**1)Introduction to Java**

Topic 1: Power of Java

**Java** is an [object-oriented programming](http://www.freejavaguide.com/object_oriented_programming.html) language developed by James Gosling and colleagues at Sun Microsystems in the early 1990s. Unlike conventional languages which are generally designed either to be compiled to native (machine) code, or to be interpreted from source code at runtime, Java is intended to be compiled to a bytecode, which is then run (generally using JIT compilation) by a [Java Virtual Machine](http://www.freejavaguide.com/virtual_machine.html).

The language itself borrows much syntax from C and C++ but has a simpler object model and fewer low-level facilities. Java is only distantly related to JavaScript, though they have similar names and share a C-like syntax.

**History**

Java was started as a project called "Oak" by James Gosling in June 1991. Gosling's goals were to implement a virtual machine and a language that had a familiar C-like notation but with greater uniformity and simplicity than C/C++. The first public implementation was Java 1.0 in 1995. It made the promise of "Write Once, Run Anywhere", with free runtimes on popular platforms. It was fairly secure and its security was configurable, allowing for network and file access to be limited. The major web browsers soon incorporated it into their standard configurations in a secure "[applet](http://www.freejavaguide.com/java_applet.html)" configuration. popular quickly. New versions for large and small platforms (J2EE and J2ME) soon were designed with the advent of "Java 2". Sun has not announced any plans for a "Java 3".

In 1997, Sun approached the ISO/IEC JTC1 standards body and later the Ecma International to formalize Java, but it soon withdrew from the process. Java remains a proprietary de facto standard that is controlled through the Java Community Process. Sun makes most of its Java implementations available without charge, with revenue being generated by specialized products such as the Java Enterprise System. Sun distinguishes between its Software Development Kit (SDK) and Runtime Environment (JRE) which is a subset of the SDK, the primary distinction being that in the JRE the compiler is not present.

**Philosophy**

There were five primary goals in the creation of the Java language:  
  
1. It should use the [object-oriented programming](http://www.freejavaguide.com/object_oriented_programming.html) methodology.  
2. It should allow the same program to be executed on multiple operating systems.  
3. It should contain built-in support for using computer networks.  
4. It should be designed to execute code from remote sources securely.  
5. It should be easy to use by selecting what was considered the good parts of other object-oriented languages.  
  
To achieve the goals of networking support and remote code execution, Java programmers sometimes find it necessary to use extensions such as CORBA, Internet Communications Engine, or OSGi.  
  
**Object orientation**  
  
The first characteristic, object orientation ("OO"), refers to a method of programming and language design. Although there are many interpretations of OO, one primary distinguishing idea is to design software so that the various types of data it manipulates are combined together with their relevant operations. Thus, data and code are combined into entities called objects. An object can be thought of as a self-contained bundle of behavior (code) and state (data). The principle is to separate the things that change from the things that stay the same; often, a change to some data structure requires a corresponding change to the code that operates on that data, or vice versa. This separation into coherent objects provides a more stable foundation for a software system's design. The intent is to make large software projects easier to manage, thus improving quality and reducing the number of failed projects.  
  
Another primary goal of OO programming is to develop more generic objects so that software can become more reusable between projects. A generic "customer" object, for example, should have roughly the same basic set of behaviors between different software projects, especially when these projects overlap on some fundamental level as they often do in large organizations. In this sense, software objects can hopefully be seen more as pluggable components, helping the software industry build projects largely from existing and well-tested pieces, thus leading to a massive reduction in development times. Software reusability has met with mixed practical results, with two main difficulties: the design of truly generic objects is poorly understood, and a methodology for broad communication of reuse opportunities is lacking. Some open source communities want to help ease the reuse problem, by providing authors with ways to disseminate information about generally reusable objects and object libraries.  
  
**Platform independence**  
  
The second characteristic, platform independence, means that programs written in the Java language must run similarly on diverse hardware. One should be able to write a program once and run it anywhere.  
  
This is achieved by most Java compilers by compiling the Java language code "halfway" to bytecode (specifically Java bytecode)—simplified machine instructions specific to the Java platform. The code is then run on a virtual machine (VM), a program written in native code on the host hardware that interprets and executes generic Java bytecode. Further, standardized libraries are provided to allow access to features of the host machines (such as graphics, threading and networking) in unified ways. Note that, although there's an explicit compiling stage, at some point, the Java bytecode is interpreted or converted to native machine instructions by the JIT compiler.  
  
There are also implementations of Java compilers that compile to native object code, such as GCJ, removing the intermediate bytecode stage, but the output of these compilers can only be run on a single architecture.  
  
Sun's license for Java insists that all implementations be "compatible". This resulted in a legal dispute with Microsoft after Sun claimed that the Microsoft implementation did not support the RMI and JNI interfaces and had added platform-specific features of their own. In response, Microsoft no longer ships Java with Windows, and in recent versions of Windows, Internet Explorer cannot support [Java applets](http://www.freejavaguide.com/java_applet.html) without a third-party plug-in. However, Sun and others have made available Java run-time systems at no cost for those and other versions of Windows.  
  
The first implementations of the language used an interpreted virtual machine to achieve portability. These implementations produced programs that ran more slowly than programs compiled to native executables, for instance written in C or C++, so the language suffered a reputation for poor performance. More recent [JVM](http://www.freejavaguide.com/virtual_machine.html)implementations produce programs that run significantly faster than before, using multiple techniques.  
  
The first technique is to simply compile directly into native code like a more traditional compiler, skipping bytecodes entirely. This achieves good performance, but at the expense of portability. Another technique, known as just-in-time compilation (JIT), translates the Java bytecodes into native code at the time that the program is run which results in a program that executes faster than interpreted code but also incurs compilation overhead during execution. More sophisticated VMs use dynamic recompilation, in which the VM can analyze the behavior of the running program and selectively recompile and optimize critical parts of the program. Dynamic recompilation can achieve optimizations superior to static compilation because the dynamic compiler can base optimizations on knowledge about the runtime environment and the set of loaded classes. JIT compilation and dynamic recompilation allow Java programs to take advantage of the speed of native code without losing portability.  
  
Portability is a technically difficult goal to achieve, and Java's success at that goal has been mixed. Although it is indeed possible to write programs for the Java platform that behave consistently across many host platforms, the large number of available platforms with small errors or inconsistencies led some to parody Sun's "Write once, run anywhere" slogan as "Write once, debug everywhere".  
  
Platform-independent Java is however very successful with server-side applications, such as Web services, [servlets](http://www.freejavaguide.com/java_servlet.html), and Enterprise JavaBeans, as well as with Embedded systems based on OSGi, using Embedded Java environments.  
  
**Automatic garbage collection**  
  
One idea behind Java's automatic memory management model is that programmers should be spared the burden of having to perform manual memory management. In some languages the programmer allocates memory to create any object stored on the heap and is responsible for later manually deallocating that memory to delete any such objects. If a programmer forgets to deallocate memory or writes code that fails to do so in a timely fashion, a memory leak can occur: the program will consume a potentially arbitrarily large amount of memory. In addition, if a region of memory is deallocated twice, the program can become unstable and may crash. Finally, in non garbage collected environments, there is a certain degree of overhead and complexity of user-code to track and finalize allocations.  
  
In Java, this potential problem is avoided by automatic garbage collection. The programmer determines when objects are created, and the Java runtime is responsible for managing the object's lifecycle. The program or other objects can reference an object by holding a reference to it (which, from a low-level point of view, is its address on the heap). When no references to an object remain, the Java garbage collector automatically deletes the unreachable object, freeing memory and preventing a memory leak. Memory leaks may still occur if a programmer's code holds a reference to an object that is no longer needed—in other words, they can still occur but at higher conceptual levels.  
  
The use of garbage collection in a language can also affect programming paradigms. If, for example, the developer assumes that the cost of memory allocation/recollection is low, they may choose to more freely construct objects instead of pre-initializing, holding and reusing them. With the small cost of potential performance penalties (inner-loop construction of large/complex objects), this facilitates thread-isolation (no need to synchronize as different threads work on different object instances) and data-hiding. The use of transient immutable value-objects minimizes side-effect programming.  
  
Comparing Java and C++, it is possible in C++ to implement similar functionality (for example, a memory management model for specific classes can be designed in C++ to improve speed and lower memory fragmentation considerably), with the possible cost of extra development time and some application complexity. In Java, garbage collection is built-in and virtually invisible to the developer. That is, developers may have no notion of when garbage collection will take place as it may not necessarily correlate with any actions being explicitly performed by the code they write. Depending on intended application, this can be beneficial or disadvantageous: the programmer is freed from performing low-level tasks, but at the same time loses the option of writing lower level code.  
  
**Syntax**  
  
The syntax of Java is largely derived from C++. However, unlike C++, which combines the syntax for structured, generic, and [object-oriented programming](http://www.freejavaguide.com/object_oriented_programming.html), Java was built from the ground up to be virtually fully object-oriented: everything in Java is an object with the exceptions of atomic datatypes (ordinal and real numbers, boolean values, and characters) and everything in Java is written inside a class.  
  
**Applet**  
  
[Java applets](http://www.freejavaguide.com/java_applet.html) are programs that are embedded in other applications, typically in a Web page displayed in a Web browser.  
  
// Hello.java  
import java.applet.Applet;  
import java.awt.Graphics;  
  
public class Hello extends Applet {  
public void paint(Graphics gc) {  
gc.drawString("Hello, world!", 65, 95);  
}   
}  
  
This applet will simply draw the string "Hello, world!" in the rectangle within which the applet will run. This is a slightly better example of using Java's OO features in that the class explicitly extends the basic "Applet" class, that it overrides the "paint" method and that it uses import statements.  
  
<!-- Hello.html -->  
<html>  
<head>  
<title>Hello World Applet</title>  
</head>  
<body>  
<applet code="Hello" width="200" height="200">  
</applet>  
</body>  
</html>  
  
An applet is placed in an [HTML](http://www.freejavaguide.com/html.htm) document using the <applet> HTML element. The applet tag has three attributes set: code="Hello" specifies the name of the Applet class and width="200" height="200" sets the pixel width and height of the applet. (Applets may also be embedded in HTML using either the object or embed element, although support for these elements by Web browsers is inconsistent.  
  
**Servlet**  
  
[Java servlets](http://www.freejavaguide.com/java_servlet.html) are server-side Java EE components that generate responses to requests from clients.  
  
// Hello.java  
import java.io.\*;  
import javax.servlet.\*;  
  
public class Hello extends GenericServlet {  
public void service(ServletRequest request, ServletResponse response)  
throws ServletException, IOException  
{  
response.setContentType("text/html");  
PrintWriter pw = response.getWriter();  
pw.println("Hello, world!");  
pw.close();  
}  
}  
  
The import statements direct the Java compiler to include all of the public classes and interfaces from the java.io and javax.servlet packages in the compilation. The Hello class extends the GenericServlet class; the GenericServlet class provides the interface for the server to forward requests to the servlet and control the servlet's lifecycle.  
  
