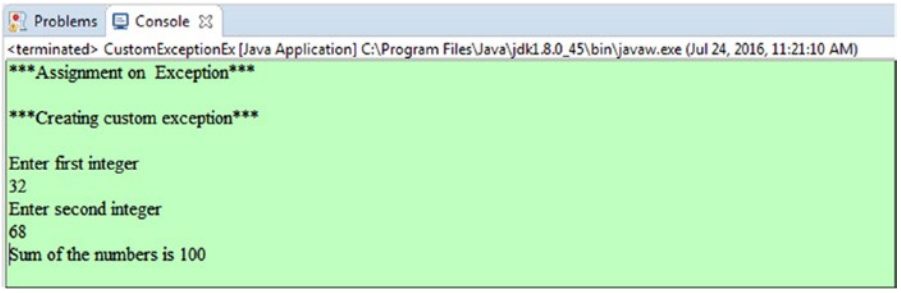
```
public class CustomExceptionEx {
        public static void main(String args[])
        {
                System.out.println("***Assignment on  Exception***\n");
                System.out.println("***Creating custom exception***\n");
                //For Java old versions-use  BufferedReader e.g.
                //BufferedReader br = new BufferedReader( new InputStreamReader( System.in ));
                //String c = br.readLine();
                Scanner in= new Scanner(System.in);//To take input from user
                int number1,number2;
                System.out.println("Enter first integer");
                number1=Integer.parseInt(in.nextLine());
                System.out.println("Enter second integer");
                number2=Integer.parseInt(in.nextLine());
                DemoClass demo=new DemoClass();
                try
                {
                        int result=demo.sum(number1, number2);
                        System.out.println("Sum of the numbers is "+ result);
                }
                catch(SumGreaterThan100Exception  e)
                {
                        System.out.println( "Caught the custom exception : "+e);
                        e.printStackTrace();
                }
        }
}
```

Output

Case 1: When sum of the inputs are greater than 100

```
<terminated> CustomExceptionEx (1) [Java Application] C:\Program Files\Java\jdk1.8.0_45\bin\javaw.exe (Nov 27, 2016, 10:55:36 AM)
***Assignment on  Exception***

***Creating custom exception***

Enter first integer
45
Enter second integer
78
Caught the custom exception : assignments.SumGreaterThan100Exception
assignments.SumGreaterThan100Exception
        at assignments.DemoClass.sum(CustomExceptionEx.java:28)
        at assignments.CustomExceptionEx.main(CustomExceptionEx.java:49)
```

Case 2: When sum of the inputs are less or equal to 100



Discussion

For the above assignment, it was not required to create an interface on top of our `DemoClass`. But you will later find that in real world programming, converting class types to interface types can give you many benefits. We have already discussed such an advantage in the discussion of a tagging interface.

APPENDIX B

Frequently asked questions

Now it is the time to test your understanding. Please go through the questions. If there is any doubt, please go back to the respective topic/s.

1. What is a class?
2. What is an object?
3. Differentiate between object and reference?
4. Can we implement multiple inheritance in Java?
5. Can we implement hybrid inheritance in Java?
6. Differentiate between an abstract class and an interface.
7. Differentiate between method overloading and method overriding.
8. How you can implement dynamic polymorphism in Java?
9. "Package statement should always come on top"-is it true?
10. What is JVM?
11. Differentiate between JRE and JDK.
12. What is an inner class?
13. How you can create a static class in java?
14. How you can implement abstraction and encapsulation in Java?
15. Differentiate between a static binding and a dynamic binding in Java.
16. What is use of `super` in Java?
17. What is/are the use/s of `this` in Java?
18. What is use of `default` in Java?
19. Can you use an abstract class without an abstract method?
20. Can you inherit constructors?
21. What is the use of `final` in Java?
22. Differentiate between an instance method and a class method (static method)?
23. Can you create a static block? What is its use?

V. Sarcar, *Interactive Object Oriented Programming in Java*, DOI 10.1007/978-1-4842-2544-8

24. What is the default package in Java?
25. Does Java support pointers?
26. What is an exception?
27. What is the superclass of `Exception`?
28. What do you mean by checked exceptions?
29. What do you mean by unchecked exceptions?
30. How can you create your own exception?
31. Mention some example of error condition.
32. What is the difference between `throw` and `throws`?
33. What will happen if you encounter exception in `finally` block?
34. What is the drawback of handling exceptions?
35. What is the advantage of handling exceptions?
36. What do you mean by chained exceptions?
37. What are the key methods associated with chained exceptions?
38. What is garbage collection?
39. What is finalization?
40. How can you remove memory leak in Java?
41. What is the purpose of a constructor?
42. What are the different types of constructors?
43. How can you differentiate a user defined no-argument constructor with a default constructor in Java?
44. Can the use of default method in the interfaces lead to diamond problem in Java?
45. Java does not support multiple inheritance through classes but C++ supports. Do you treat it as an advantage or disadvantage of Java?
46. Can multiple variable reference a same object in memory?
47. What is the expected output if you use the following line of code?

```
System.out.print(anObject);
```

48. What is constructor chaining?
49. How can you pass variable number of arguments inside a method?
50. What is the difference between `char` in Java vs `char` in C/C++?
51. What do you mean by automatic type conversion?
52. Do you treat Java as a purely object oriented language? If not why?
53. Can a constructor be private?

54. Can a constructor be final or abstract or static?
55. Differentiate among final, finally and finalize.
56. What is the difference between a default constructor and a no-argument constructor?
57. Can we have both `this()` and `super()` in the same constructor? If not, why?
58. Can we have backward inheritance?
59. In an inheritance hierarchy, how can you decide a parent class or a child class?
60. What is a blank `final` variable? How can you use it in your program?
61. Can we override an overloaded method?
62. If we make the `main()` method final, will we receive any compile time or run time error?
63. What is the advantages of a tagging interface?
64. Is there any alternative to marker interfaces?
65. Which one you like- use of marker interfaces or use of marker annotations? Why?
66. How can you implement the concept of generalization/specialization in Java?
67. How can you implement the concept of realization in Java?
68. Is there any alternative to inheritance? What is that? When you can use that concept?
69. Does Java support structures? If not, why?
70. What is the basic difference between String and StringBuffer?
71. How you can distinguish a Java applet from a Java application.
72. What is the difference between StringBuffer and StringBuilder?
73. In which scenario, you prefer StringBuilder over StringBuffer (and vice versa)?

APPENDIX C

Some Useful Resources

Java: The Complete Reference by Herbert Schildt. Publisher: McGraw Hill Education
Head First Java by Sierra, Bates Publisher: O'Reilly Media
Java Design Patterns by Vaskaran Sarcar Publisher: Apress

Online Resources:

- http://www.javatpoint.com/
- http://www.tutorialspoint.com/
- http://beginnersbook.com/
- https://docs.oracle.com/javase/tutorial
- http://www.javabeat.net/
- http://www.geeksforgeeks.org/
- http://mindprod.com/
- http://www.programmerinterview.com/
- http://www.javaworld.com/

V. Sarcar, *Interactive Object Oriented Programming in Java*, DOI 10.1007/978-1-4842-2544-8

Index

V. Sarcar, *Interactive Object Oriented Programming in Java*, DOI 10.1007/978-1-4842-2544-8

E, F

G

H

I, J, K, L

M

N

O

P, Q

R

S, T

U, V

W, X, Y, Z