The Hello class overrides the service(ServletRequest, ServletResponse) method defined by the Servlet interface to provide the code for the service request handler. The service() method is passed a ServletRequest object that contains the request from the client and a ServletResponse object used to create the response returned to the client. The service() method declares that it throws the exceptions ServletException and IOException if a problem prevents it from responding to the request.  
  
The setContentType(String) method in the response object is called to set the MIME content type of the returned data to "text/html". The getWriter() method in the response returns a PrintWriter object that is used to write the data that is sent to the client. The println(String) method is called to write the "Hello, world!" string to the response and then the close() method is called to close the print writer, which causes the data that has been written to the stream to be returned to the client.  
  
**Swing application**  
  
[Swing](http://www.freejavaguide.com/java_swing.html) is the advanced graphical user interface library for the Java SE platform.  
  
// Hello.java  
import javax.swing.\*;  
  
public class Hello extends JFrame {  
Hello() {  
setDefaultCloseOperation(WindowConstants.DISPOSE\_ON\_CLOSE);  
add(new JLabel("Hello, world!"));  
pack();  
}  
  
public static void main(String[] args) {  
new Hello().setVisible(true);  
}  
}  
  
The import statement directs the Java compiler to include all of the public classes and interfaces from the javax.swing package in the compilation. The Hello class extends the JFrame class; the JFrame class implements a window with a title bar with a close control.  
  
The Hello() constructor initializes the frame by first calling the setDefaultCloseOperation(int) method inherited from JFrame to set the default operation when the close control on the title bar is selected to WindowConstants.DISPOSE\_ON\_CLOSE—this causes the JFrame to be disposed of when the frame is closed (as opposed to merely hidden), which allows the JVM to exit and the program to terminate. Next a new JLabel is created for the string "Hello, world!" and the add(Component) method inherited from the Container superclass is called to add the label to the frame. The pack() method inherited from the Window superclass is called to size the window and layout its contents.  
  
The main() method is called by the JVM when the program starts. It instantiates a new Hello frame and causes it to be displayed by calling the setVisible(boolean) method inherited from the Component superclass with the boolean parameter true. Note that once the frame is displayed, exiting the main method does not cause the program to terminate because the AWT event dispatching thread remains active until all of the Swing top-level windows have been disposed.  
  
**Look and feel**  
  
The default look and feel of GUI applications written in Java using the [Swing toolkit](http://www.freejavaguide.com/java_swing.html) is very different from native applications. It is possible to specify a different look and feel through the pluggable look and feel system of Swing. Clones of Windows, GTK and Motif are supplied by Sun. Apple also provides an Aqua look and feel for Mac OS X. Though prior implementations of these look and feels have been considered lacking, Swing in Java SE 6 addresses this problem by using more native widget drawing routines of the underlying platforms. Alternatively, third party toolkits such as wx4j or SWT may be used for increased integration with the native windowing system.  
  
**Lack of OO purity and facilities**  
  
Java's primitive types are not objects. Primitive types hold their values in the stack rather than being references to values. This was a conscious decision by Java's designers for performance reasons. Because of this, Java is not considered to be a pure object-oriented programming language. However, as of Java 5.0, autoboxing enables programmers to write as if primitive types are their wrapper classes, and freely interchange between them for improved flexibility. Java designers decided not to implement certain features present in other OO languages, including:  
  
\* multiple inheritance  
\* operator overloading  
\* class properties  
\* tuples   
  
**Java Runtime Environment**  
  
The Java Runtime Environment or JRE is the software required to run any application deployed on the Java Platform. End-users commonly use a JRE in software packages and Web browser plugins. Sun also distributes a superset of the JRE called the Java 2 SDK (more commonly known as the JDK), which includes development tools such as the Java compiler, Javadoc, and debugger.

Topic 2: Java features:

Features of Java

1. [Features of Java](http://www.javatpoint.com/features-of-java#features)
   1. [Simple](http://www.javatpoint.com/features-of-java#featuressimple)
   2. [Object-Oriented](http://www.javatpoint.com/features-of-java#featuresobject)
   3. [Platform Independent](http://www.javatpoint.com/features-of-java#featuresplateform)
   4. [secured](http://www.javatpoint.com/features-of-java#featuressecured)
   5. [Robust](http://www.javatpoint.com/features-of-java#featuresrobust)
   6. [Architecture Neutral](http://www.javatpoint.com/features-of-java#featuresarchitecture)
   7. [Portable](http://www.javatpoint.com/features-of-java#featuresportable)
   8. [High Performance](http://www.javatpoint.com/features-of-java#featureshigh)
   9. [Distributed](http://www.javatpoint.com/features-of-java#featuresdistributed)
   10. [Multi-threaded](http://www.javatpoint.com/features-of-java#featuresmulti)

There is given many features of java. They are also known as java buzzwords. The Java Features given below are simple and easy to understand.

1. Simple
2. Object-Oriented
3. Platform independent
4. Secured
5. Robust
6. Architecture neutral
7. Portable
8. Dynamic
9. Interpreted
10. High Performance
11. Multithreaded
12. Distributed

Simple

|  |
| --- |
| According to Sun, Java language is simple because: |
| syntax is based on C++ (so easier for programmers to learn it after C++). |
| removed many confusing and/or rarely-used features e.g., explicit pointers, operator overloading etc. |
| No need to remove unreferenced objects because there is Automatic Garbage Collection in java. |

Object-oriented

|  |
| --- |
| Object-oriented means we organize our software as a combination of different types of objects that incorporates both data and behaviour. |
| Object-oriented programming(OOPs) is a methodology that simplify software development and maintenance by providing some rules. |
| Basic concepts of OOPs are: |
| 1. Object 2. Class 3. Inheritance 4. Polymorphism 5. Abstraction 6. Encapsulation |

Platform Independent

|  |
| --- |
| A platform is the hardware or software environment in which a program runs. There are two types of platforms software-based and hardware-based. Java provides software-based platform. The Java platform differs from most other platforms in the sense that it's a software-based platform that runs on top of other hardware-based platforms.It has two components:   1. Runtime Environment 2. API(Application Programming Interface) |

|  |
| --- |
| java is platform independentJava code can be run on multiple platforms e.g.Windows,Linux,Sun Solaris,Mac/OS etc. Java code is compiled by the compiler and converted into bytecode.This bytecode is a platform independent code because it can be run on multiple platforms i.e. Write Once and Run Anywhere(WORA). |

Secured

|  |
| --- |
| Java is secured because: |
| * No explicit pointer * Programs run inside virtual machine sandbox. |

|  |  |
| --- | --- |
| how java is secured | how java is secured |

|  |
| --- |
| * **Classloader-** adds security by separating the package for the classes of the local file system from those that are imported from network sources. * **Bytecode Verifier-** checks the code fragments for illegal code that can violate access right to objects. * **Security Manager-** determines what resources a class can access such as reading and writing to the local disk. |
| These security are provided by java language. Some security can also be provided by application developer through SSL,JAAS,cryptography etc. |

Robust

|  |
| --- |
| Robust simply means strong. Java uses strong memory management. There are lack of pointers that avoids security problem. There is automatic garbage collection in java. There is exception handling and type checking mechanism in java. All these points makes java robust. |

Architecture-neutral

|  |
| --- |
| There is no implementation dependent features e.g. size of primitive types is set. |

Portable

|  |
| --- |
| We may carry the java bytecode to any platform. |

High-performance

|  |
| --- |
| Java is faster than traditional interpretation since byte code is "close" to native code still somewhat slower than a compiled language (e.g., C++) |

Distributed

|  |
| --- |
| We can create distributed applications in java. RMI and EJB are used for creating distributed applications. We may access files by calling the methods from any machine on the internet. |

Multi-threaded

A thread is like a separate program, executing concurrently. We can write Java programs that deal with many tasks at once by defining multiple threads. The main advantage of multi-threading is that it shares the same memory. Threads are important for multi-media, Web applications etc.

Topic 3:Compilation and Execution:

The computer age is here to stay. Households and businesses all over the world use computers in one way or another because computers help individuals and businesses perform a wide range of tasks with speed, accuracy, and efficiency. Computers can perform all kinds of tasks ranging from running an animated 3D graphics application with background sound to calculating the number of vacation days you have coming to handling the payroll for a Fortune 500 company.

When you want a computer to perform tasks, you write a program. A program is a sequence of instructions that define tasks for the computer to execute. This lesson explains how to write, compile, and run a simple program written in the Java language (Java program) that tells your computer to print a one-line string of text on the console.

But before you can write and compile programs, you need to understand what the Java platform is, and set your computer up to run the programs.

* [A Word About the Java Platform](http://www.oracle.com/technetwork/java/compile-136656.html#platform)
* [Setting Up Your Computer](http://www.oracle.com/technetwork/java/compile-136656.html#setup)
* [Writing a Program](http://www.oracle.com/technetwork/java/compile-136656.html#simple)
* [Compiling the Program](http://www.oracle.com/technetwork/java/compile-136656.html#comp)
* [Interpreting and Running the Program](http://www.oracle.com/technetwork/java/compile-136656.html#run)
* [Common Compiler and Interpreter Problems](http://www.oracle.com/technetwork/java/compile-136656.html#debug)
* [Code Comments](http://www.oracle.com/technetwork/java/compile-136656.html#comm)
* [API Documentation](http://www.oracle.com/technetwork/java/compile-136656.html#api)
* [More Information](http://www.oracle.com/technetwork/java/compile-136656.html#more)

|  |
| --- |
|  |

**A Word About the Java Platform**

The Java platform consists of the Java application programming interfaces (APIs) and the Java [1](http://www.oracle.com/technetwork/java/compile-136656.html#TJVM)virtual machine (JVM).

Java APIs are libraries of compiled code that you can use in your programs. They let you add ready-made and customizable functionality to save you programming time.

The simple program in this lesson uses a Java API to print a line of text to the console. The console printing capability is provided in the API ready for you to use; you supply the text to be printed.

Java programs are run (or interpreted) by another program called the Java VM. If you are familiar with Visual Basic or another interpreted language, this concept is probably familiar to you. Rather than running directly on the native operating system, the program is interpreted by the Java VM for the native operating system. This means that any computer system with the Java VM installed can run Java programs regardless of the computer system on which the applications were originally developed.

For example, a Java program developed on a Personal Computer (PC) with the Windows NT operating system should run equally well without modification on a Sun Ultra workstation with the Solaris operating system, and vice versa.

**Setting Up Your Computer**

Before you can write and run the simple Java program in this lesson, you need to install the Java platform on your computer system.

The Java platform is available free of charge from the [Java](http://www.oracle.com/technetwork/java/j2se-eol-137618.html) web site. You can choose between the Java® 2 Platform software for Windows 95/98/NT or for Solaris. The download page contains the information you need to install and configure the Java platform for writing and running Java programs.

|  |
| --- |
|  |

**Note:** Make sure you have the Java platform installed and configured for your system before you try to write and run the simple program presented next.

|  |
| --- |
|  |

**Writing a Program**

The easiest way to write a simple program is with a text editor. So, using the text editor of your choice, create a text file with the following text, and be sure to name the text fileExampleProgram.java. Java programs are case sensitive, so if you type the code in yourself, pay particular attention to the capitalization.

|  |
| --- |
| //A Very Simple Example  class ExampleProgram {  public static void main(String[] args){  System.out.println("I'm a Simple Program");  }  } |

Here is the [ExampleProgram.java](http://www.oracle.com/technetwork/java/compile-136656.html) source code file if you do not want to type the program text in yourself.

**Compiling the Program**

A program has to be converted to a form the Java VM can understand so any computer with a Java VM can interpret and run the program. Compiling a Java program means taking the programmer-readable text in your program file (also called source code) and converting it to bytecodes, which are platform-independent instructions for the Java VM.

The Java compiler is invoked at the command line on Unix and DOS shell operating systems as follows:

javac ExampleProgram.java

|  |
| --- |
|  |

**Note:** Part of the configuration process for setting up the Java platform is setting the class path. The class path can be set using either the -classpath option with the javac compiler command andjava interpreter command, or by setting the CLASSPATH environment variable. You need to set the class path to point to the directory where the ExampleProgram class is so the compiler and interpreter commands can find it. See [Java 2 SDK Tools](http://www.oracle.com/technetwork/java/j2se-eol-137618.html) for more information.

|  |
| --- |
|  |

**Interpreting and Running the Program**

Once your program successfully compiles into Java bytecodes, you can interpret and run applications on any Java VM, or interpret and run applets in any Web browser with a Java VM built in such as Netscape or Internet Explorer. Interpreting and running a Java program means invoking the Java VM byte code interpreter, which converts the Java byte codes to platform-dependent machine codes so your computer can understand and run the program.

The Java interpreter is invoked at the command line on Unix and DOS shell operating systems as follows:

java ExampleProgram

At the command line, you should see:

I'm a Simple Program

Here is how the entire sequence looks in a terminal window:

**Common Compiler and Interpreter Problems**

If you have trouble compiling or running the simple example in this lesson, refer to the [Common Compiler and Interpreter Problems](http://docs.oracle.com/javase/tutorial/getStarted/problems/index.html) lesson in The Java Tutorial for troubleshooting help.

**Code Comments**

Code comments are placed in source files to describe what is happening in the code to someone who might be reading the file, to comment-out lines of code to isolate the source of a problem for debugging purposes, or to generate API documentation. To these ends, the Java language supports three kinds of comments: double slashes, C-style, and doc comments.

**Double Slashes**

Double slashes ( //) are used in the C++ programming language, and tell the compiler to treat everything from the slashes to the end of the line as text.

|  |
| --- |
| //A Very Simple Example  class ExampleProgram {  public static void main(String[] args){  System.out.println("I'm a Simple Program");  }  } |

**C-Style Comments**

Instead of double slashes, you can use C-style comments ( /\* \*/) to enclose one or more lines of code to be treated as text.

|  |
| --- |
| /\* These are  C-style comments  \*/  class ExampleProgram {  public static void main(String[] args){  System.out.println("I'm a Simple Program");  }  } |

**Doc Comments**

To generate documentation for your program, use the doc comments ( /\*\* \*/) to enclose lines of text for the javadoc tool to find. The javadoc tool locates the doc comments embedded in source files and uses those comments to generate API documentation.

|  |
| --- |
| /\*\* This class displays a text string at  \* the console.  \*/  class ExampleProgram {  public static void main(String[] args){  System.out.println("I'm a Simple Program");  }  } |

HTML[javadoc Home Page](http://www.oracle.com/technetwork/java/javase/documentation/index-jsp-135444.html)javadoc**API Documentation**

The Java platform installation includes API Documentation, which describes the APIs available for you to use in your programs. The files are stored in a doc directory beneath the directory where you installed the platform. For example, if the platform is installed in /usr/local/java/jdk1.2, the API Documentation is in /usr/local/java/jdk1.2/doc/api. **More Information**

See [Java 2 SDK Tools](http://www.oracle.com/technetwork/java/j2se-eol-137618.html) for more information on setting the class path and using the javac, andjava commands.

See [Common Compiler and Interpreter Problems](http://docs.oracle.com/javase/tutorial/getStarted/problems/index.html) lesson in The Java Tutorial for troubleshooting help.

The [javadoc Home Page](http://www.oracle.com/technetwork/java/javase/documentation/index-jsp-135444.html) has more information on the javadoc command and its output.

You can also view the API Documentation for the Java 2 Platform on the [Java Sun](http://www.oracle.com/technetwork/java/j2se-eol-137618.html) site.

\_\_\_\_\_\_\_   
1 As used on this web site, the terms "Java virtual machine" or "JVM" mean a virtual machine for the Java platform.

Topic 4:JDK-JRE-JVM:

JVM (Java Virtual Machine)

1. [Java Virtual Machine](http://www.javatpoint.com/internal-details-of-jvm)
2. [Internal Architecture of JVM](http://www.javatpoint.com/internal-details-of-jvm#jvminternalarch)

JVM (Java Virtual Machine) is an abstract machine. It is a specification that provides runtime environment in which java bytecode can be executed.

JVMs are available for many hardware and software platforms (i.e.JVM is plateform dependent).

What is JVM?

It is:

1. **A specification** where working of Java Virtual Machine is specified. But implementation provider is independent to choose the algorithm. Its implementation has been provided by Sun and other companies.
2. **An implementation** Its implementation is known as JRE (Java Runtime Environment).
3. **Runtime Instance** Whenever you write java command on the command prompt to run the java class, and instance of JVM is created.

What it does?

The JVM performs following operation:

* Loads code
* Verifies code
* Executes code
* Provides runtime environment

JVM provides definitions for the:

* Memory area
* Class file format
* Register set
* Garbage-collected heap
* Fatal error reporting etc.

Internal Architecture of JVM

|  |
| --- |
| Let's understand the internal architecture of JVM. It contains classloader, memory area, execution engine etc. |

1) Classloader:

Classloader is a subsystem of JVM that is used to load class files.

2) Class(Method) Area:

Class(Method) Area stores per-class structures such as the runtime constant pool, field and method data, the code for methods.

3) Heap:

It is the runtime data area in which objects are allocated.

4) Stack:

|  |
| --- |
| Java Stack stores frames.It holds local variables and partial results, and plays a part in method invocation and return. |
| Each thread has a private JVM stack, created at the same time as thread. |
| A new frame is created each time a method is invoked. A frame is destroyed when its method invocation completes. |

5) Program Counter Register:

PC (program counter) register. It contains the address of the Java virtual machine instruction currently being executed.

6) Native Method Stack:

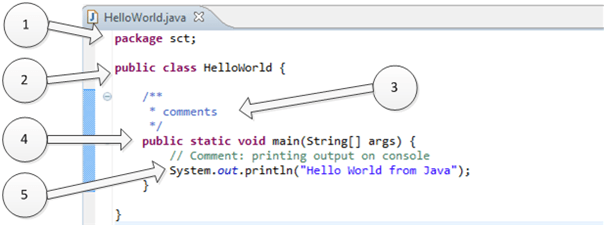
It contains all the native methods used in the application.

7) Execution Engine:

|  |
| --- |
| It contains: |
| **1) A virtual processor** |
| **2) Interpreter:**Read bytecode stream then execute the instructions. |
| **3) Just-In-Time(JIT) compiler:**It is used to improve the performance.JIT compiles parts of the byte code that have similar functionality at the same time, and hence reduces the amount of time needed for compilation.Here the term ?compiler? refers to a translator from the instruction set of a Java virtual machine (JVM) to the instruction set of a specific CPU. |

Toopic 5:Structure of Java Program

# Let’s use example of HelloWorld Java program to understand structure and features of class. This program is written on few lines, and its only task is to print “Hello World from Java” on the screen. Refer the following picture.



## 1. “package sct”:

It is package declaration statement. The package statement defines a name space in which classes are stored. Package is used to organize the classes based on functionality. If you omit the package statement, the class names are put into the default package, which has no name. Package statement cannot appear anywhere in program. It must be first line of your program or you can omit it.

## 2. “public class HelloWorld”:

This line has various aspects of java programming.

**a.** public: This is access modifier keyword which tells compiler access to class. Various values of access modifiers can be public, protected,private or default (no value).

**b.** class: This keyword used to declare class. Name of class (HelloWorld) followed by this keyword.

## 3. Comments section:

We can write comments in java in two ways.

**a.** Line comments: It start with two forward slashes (//) and continue to the end of the current line. Line comments do not require an ending symbol.

**b.** Block comments start with a forward slash and an asterisk (/\*) and end with an asterisk and a forward slash (\*/).Block comments can also extend across as many lines as needed.

## 4. “public static void main (String [ ] args)”:

Its method (Function) named main with string array as argument.

**a.** public : Access Modifier

**b.** static: static is reserved keyword which means that a method is accessible and usable even though no objects of the class exist.

**c.** void: This keyword declares nothing would be returned from method. Method can return any primitive or object.

**d.** Method content inside curly braces. { } asdfla;sd

## 5. System.out.println("Hello World from Java") :

**a.** System:It is name of Java utility class.

**b.** out:It is an object which belongs to System class.

**c.** println:It is utility method name which is used to send any String to console.

**d.** “Hello World from Java”:It is String literal set as argument to println method.

## More Information regarding Java Class:

* Java is an object-oriented language, which means that it has constructs to represent objects from the real world. Each Java program has at least one class that knows how to do certain things or how to represent some type of object. For example, the simplest class, HelloWorld,knows how to greet the world.
* Classes in Java may have methods (or functions) and fields (or attributes or properties).
* Let’s take example of Car object which has various properties like color, max speed etc. along with it has functions like run and stop. In Java world we will represent it like below:

[view plaincopy to clipboardprint?](http://www.w3resource.com/java-tutorial/java-program-structure.php)

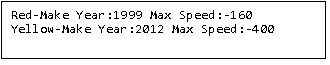
1. package sct;
2. **public** **class** Car {
3. **private** String color;
4. **private** int maxSpeed;
5. **public** String carInfo(){
6. **return** color + " Max Speed:-" + maxSpeed;
7. }
8. //This is constructor of Car Class
9. Car(String carColor, int speedLimit){
10. this.color = carColor;
11. this.maxSpeed =speedLimit;
12. }
13. }

* Lets make a class named CarTestwhich will instantiate the car class object and call carInfo method of it and see output.

[view plaincopy to clipboardprint?](http://www.w3resource.com/java-tutorial/java-program-structure.php)

1. package sct;
2. //This is car test class to instantiate and call Car objects.
3. **public** **class** CarTest {
4. **public** **static** void main(String[] args) {
5. Car maruti = **new** Car("Red", 160);
6. Car ferrari = **new** Car ("Yellow", 400);
7. System.out.println(maruti.carInfo());
8. System.out.println(ferrari.carInfo());
9. }
10. }

Output of above CarTest java class is as below. We can run CarTest java program because it has main method. Main method is starting point for any java program execution. Running a program means telling the Java VIrtual Machine (JVM) to "Load theclass, then start executing its main () method. Keep running 'til all thecode in main is finished."



## Common Programming guidelines:

* Java identifiers must start with with a letter, a currency character ($), or a connecting character such as the underscore ( \_ ). Identifiers cannot start with a number. After first character identifiers can contain any combination of letters,currency characters, connecting characters, or numbers. For example,
  + int variable1 = 10 ; //This is valid
  + int 4var =10 ; // this is invalid, identifier can’t start with digit.
* Identifiers, method names, class names are case-sensitive; var and Varare two different identifiers.
* You can’t use Java keywords as identifiers, Below table shows list of Java keywords.

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| abstract | boolean | break | byte | case | catch |
| char | class | const | continue | default | do |
| double | clse | extends | final | finally | float |
| for | goto | if | implements | import | instanceof |
| int | interface | long | native | new | package |
| private | protected | public | return | short | static |
| strictfp | super | switch | synchronized | this | throw |
| throws | transient | try | void | volatile | while |
| assert | enum |  |  |  |  |

* **Classes and interfaces:** The first letter should be capitalized, and if several words are linked together to form the name, the first letter of the inner words should be uppercase (a format that's sometimes called "camelCase").
* **Methods:** The first letter should be lowercase, and then normal camelCaserules should be used.For example:
  + getBalance
  + doCalculation
  + setCustomerName
* **Variables:** Like methods, the camelCase format should be used, starting witha lowercase letter. Sun recommends short, meaningful names, which sounds good to us. Some examples:
  + buttonWidth
  + accountBalance
  + empName
* **Constants:** Java constants are created by marking variables static and final. They should be named using uppercase letters with underscore characters as separators:
  + MIN\_HEIGHT
* There can be only one public class per source code file.
* Comments can appear at the beginning or end of any line in the source code file; they are independent of any of the positioning rules discussed here.
* If there is a public class in a file, the name of the file must match the nameof the public class. For example, a class declared as “public class Dog { }” must be in a source code file named Dog.java.
* To understand more about access modifiers applicable to classes, methods, variables in [access modifier tutorial.](http://www.w3resource.com/java-tutorial/java-class-methods-instance-variables.php)

We will understand more about constructors, access modifiers in coming tutorials. Source code of sample discussed attached here to run directly on your system.

## Summary

* Java program has first statement as package statement (if package declared).
* One Java file can have multiple classes declared in it but only one public class which should be same as file name.
* Java class can have instance variable such as methods, instance variable.

In next session we will discuss about Setup Classpath/ Environment variable and Compiling, running and debugging Java programs.

Chapter 2:Language Fundamentals

Topic 1:Primitive Data Types

Variables are nothing but reserved memory locations to store values. This means that when you create a variable you reserve some space in memory.

Based on the data type of a variable, the operating system allocates memory and decides what can be stored in the reserved memory. Therefore, by assigning different data types to variables, you can store integers, decimals, or characters in these variables.

There are two data types available in Java:

* Primitive Data Types
* Reference/Object Data Types

## Primitive Data Types:

There are eight primitive data types supported by Java. Primitive data types are predefined by the language and named by a keyword. Let us now look into detail about the eight primitive data types.

## byte:

* Byte data type is an 8-bit signed two's complement integer.
* Minimum value is -128 (-2^7)
* Maximum value is 127 (inclusive)(2^7 -1)
* Default value is 0
* Byte data type is used to save space in large arrays, mainly in place of integers, since a byte is four times smaller than an int.
* Example: byte a = 100 , byte b = -50

## short:

* Short data type is a 16-bit signed two's complement integer.
* Minimum value is -32,768 (-2^15)
* Maximum value is 32,767 (inclusive) (2^15 -1)
* Short data type can also be used to save memory as byte data type. A short is 2 times smaller than an int
* Default value is 0.
* Example: short s = 10000, short r = -20000

## int:

* Int data type is a 32-bit signed two's complement integer.
* Minimum value is - 2,147,483,648.(-2^31)
* Maximum value is 2,147,483,647(inclusive).(2^31 -1)
* Int is generally used as the default data type for integral values unless there is a concern about memory.
* The default value is 0.
* Example: int a = 100000, int b = -200000

## long:

* Long data type is a 64-bit signed two's complement integer.
* Minimum value is -9,223,372,036,854,775,808.(-2^63)
* Maximum value is 9,223,372,036,854,775,807 (inclusive). (2^63 -1)
* This type is used when a wider range than int is needed.
* Default value is 0L.
* Example: long a = 100000L, int b = -200000L

## float:

* Float data type is a single-precision 32-bit IEEE 754 floating point.
* Float is mainly used to save memory in large arrays of floating point numbers.
* Default value is 0.0f.
* Float data type is never used for precise values such as currency.
* Example: float f1 = 234.5f

## double:

* double data type is a double-precision 64-bit IEEE 754 floating point.
* This data type is generally used as the default data type for decimal values, generally the default choice.
* Double data type should never be used for precise values such as currency.
* Default value is 0.0d.
* Example: double d1 = 123.4

## boolean:

* boolean data type represents one bit of information.
* There are only two possible values: true and false.
* This data type is used for simple flags that track true/false conditions.
* Default value is false.
* Example: boolean one = true

## char:

* char data type is a single 16-bit Unicode character.
* Minimum value is '\u0000' (or 0).
* Maximum value is '\uffff' (or 65,535 inclusive).
* Char data type is used to store any character.
* Example: char letterA ='A'

## Reference Data Types:

* Reference variables are created using defined constructors of the classes. They are used to access objects. These variables are declared to be of a specific type that cannot be changed. For example, Employee, Puppy etc.
* Class objects, and various type of array variables come under reference data type.
* Default value of any reference variable is null.
* A reference variable can be used to refer to any object of the declared type or any compatible type.
* Example: Animal animal = new Animal("giraffe");

## Java Literals:

A literal is a source code representation of a fixed value. They are represented directly in the code without any computation.

Literals can be assigned to any primitive type variable. For example:

byte a = 68;

char a = 'A'

byte, int, long, and short can be expressed in decimal(base 10), hexadecimal(base 16) or octal(base 8) number systems as well.

Prefix 0 is used to indicate octal and prefix 0x indicates hexadecimal when using these number systems for literals. For example:

int decimal = 100;

int octal = 0144;

int hexa = 0x64;

String literals in Java are specified like they are in most other languages by enclosing a sequence of characters between a pair of double quotes. Examples of string literals are:

"Hello World"

"two\nlines"

"\"This is in quotes\""

String and char types of literals can contain any Unicode characters. For example:

char a = '\u0001';

String a = "\u0001";

Java language supports few special escape sequences for String and char literals as well. They are:

|  |  |
| --- | --- |
| **Notation** | **Character represented** |
| \n | Newline (0x0a) |
| \r | Carriage return (0x0d) |
| \f | Formfeed (0x0c) |
| \b | Backspace (0x08) |
| \s | Space (0x20) |
| \t | tab |
| \" | Double quote |
| \' | Single quote |
| \\ | backslash |
| \ddd | Octal character (ddd) |
| \uxxxx | Hexadecimal UNICODE character (xxxx) |

## What is Next?

This chapter explained you various data types, next topic explains different variable types and their usage. This will give you a good understanding about how they can be used in the Java classes, interfaces, etc.

Topic 1:Declaring Variables

A variable provides us with named storage that our programs can manipulate. Each variable in Java has a specific type, which determines the size and layout of the variable's memory; the range of values that can be stored within that memory; and the set of operations that can be applied to the variable.

You must declare all variables before they can be used. The basic form of a variable declaration is shown here:

data type variable [ = value][, variable [= value] ...] ;

Here *data type* is one of Java's datatypes and *variable* is the name of the variable. To declare more than one variable of the specified type, you can use a comma-separated list.

Following are valid examples of variable declaration and initialization in Java:

int a, b, c; // Declares three ints, a, b, and c.

int a = 10, b = 10; // Example of initialization

byte B = 22; // initializes a byte type variable B.

double pi = 3.14159; // declares and assigns a value of PI.

char a = 'a'; // the char variable a iis initialized with value 'a'

This chapter will explain various variable types available in Java Language. There are three kinds of variables in Java:

* Local variables
* Instance variables
* Class/static variables

## Local variables:

* Local variables are declared in methods, constructors, or blocks.
* Local variables are created when the method, constructor or block is entered and the variable will be destroyed once it exits the method, constructor or block.
* Access modifiers cannot be used for local variables.
* Local variables are visible only within the declared method, constructor or block.
* Local variables are implemented at stack level internally.
* There is no default value for local variables so local variables should be declared and an initial value should be assigned before the first use.

## Example:

Here, *age* is a local variable. This is defined inside *pupAge()* method and its scope is limited to this method only.

public class Test{

public void pupAge(){

int age = 0;

age = age + 7;

System.out.println("Puppy age is : " + age);

}

public static void main(String args[]){

Test test = new Test();

test.pupAge();

}

}

This would produce the following result:

Puppy age is: 7

## Example:

Following example uses *age* without initializing it, so it would give an error at the time of compilation.

public class Test{

public void pupAge(){

int age;

age = age + 7;

System.out.println("Puppy age is : " + age);

}

public static void main(String args[]){

Test test = new Test();

test.pupAge();

}

}

This would produce the following error while compiling it:

Test.java:4:variable number might not have been initialized

age = age + 7;

^

1 error

## Instance variables:

* Instance variables are declared in a class, but outside a method, constructor or any block.
* When a space is allocated for an object in the heap, a slot for each instance variable value is created.
* Instance variables are created when an object is created with the use of the keyword 'new' and destroyed when the object is destroyed.
* Instance variables hold values that must be referenced by more than one method, constructor or block, or essential parts of an object's state that must be present throughout the class.
* Instance variables can be declared in class level before or after use.
* Access modifiers can be given for instance variables.
* The instance variables are visible for all methods, constructors and block in the class. Normally, it is recommended to make these variables private (access level). However visibility for subclasses can be given for these variables with the use of access modifiers.
* Instance variables have default values. For numbers the default value is 0, for Booleans it is false and for object references it is null. Values can be assigned during the declaration or within the constructor.
* Instance variables can be accessed directly by calling the variable name inside the class. However within static methods and different class ( when instance variables are given accessibility) should be called using the fully qualified name .*ObjectReference.VariableName*.

## Example:

import java.io.\*;

public class Employee{

// this instance variable is visible for any child class.

public String name;

// salary variable is visible in Employee class only.

private double salary;

// The name variable is assigned in the constructor.

public Employee (String empName){

name = empName;

}

// The salary variable is assigned a value.

public void setSalary(double empSal){

salary = empSal;

}

// This method prints the employee details.

public void printEmp(){

System.out.println("name : " + name );

System.out.println("salary :" + salary);

}

public static void main(String args[]){

Employee empOne = new Employee("Ransika");

empOne.setSalary(1000);

empOne.printEmp();

}

}

This would produce the following result:

name : Ransika

salary :1000.0

## Class/static variables:

* Class variables also known as static variables are declared with the *static* keyword in a class, but outside a method, constructor or a block.
* There would only be one copy of each class variable per class, regardless of how many objects are created from it.
* Static variables are rarely used other than being declared as constants. Constants are variables that are declared as public/private, final and static. Constant variables never change from their initial value.
* Static variables are stored in static memory. It is rare to use static variables other than declared final and used as either public or private constants.
* Static variables are created when the program starts and destroyed when the program stops.
* Visibility is similar to instance variables. However, most static variables are declared public since they must be available for users of the class.
* Default values are same as instance variables. For numbers, the default value is 0; for Booleans, it is false; and for object references, it is null. Values can be assigned during the declaration or within the constructor. Additionally values can be assigned in special static initializer blocks.
* Static variables can be accessed by calling with the class name .*ClassName.VariableName*.
* When declaring class variables as public static final, then variables names (constants) are all in upper case. If the static variables are not public and final the naming syntax is the same as instance and local variables.

## Example:

import java.io.\*;

public class Employee{

// salary variable is a private static variable

private static double salary;

// DEPARTMENT is a constant

public static final String DEPARTMENT = "Development ";

public static void main(String args[]){

salary = 1000;

System.out.println(DEPARTMENT+"average salary:"+salary);

}

}

This would produce the following result:

Development average salary:1000

**Note:** If the variables are access from an outside class the constant should be accessed as Employee.DEPARTMENT

## What is Next?

You already have used access modifiers ( public & private ) in this chapter. The next chapter will explain you Access Modifiers and Non Access Modifiers in detail.

Topic 1:Literals

**By literal we mean any number, text, or other information that represents a value. This means what you type is what you get. We will use literals in addition to variables in Java statement.**

Java Literals

**[http://www.roseindia.net/images/bt_home.gif](http://www.roseindia.net/java/language/index.shtml)**  **[http://www.roseindia.net/images/next.gif](http://www.roseindia.net/java/language/introduction-to-java-arrays.shtml)**

By literal we mean any number, text, or other information that represents a value. This means what you type is what you get. We will use literals in addition to variables in Java statement. While writing a source code as a character sequence, we can specify any value as a literal such as an integer. This character sequence will specify the syntax based on the value's type. This will give a literal as a result. For instance

int month  = 10;

In the above statement the literal is an integer value i.e 10. The literal is 10 because it directly represents the integer value.

In Java programming language there are some special type of literals that represent numbers, characters, strings and boolean values. Lets have a closer look on each of the following.

**Integer Literals**Integer literals is a sequence of digits and a suffix as L. To represent the type as long integer we use L as a suffix. We can specify the integers either in decimal, hexadecimal or octal format. To indicate a **decimal format**put the left most digit as nonzero. Similarly put the characters as *ox* to the left of at least one hexadecimal digit to indicate **hexadecimal format**. Also we can indicate the **octal format**by a zero digit followed by the digits 0 to 7. Lets tweak the table below.

|  |  |
| --- | --- |
| 659L | Decimal integer literal of type long integer |
| 0x4a | Hexadecimal integer literal of type integer |
| 057L | Octal integer literal of type long integer |

**Character Literals**We can specify a character literal as a single printable character in a pair of single quote characters such as 'a', '#', and '3'. You must be knowing about the ASCII character set. The ASCII character set includes 128 characters including letters, numerals, punctuations etc. There are few character literals which are not readily printable through a keyboard. The table below shows the codes that can  represent these special characters. The letter d such as in the octal, hex etc represents a number.

|  |  |
| --- | --- |
| **Escape** | **Meaning** |
| \n | New line |
| \t | Tab |
| \b | Backspace |
| \r | Carriage return |
| \f | Formfeed |
| \\ | Backslash |
| \' | Single quotation mark |
| \" | Double quotation mark |
| \d | Octal |
| \xd | Hexadecimal |
| \ud | Unicode character |

It is very interesting to know that if we want to specify a single quote, a backslash, or a nonprintable character as a character literal use an **escape sequence.** An escape sequence uses a special syntax to represents a character. The syntax begins with a single backslash character.   
Lets see the table below in which the character literals use Unicode escape sequence to represent printable and nonprintable characters both.

|  |  |
| --- | --- |
| 'u0041' | Capital letter A |
| '\u0030' | Digit 0 |
| '\u0022' | Double quote " |
| '\u003b' | Punctuation ; |
| '\u0020' | Space |
| '\u0009' | Horizontal Tab |

**Boolean Literals**The values true and false are also treated as literals in Java programming. When we assign a value to a boolean variable, we can only use these two values. Unlike C, we can't presume that the value of 1 is equivalent to true and 0 is equivalent to false in Java. We have to use the values true and false to represent a Boolean value. Like   
boolean chosen = true;  
Remember that the **literal true** is not represented by the quotation marks around it. The Java compiler will take it as a string of characters, if its in quotation marks.  
  
**Floating-point literals**Floating-point numbers are like real numbers in mathematics, for example, 4.13179, -0.000001. Java has two kinds of floating-point numbers: float and double. The default type when you write a floating-point literal is double.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Type | Size | | Range | Precision |
| name | bytes | bits | approximate | in decimal digits |
| float | 4 | 32 | +/- 3.4 \* 1038 | 6-7 |
| double | 8 | 64 | +/- 1.8 \* 10308 | 15 |

A floating-point literal can be denoted as a decimal point, a fraction part, an exponent (represented by E or e) and as an integer. We also add a suffix to the floating point literal as D, d, F or f.  The type of a floating-point literal defaults to double-precision floating-point.The following floating-point literals represent double-precision floating-point and floating-point values.

|  |  |
| --- | --- |
| 6.5E+32 (or 6.5E32) | Double-precision floating-point literal |
| 7D | Double-precision floating-point literal |
| .01f | Floating-point literal |

**String Literals**The string of characters is represented as String literals in Java. In Java a string is not a basic data type, rather it is an object. These strings are not stored in arrays as in C language. There are few methods provided in Java to combine strings, modify strings and to know whether to strings have the same value.  
We represent string literals as  
String myString = "How are you?";  
The above example shows how to represent a string. It consists of a series of characters inside double quotation marks.

Lets see some more examples of string literals:  
  
""    // the empty string  
"\""   // a string containing "  
"This is a string"   // a string containing 16 characters  
"This is a " +   // actually a string-valued constant expression,  
"two-line string"   // formed from two string literals

Strings can include the character escape codes as well, as shown here:  
String example = "Your Name, \"Sumit\"";  
System.out.println("Thankingyou,\nRichards\n");

**Null Literals**  
The final literal that we can use in Java programming is a Null literal. We specify the Null literal in the source code as 'null'. To reduce the number of references to an object, use null literal. The type of the null literal is always null. We typically assign null literals to object reference variables. For instance  
  
s = null;  
  
An this example an object is referenced by s. We reduce the number of references to an object by assigning null to s. Now, as in this example the object is no longer referenced so it will be available for the garbage collection i.e. the compiler will destroy it and the free memory will be allocated to the other object. Well, we will later learn about garbage collection.

Topic 1:Escape Sequences

|  |  |
| --- | --- |
| **Escape**  **Sequence** | **Character** |
| \n | newline |
| \t | tab |
| \b | backspace |
| \f | form feed |
| \r | return |
| \" | "   (double quote) |
| \' | '    (single quote) |
| \\ | \    (back slash) |
| \uDDDD | character from the Unicode character set (DDDD is four hex digits) |

Topic 1:Comments

## Comments in Java Code

The bold characters in the following listing are comments.

**/\*\***

**\* The HelloWorldApp class implements an application that**

**\* simply displays "Hello World!" to the standard output.**

**\*/**

class HelloWorldApp {

public static void main(String[] args) {

System.out.println("Hello World!"); **//Display the string.**

}

}

The Java language supports three kinds of comments:

/\* text \*/

The compiler ignores everything from /\* to \*/.

/\*\* documentation \*/

This indicates a documentation comment (doc comment, for short). The compiler ignores this kind of comment, just like it ignores comments that use /\* and \*/. The JDK javadoctool uses doc comments when preparing automatically generated documentation. For more information on javadoc, see the [Java tool documentation](http://java.sun.com/products/JDK/tools/index.html).

// text

The compiler ignores everything from // to the end of the line.

Topic 1:Identifiers and Keywords

## *Keywords*

Keywords have special meaning to the Java compiler. They help in indentifying a data type name or program construct name.  
  
Java Programming Language Keywords :

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| abstract | continue | for | new | switch |
| assert | default | goto | package | synchronized |
| boolean | do | if | private | this |
| break | double | implements | protected | throw |
| byte | else | import | public | throws |
| case | enum | instanceof | return | transient |
| catch | extends | int | short | try |
| char | final | interface | static | void |
| class | finally | long | strictfp | volatile |
| const | float | native | super | while |

## *Literals*

A constant value in a program is denoted by a *literal.* Literals represent numerical (integer or floating-point), character, boolean or string values.  
  
**Example of literals:**  
  
Integer literals:  
33 0 -9  
Floating-point literals:  
.3 0.3 3.14  
Character literals:  
'(' 'R' 'r' '{'  
Boolean literals:(predefined values)  
true false  
  
String literals:  
"language" "0.2" "r" ""

## *Identifiers*

In Java programming language, an *identifier* is a name given to a variable, class or method. Identifiers start with a letter, underscore(\_) or dollar sign ($). The following characters can be digits. Identifiers are case sensitive and have no maximum length.

e.g.

* userName
* user\_name
* \_sys\_var1
* $change

Chapter 3:Operators and Assignments

Topic 1:Precedence and Associativity

Java provides a rich set of operators to manipulate variables. We can divide all the Java operators into the following groups:

* Arithmetic Operators
* Relational Operators
* Bitwise Operators
* Logical Operators
* Assignment Operators
* Misc Operators

## The Arithmetic Operators:

Arithmetic operators are used in mathematical expressions in the same way that they are used in algebra. The following table lists the arithmetic operators:

Assume integer variable A holds 10 and variable B holds 20, then:

[**Show Examples**](http://www.tutorialspoint.com/java/java_arithmatic_operators_examples.htm)

|  |  |  |
| --- | --- | --- |
| **Operator** | **Description** | **Example** |
| + | Addition - Adds values on either side of the operator | A + B will give 30 |
| - | Subtraction - Subtracts right hand operand from left hand operand | A - B will give -10 |
| \* | Multiplication - Multiplies values on either side of the operator | A \* B will give 200 |
| / | Division - Divides left hand operand by right hand operand | B / A will give 2 |
| % | Modulus - Divides left hand operand by right hand operand and returns remainder | B % A will give 0 |
| ++ | Increment - Increases the value of operand by 1 | B++ gives 21 |
| -- | Decrement - Decreases the value of operand by 1 | B-- gives 19 |

## The Relational Operators:

There are following relational operators supported by Java language

Assume variable A holds 10 and variable B holds 20, then:

[**Show Examples**](http://www.tutorialspoint.com/java/java_relational_operators_examples.htm)

|  |  |  |
| --- | --- | --- |
| **Operator** | **Description** | **Example** |
| == | Checks if the values of two operands are equal or not, if yes then condition becomes true. | (A == B) is not true. |
| != | Checks if the values of two operands are equal or not, if values are not equal then condition becomes true. | (A != B) is true. |
| > | Checks if the value of left operand is greater than the value of right operand, if yes then condition becomes true. | (A > B) is not true. |
| < | Checks if the value of left operand is less than the value of right operand, if yes then condition becomes true. | (A < B) is true. |
| >= | Checks if the value of left operand is greater than or equal to the value of right operand, if yes then condition becomes true. | (A >= B) is not true. |
| <= | Checks if the value of left operand is less than or equal to the value of right operand, if yes then condition becomes true. | (A <= B) is true. |

## The Bitwise Operators:

Java defines several bitwise operators, which can be applied to the integer types, long, int, short, char, and byte.

Bitwise operator works on bits and performs bit-by-bit operation. Assume if a = 60; and b = 13; now in binary format they will be as follows:

a = 0011 1100

b = 0000 1101

-----------------

a&b = 0000 1100

a|b = 0011 1101

a^b = 0011 0001

~a  = 1100 0011

The following table lists the bitwise operators:

Assume integer variable A holds 60 and variable B holds 13 then:

[**Show Examples**](http://www.tutorialspoint.com/java/java_bitwise_operators_examples.htm)

|  |  |  |
| --- | --- | --- |
| **Operator** | **Description** | **Example** |
| & | Binary AND Operator copies a bit to the result if it exists in both operands. | (A & B) will give 12 which is 0000 1100 |
| | | Binary OR Operator copies a bit if it exists in either operand. | (A | B) will give 61 which is 0011 1101 |
| ^ | Binary XOR Operator copies the bit if it is set in one operand but not both. | (A ^ B) will give 49 which is 0011 0001 |
| ~ | Binary Ones Complement Operator is unary and has the effect of 'flipping' bits. | (~A ) will give -61 which is 1100 0011 in 2's complement form due to a signed binary number. |
| << | Binary Left Shift Operator. The left operands value is moved left by the number of bits specified by the right operan | A << 2 will give 240 which is 1111 0000 |
| >> | Binary Right Shift Operator. The left operands value is moved right by the number of bits specified by the right operand. | A >> 2 will give 15 which is 1111 |
| >>> | Shift right zero fill operator. The left operands value is moved right by the number of bits specified by the right operand and shifted values are filled up with zeros. | A >>>2 will give 15 which is 0000 1111 |

## The Logical Operators:

The following table lists the logical operators:

Assume Boolean variables A holds true and variable B holds false, then:

[**Show Examples**](http://www.tutorialspoint.com/java/java_logical_operators_examples.htm)

|  |  |  |
| --- | --- | --- |
| **Operator** | **Description** | **Example** |
| && | Called Logical AND operator. If both the operands are non-zero, then the condition becomes true. | (A && B) is false. |
| || | Called Logical OR Operator. If any of the two operands are non-zero, then the condition becomes true. | (A || B) is true. |
| ! | Called Logical NOT Operator. Use to reverses the logical state of its operand. If a condition is true then Logical NOT operator will make false. | !(A && B) is true. |

## The Assignment Operators:

There are following assignment operators supported by Java language:

[**Show Examples**](http://www.tutorialspoint.com/java/java_assignment_operators_examples.htm)

|  |  |  |
| --- | --- | --- |
| **Operator** | **Description** | **Example** |
| = | Simple assignment operator, Assigns values from right side operands to left side operand | C = A + B will assign value of A + B into C |
| += | Add AND assignment operator, It adds right operand to the left operand and assign the result to left operand | C += A is equivalent to C = C + A |
| -= | Subtract AND assignment operator, It subtracts right operand from the left operand and assign the result to left operand | C -= A is equivalent to C = C - A |
| \*= | Multiply AND assignment operator, It multiplies right operand with the left operand and assign the result to left operand | C \*= A is equivalent to C = C \* A |
| /= | Divide AND assignment operator, It divides left operand with the right operand and assign the result to left operand | C /= A is equivalent to C = C / A |
| %= | Modulus AND assignment operator, It takes modulus using two operands and assign the result to left operand | C %= A is equivalent to C = C % A |
| <<= | Left shift AND assignment operator | C <<= 2 is same as C = C << 2 |
| >>= | Right shift AND assignment operator | C >>= 2 is same as C = C >> 2 |
| &= | Bitwise AND assignment operator | C &= 2 is same as C = C & 2 |
| ^= | bitwise exclusive OR and assignment operator | C ^= 2 is same as C = C ^ 2 |
| |= | bitwise inclusive OR and assignment operator | C |= 2 is same as C = C | 2 |

## Misc Operators

There are few other operators supported by Java Language.

## Conditional Operator ( ? : ):

Conditional operator is also known as the ternary operator. This operator consists of three operands and is used to evaluate Boolean expressions. The goal of the operator is to decide which value should be assigned to the variable. The operator is written as:

variable x = (expression) ? value if true : value if false

Following is the example:

public class Test {

public static void main(String args[]){

int a , b;

a = 10;

b = (a == 1) ? 20: 30;

System.out.println( "Value of b is : " + b );

b = (a == 10) ? 20: 30;

System.out.println( "Value of b is : " + b );

}

}

This would produce the following result:

Value of b is : 30

Value of b is : 20

## instanceof Operator:

This operator is used only for object reference variables. The operator checks whether the object is of a particular type(class type or interface type). instanceof operator is wriiten as:

( Object reference variable ) instanceof (class/interface type)

If the object referred by the variable on the left side of the operator passes the IS-A check for the class/interface type on the right side, then the result will be true. Following is the example:

public class Test {

public static void main(String args[]){

String name = "James";

// following will return true since name is type of String

boolean result = name instanceof String;

System.out.println( result );

}

}

This would produce the following result:

true

This operator will still return true if the object being compared is the assignment compatible with the type on the right. Following is one more example:

class Vehicle {}

public class Car extends Vehicle {

public static void main(String args[]){

Vehicle a = new Car();

boolean result = a instanceof Car;

System.out.println( result );

}

}

This would produce the following result:

true

## Precedence of Java Operators:

Operator precedence determines the grouping of terms in an expression. This affects how an expression is evaluated. Certain operators have higher precedence than others; for example, the multiplication operator has higher precedence than the addition operator:

For example, x = 7 + 3 \* 2; here x is assigned 13, not 20 because operator \* has higher precedence than +, so it first gets multiplied with 3\*2 and then adds into 7.

Here, operators with the highest precedence appear at the top of the table, those with the lowest appear at the bottom. Within an expression, higher precedence operators will be evaluated first.

|  |  |  |
| --- | --- | --- |
| **Category** | **Operator** | **Associativity** |
| Postfix | () [] . (dot operator) | Left toright |
| Unary | ++ - - ! ~ | Right to left |
| Multiplicative | \* / % | Left to right |
| Additive | + - | Left to right |
| Shift | >> >>> << | Left to right |
| Relational | > >= < <= | Left to right |
| Equality | == != | Left to right |
| Bitwise AND | & | Left to right |
| Bitwise XOR | ^ | Left to right |
| Bitwise OR | | | Left to right |
| Logical AND | && | Left to right |
| Logical OR | || | Left to right |
| Conditional | ?: | Right to left |
| Assignment | = += -= \*= /= %= >>= <<= &= ^= |= | Right to left |
| Comma | , | Left to right |

## What is Next?

Next chapter would explain about loop control in Java programming. The chapter will describe various types of loops and how these loops can be used in Java program development and for what purposes they are being used.

Topic 1:Conversion

## JAVA OPERATORS

They are used to manipulate primitive data types. **Java operators** can be classified as unary, binary, or ternary—meaning taking one, two, or three arguments, respectively. A unary operator may appear  
before (prefix) its argument or after (postfix) its argument. A binary or ternary operator appears between its arguments.

Operators in java fall into 8 different categories:

Java operators fall into eight different categories: **assignment, arithmetic, relational, logical, bitwise, compound assignment, conditional, and type**.

Assignment Operators =

 Arithmetic Operators - + \* / % ++ --

 Relational Operators > < >= <= == !=

 Logical Operators && || & | ! ^

 Bit wise Operator & | ^ >> >>>

 Compound Assignment Operators += -= \*= /= %=

  <<= >>= >>>= Conditional Operator ?:

Java has eight different operator types: assignment, arithmetic, relational, logical, bitwise, compound assignment, conditional, and type.

## ASSIGNMENT OPERATORS

The java assignment operator statement has the following syntax:

variable> = <*expression*>

If the value already exists in the variable it is overwritten by the assignment operator (=).

public class AssignmentOperatorsDemo {

public AssignmentOperatorsDemo() {

// Assigning Primitive Values

int j, k;

j = 10; // j gets the value 10.

j = 5; // j gets the value 5. Previous value is overwritten.

k = j; // k gets the value 5.

System.out.println("j is : " + j);

System.out.println("k is : " + k);

// Assigning References

Integer i1 = new Integer("1");

Integer i2 = new Integer("2");

System.out.println("i1 is : " + i1);

System.out.println("i2 is : " + i2);

i1 = i2;

System.out.println("i1 is : " + i1);

System.out.println("i2 is : " + i2);

// Multiple Assignments

k = j = 10; // (k = (j = 10))

System.out.println("j is : " + j);

System.out.println("k is : " + k);

}

public static void main(String args[]) {

new AssignmentOperatorsDemo();

}

}

[**Download**](http://www.wideskills.com/sites/default/files/Zip_Files/Assignment-Operators.zip) AssignmentOperatorsDemoSource code

## ARITHMETIC OPERATORS

Java provides eight Arithmetic operators. They are for addition, subtraction, multiplication, division, modulo (or remainder), increment (or add 1), decrement (or subtract 1), and negation. An example program is shown below that demonstrates the different arithmetic operators in java.

The binary operator + is overloaded in the sense that the operation performed is determined by the type of the operands. When one of the operands is a String object, the other operand is implicitly converted to its string representation and string concatenation is performed.

*String message = 100 + “Messages”; //”100 Messages”*

public class ArithmeticOperatorsDemo {

public ArithmeticOperatorsDemo() {

int x, y = 10, z = 5;

x = y + z;

System.out.println("+ operator resulted in " + x);

x = y - z;

System.out.println("- operator resulted in " + x);

x = y \* z;

System.out.println("\* operator resulted in " + x);

x = y / z;

System.out.println("/ operator resulted in " + x);

x = y % z;

System.out.println("% operator resulted in " + x);

x = y++;

System.out.println("Postfix ++ operator resulted in " + x);

x = ++z;

System.out.println("Prefix ++ operator resulted in " + x);

x = -y;

System.out.println("Unary operator resulted in " + x);

// Some examples of special Cases

int tooBig = Integer.MAX\_VALUE + 1; // -2147483648 which is

// Integer.MIN\_VALUE.

int tooSmall = Integer.MIN\_VALUE - 1; // 2147483647 which is

// Integer.MAX\_VALUE.

System.out.println("tooBig becomes " + tooBig);

System.out.println("tooSmall becomes " + tooSmall);

System.out.println(4.0 / 0.0); // Prints: Infinity

System.out.println(-4.0 / 0.0); // Prints: -Infinity

System.out.println(0.0 / 0.0); // Prints: NaN

double d1 = 12 / 8; // result: 1 by integer division.

  // d1 gets the value 1.0.

double d2 = 12.0F / 8; // result: 1.5

System.out.println("d1 is " + d1);

System.out.println("d2 iss " + d2);

}

public static void main(String args[]) {

new ArithmeticOperatorsDemo();

}

}

[**Download**](http://www.wideskills.com/sites/default/files/Zip_Files/Arithmetic-Operators.zip) ArithmeticOperatorsDemo Source code

## RELATIONAL OPERATORS

Relational operators in Java are used to compare 2 or more objects. Java provides six relational operators:

greater than (>), less than (<), greater than or equal (>=), less than or equal (<=), equal (==), and not equal (!=).

All relational operators are binary operators, and their operands are numeric expressions.

Binary numeric promotion is applied to the operands of these operators. The evaluation results in a boolean value. Relational operators have precedence lower than arithmetic operators, but higher than that of the assignment operators. An example program is shown below that demonstrates the different relational operators in java.

public class RelationalOperatorsDemo {

public RelationalOperatorsDemo( ) {

int x = 10, y = 5;

System.out.println("x &gt; y : "+(x &gt; y));

System.out.println("x &lt; y : "+(x &lt; y));

System.out.println("x &gt;= y : "+(x &gt;= y));

System.out.println("x &lt;= y : "+(x &lt;= y));

System.out.println("x == y : "+(x == y));

System.out.println("x != y : "+(x != y));

}

public static void main(String args[]){

new RelationalOperatorsDemo();

}

}

[**Download**](http://www.wideskills.com/sites/default/files/Zip_Files/Relational-Operators.zip) RelationalOperatorsDemo Source code

## LOGICAL OPERATORS

Logical operators return a true or false value based on the state of the Variables. There are six logical, or boolean, operators. They are AND, conditional AND, OR, conditional OR, exclusive OR, and NOT. Each argument to a logical operator must be a boolean data type, and the result is always a boolean data type. An example program is shown below that demonstrates the different Logical operators in java.

public class LogicalOperatorsDemo {

public LogicalOperatorsDemo() {

boolean x = true;

boolean y = false;

System.out.println("x &amp; y : " + (x &amp; y));

System.out.println("x &amp;&amp; y : " + (x &amp;&amp; y));

System.out.println("x | y : " + (x | y));

System.out.println("x || y: " + (x || y));

System.out.println("x ^ y : " + (x ^ y));

System.out.println("!x : " + (!x));

}

public static void main(String args[]) {

new LogicalOperatorsDemo();

}

}

[**Download**](http://www.wideskills.com/sites/default/files/Zip_Files/Logical-Operators.zip) LogicalOperatorsDemo Source code

Given that x and y represent boolean expressions, the boolean logical operators are defined in the Table below.

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **x** | **y** | **!x** | **x & y**  **x && y** | **x | y**  **x || y** | **x ^ y** |
| true | true | false | true | true | false |
| true | false | false | false | true | true |
| false | true | true | false | true | true |
| false | false | true | false | false | false |

### BITWISE OPERATORS

Java provides Bit wise operators to manipulate the contents of variables at the bit level.

These variables must be of numeric data type ( char, short, int, or long). Java provides seven bitwise

operators. They are AND, OR, Exclusive-OR, Complement, Left-shift, Signed Right-shift, and Unsigned Right-shift. An example program is shown below that demonstrates the different Bit wise operators in java.

public class BitwiseOperatorsDemo {

public BitwiseOperatorsDemo() {

int x = 0xFAEF; //1 1 1 1 1 0 1 0 1 1 1 0 1 1 1 1

int y = 0xF8E9; //1 1 1 1 1 0 0 0 1 1 1 0 1 0 0 1

int z; System.out.println("x &amp; y : " + (x &amp; y));

System.out.println("x | y : " + (x | y));

System.out.println("x ^ y : " + (x ^ y));

System.out.println("~x : " + (~x));

System.out.println("x &lt;&lt; y : " + (x &lt;&lt; y));

System.out.println("x &gt;&gt; y : " + (x &gt;&gt; y));

System.out.println("x &gt;&gt;&gt; y : " + (x &gt;&gt;&gt; y));

//There is no unsigned left shift operator

}

public static void main(String args[])

new BitwiseOperatorsDemo();

}

[**Download**](http://www.wideskills.com/sites/default/files/Zip_Files/Bitwise-Operators.zip) BitwiseOperatorsDemo Source code

The result of applying bitwise operators between two corresponding bits in the operands is shown in the Table below.

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **A** | **B** | **~A** | **A & B** | **A | B** | **A ^ B** |
| **1** | **1** | 0 | 1 | 1 | 0 |
| **1** | **0** | 0 | 0 | 1 | 1 |
| **0** | **1** | 1 | 0 | 1 | 1 |
| **0** | **0** | 1 | 0 | 0 | 0 |

**Output**

/\*

\* The below program demonstrates bitwise operators keeping in mind operator precedence

\* Operator Precedence starting with the highest is -&gt; |, ^, &amp;

\*/

public class BitwisePrecedenceEx {

public static void main(String[] args) {

int a = 1 | 2 ^ 3 &amp; 5;

int b = ((1 | 2) ^ 3) &amp; 5;

int c = 1 | (2 ^ (3 &amp; 5));

System.out.print(a + "," + b + "," + c);

}

}

[**Download**](http://www.wideskills.com/sites/default/files/Zip_Files/Bitwise-Operators-2.zip) BitwiseOperatorsDemo2 Source code

### COMPOUND OPERATORS

compound operators perform shortcuts in common programming operations. Java has eleven compound assignment operators.

Syntax:

 = argument2.

The above statement is the same as, argument1 = argument1 operator argument2. An example program is shown below that demonstrates the different Compound operators in java.

public class CompoundOperatorsDemo {

public CompoundOperatorsDemo() {

int x = 0, y = 5;

x += 3;

System.out.println("x : " + x);

y \*= x;

System.out.println("y : " + y);

/\*Similarly other operators can be applied as shortcuts. Other

compound assignment operators include boolean logical

, bitwiseand shift operators\*/

}

public static void main(String args[]) {

new CompoundOperatorsDemo();

}

}

[**Download**](http://www.wideskills.com/sites/default/files/Zip_Files/Compound-Operators.zip) CompoundOperatorsDemo Source code

### CONDITIONAL OPERATORS

The Conditional operator is the only ternary (operator takes three arguments) operator in Java. The operator evaluates the first argument and, if true, evaluates the second argument. If the first argument evaluates to false, then the third argument is evaluated. The conditional operator is the expression equivalent of the if-else statement. The conditional expression can be nested and the conditional operator associates from right to left: **(a?b?c?d:e:f:g) evaluates as (a?(b?(c?d:e):f):g)**

An example program is shown below that demonstrates the Ternary operator in java.

public class TernaryOperatorsDemo {

public TernaryOperatorsDemo() {

int x = 10, y = 12, z = 0;

z = x &gt; y ? x : y;

System.out.println("z : " + z);

}

public static void main(String args[]) {

new TernaryOperatorsDemo();

}

}

[**Download**](http://www.wideskills.com/sites/default/files/Zip_Files/Ternary-Operators.zip) TernaryOperatorsDemo Source code

/\*

\* The following programs shows that when no explicit parenthesis is used then the

conditional operator

\* evaluation is from right to left

\*/

public class BooleanEx1 {

static String m1(boolean b) {

return b ? "T" : "F";

}

public static void main(String[] args) {

boolean t1 = false ? false : true ? false : true ? false : true;

boolean t2 = false ? false

: (true ? false : (true ? false : true));

boolean t3 = ((false ? false : true) ? false : true) ? false

: true;

System.out.println(m1(t1) + m1(t2) + m1(t3));

}

}

**Output**

FFT

[**Download**](http://www.wideskills.com/sites/default/files/Zip_Files/Ternary-Operators-2.zip)TernaryOperatorsDemo2 Source code

Type conversion allows a value to be changed from one primitive data type to another. Conversion can occur explicitly, as specified in

the program, or implicitly, by Java itself. Java allows both type widening and type narrowing conversions.

In java Conversions can occur by the following ways:

* Using a cast operator (explicit promotion)
* Using an arithmetic operator is used with arguments of different data types (arithmetic promotion)
* A value of one type is assigned to a variable of a different type (assignment promotion)

## OPERATOR PRECEDENCE

The order in which operators are applied is known as precedence. Operators with a higher precedence are applied before operators with a lower precedence. The operator precedence order of Java is shown below. Operators at the top of the table are applied before operators lower down in the table. If two operators have the same precedence, they are applied in the order they appear in a statement.

That is, from left to right. You can use parentheses to override the default precedence.

|  |  |
| --- | --- |
| **postfix** | **[] . () expr++ expr–** |
| **unary** | **++expr –expr +expr -expr ! ~** |
| **creation/caste** | **new (type)expr** |
| **multiplicative** | **\* / %** |
| **additive** | **+ -** |
| **shift** | **>> >>>** |
| **relational** | **< <= > >= instanceof** |
| **equality** | **== !=** |
| **bitwise AND** | **&** |
| **bitwise exclusive OR** | **^** |
| **bitwise inclusive OR** | **|** |
| **logical AND** | **&&** |
| **logical OR** | **||** |
| **ternary** | **?:** |
| **assignment** | **= “op=”** |
|  | |

**Example**

In an operation such as,

result = 4 + 5 \* 3

First (5 \* 3) is evaluated and the result is added to 4 giving the Final Result value as 19. Note that ‘\*’ takes higher precedence than ‘+’ according to chart shown above. This kind of precedence of one operator over another applies to all the operators.

**QUIZ**

1. How to generate a random number between 1 to x, x being a whole number greater than 1

Ans: double result = x \* Math.random();

Topic 1:Conditional Operator

The Java programming language has includes five simple arithmetic operators like are **+ (addition), - (subtraction), \* (multiplication), / (division)**, and **% (modulo).**The following table summarizes the binary arithmetic operators in the Java programming language.

The relation operators in Java are: ==, !=, <, >, <=, and >=. The meanings of these operators are:

|  |  |
| --- | --- |
| **Use** | **Returns true if** |
| op1 **+** op2 | op1 added to op2 |
| op1 **-** op2 | op2 subtracted from op1 |
| op1 **\***op2 | op1 multiplied with op2 |
| op1 **/** op2 | op1 divided by op2 |
| op1**%**op2 | Computes the remainder of dividing op1 by op2 |

The following java program, ArithmeticProg , defines two integers and two double-precision floating-point numbers and uses the five arithmetic operators to perform different arithmetic operations. This program also uses + to concatenate strings. The arithmetic operations are shown in boldface.

public class ArithmeticProg {

public static void main(String[] args) {

//a few numbers

int i = 10;

int j = 20;

double x = 10.5;

double y = 20.5;

//adding numbers

System.out.println("Adding");

System.out.println(" i + j = " + (i + j));

System.out.println(" x + y = " + (x + y));

//subtracting numbers

System.out.println("Subtracting");

System.out.println(" i - j = " + (i - j));

System.out.println(" x - y = " + (x - y));

//multiplying numbers

System.out.println("Multiplying");

System.out.println(" i \* j = " + (i \* j));

System.out.println(" x \* y = " + (x \* y));

//dividing numbers

System.out.println("Dividing");

System.out.println(" i / j = " + (i / j));

System.out.println(" x / y = " + (x / y));

//computing the remainder resulting

//from dividing numbers

System.out.println("Modulus");

System.out.println(" i % j = " + (i % j));

System.out.println(" x % y = " + (x % y));

}

}

Topic 1:Types of Operator

Java provides a rich set of operators to manipulate variables. We can divide all the Java operators into the following groups:

* Arithmetic Operators
* Relational Operators
* Bitwise Operators
* Logical Operators
* Assignment Operators
* Misc Operators

## The Arithmetic Operators:

Arithmetic operators are used in mathematical expressions in the same way that they are used in algebra. The following table lists the arithmetic operators:

Assume integer variable A holds 10 and variable B holds 20, then:

[**Show Examples**](http://www.tutorialspoint.com/java/java_arithmatic_operators_examples.htm)

|  |  |  |
| --- | --- | --- |
| **Operator** | **Description** | **Example** |
| + | Addition - Adds values on either side of the operator | A + B will give 30 |
| - | Subtraction - Subtracts right hand operand from left hand operand | A - B will give -10 |
| \* | Multiplication - Multiplies values on either side of the operator | A \* B will give 200 |
| / | Division - Divides left hand operand by right hand operand | B / A will give 2 |
| % | Modulus - Divides left hand operand by right hand operand and returns remainder | B % A will give 0 |
| ++ | Increment - Increases the value of operand by 1 | B++ gives 21 |
| -- | Decrement - Decreases the value of operand by 1 | B-- gives 19 |

## The Relational Operators:

There are following relational operators supported by Java language

Assume variable A holds 10 and variable B holds 20, then:

[**Show Examples**](http://www.tutorialspoint.com/java/java_relational_operators_examples.htm)

|  |  |  |
| --- | --- | --- |
| **Operator** | **Description** | **Example** |
| == | Checks if the values of two operands are equal or not, if yes then condition becomes true. | (A == B) is not true. |
| != | Checks if the values of two operands are equal or not, if values are not equal then condition becomes true. | (A != B) is true. |
| > | Checks if the value of left operand is greater than the value of right operand, if yes then condition becomes true. | (A > B) is not true. |
| < | Checks if the value of left operand is less than the value of right operand, if yes then condition becomes true. | (A < B) is true. |
| >= | Checks if the value of left operand is greater than or equal to the value of right operand, if yes then condition becomes true. | (A >= B) is not true. |
| <= | Checks if the value of left operand is less than or equal to the value of right operand, if yes then condition becomes true. | (A <= B) is true. |

## The Bitwise Operators:

Java defines several bitwise operators, which can be applied to the integer types, long, int, short, char, and byte.

Bitwise operator works on bits and performs bit-by-bit operation. Assume if a = 60; and b = 13; now in binary format they will be as follows:

a = 0011 1100

b = 0000 1101

-----------------

a&b = 0000 1100

a|b = 0011 1101

a^b = 0011 0001

~a  = 1100 0011

The following table lists the bitwise operators:

Assume integer variable A holds 60 and variable B holds 13 then:

[**Show Examples**](http://www.tutorialspoint.com/java/java_bitwise_operators_examples.htm)

|  |  |  |
| --- | --- | --- |
| **Operator** | **Description** | **Example** |
| & | Binary AND Operator copies a bit to the result if it exists in both operands. | (A & B) will give 12 which is 0000 1100 |
| | | Binary OR Operator copies a bit if it exists in either operand. | (A | B) will give 61 which is 0011 1101 |
| ^ | Binary XOR Operator copies the bit if it is set in one operand but not both. | (A ^ B) will give 49 which is 0011 0001 |
| ~ | Binary Ones Complement Operator is unary and has the effect of 'flipping' bits. | (~A ) will give -61 which is 1100 0011 in 2's complement form due to a signed binary number. |
| << | Binary Left Shift Operator. The left operands value is moved left by the number of bits specified by the right operan | A << 2 will give 240 which is 1111 0000 |
| >> | Binary Right Shift Operator. The left operands value is moved right by the number of bits specified by the right operand. | A >> 2 will give 15 which is 1111 |
| >>> | Shift right zero fill operator. The left operands value is moved right by the number of bits specified by the right operand and shifted values are filled up with zeros. | A >>>2 will give 15 which is 0000 1111 |

## The Logical Operators:

The following table lists the logical operators:

Assume Boolean variables A holds true and variable B holds false, then:

[**Show Examples**](http://www.tutorialspoint.com/java/java_logical_operators_examples.htm)

|  |  |  |
| --- | --- | --- |
| **Operator** | **Description** | **Example** |
| && | Called Logical AND operator. If both the operands are non-zero, then the condition becomes true. | (A && B) is false. |
| || | Called Logical OR Operator. If any of the two operands are non-zero, then the condition becomes true. | (A || B) is true. |
| ! | Called Logical NOT Operator. Use to reverses the logical state of its operand. If a condition is true then Logical NOT operator will make false. | !(A && B) is true. |

## The Assignment Operators:

There are following assignment operators supported by Java language:

[**Show Examples**](http://www.tutorialspoint.com/java/java_assignment_operators_examples.htm)

|  |  |  |
| --- | --- | --- |
| **Operator** | **Description** | **Example** |
| = | Simple assignment operator, Assigns values from right side operands to left side operand | C = A + B will assign value of A + B into C |
| += | Add AND assignment operator, It adds right operand to the left operand and assign the result to left operand | C += A is equivalent to C = C + A |
| -= | Subtract AND assignment operator, It subtracts right operand from the left operand and assign the result to left operand | C -= A is equivalent to C = C - A |
| \*= | Multiply AND assignment operator, It multiplies right operand with the left operand and assign the result to left operand | C \*= A is equivalent to C = C \* A |
| /= | Divide AND assignment operator, It divides left operand with the right operand and assign the result to left operand | C /= A is equivalent to C = C / A |
| %= | Modulus AND assignment operator, It takes modulus using two operands and assign the result to left operand | C %= A is equivalent to C = C % A |
| <<= | Left shift AND assignment operator | C <<= 2 is same as C = C << 2 |
| >>= | Right shift AND assignment operator | C >>= 2 is same as C = C >> 2 |
| &= | Bitwise AND assignment operator | C &= 2 is same as C = C & 2 |
| ^= | bitwise exclusive OR and assignment operator | C ^= 2 is same as C = C ^ 2 |
| |= | bitwise inclusive OR and assignment operator | C |= 2 is same as C = C | 2 |

## Misc Operators

There are few other operators supported by Java Language.

## Conditional Operator ( ? : ):

Conditional operator is also known as the ternary operator. This operator consists of three operands and is used to evaluate Boolean expressions. The goal of the operator is to decide which value should be assigned to the variable. The operator is written as:

variable x = (expression) ? value if true : value if false

Following is the example:

public class Test {

public static void main(String args[]){

int a , b;

a = 10;

b = (a == 1) ? 20: 30;

System.out.println( "Value of b is : " + b );

b = (a == 10) ? 20: 30;

System.out.println( "Value of b is : " + b );

}

}

This would produce the following result:

Value of b is : 30

Value of b is : 20

## instanceof Operator:

This operator is used only for object reference variables. The operator checks whether the object is of a particular type(class type or interface type). instanceof operator is wriiten as:

( Object reference variable ) instanceof (class/interface type)

If the object referred by the variable on the left side of the operator passes the IS-A check for the class/interface type on the right side, then the result will be true. Following is the example:

public class Test {

public static void main(String args[]){

String name = "James";

// following will return true since name is type of String

boolean result = name instanceof String;

System.out.println( result );

}

}

This would produce the following result:

true

This operator will still return true if the object being compared is the assignment compatible with the type on the right. Following is one more example:

class Vehicle {}

public class Car extends Vehicle {

public static void main(String args[]){

Vehicle a = new Car();

boolean result = a instanceof Car;

System.out.println( result );

}

}

This would produce the following result:

true

## Precedence of Java Operators:

Operator precedence determines the grouping of terms in an expression. This affects how an expression is evaluated. Certain operators have higher precedence than others; for example, the multiplication operator has higher precedence than the addition operator:

For example, x = 7 + 3 \* 2; here x is assigned 13, not 20 because operator \* has higher precedence than +, so it first gets multiplied with 3\*2 and then adds into 7.

Here, operators with the highest precedence appear at the top of the table, those with the lowest appear at the bottom. Within an expression, higher precedence operators will be evaluated first.

|  |  |  |
| --- | --- | --- |
| **Category** | **Operator** | **Associativity** |
| Postfix | () [] . (dot operator) | Left toright |
| Unary | ++ - - ! ~ | Right to left |
| Multiplicative | \* / % | Left to right |
| Additive | + - | Left to right |
| Shift | >> >>> << | Left to right |
| Relational | > >= < <= | Left to right |
| Equality | == != | Left to right |
| Bitwise AND | & | Left to right |
| Bitwise XOR | ^ | Left to right |
| Bitwise OR | | | Left to right |
| Logical AND | && | Left to right |
| Logical OR | || | Left to right |
| Conditional | ?: | Right to left |
| Assignment | = += -= \*= /= %= >>= <<= &= ^= |= | Right to left |
| Comma | , | Left to right |

Chapter 4: **Control Flow Statements**

Topic 1: **Branching Statements: break, continue, return, try-catch-finally, assert**  :

## The break Statement

The break statement has two forms: labeled and unlabeled. You saw the unlabeled form in the previous discussion of the switch statement. You can also use an unlabeledbreak to terminate a for, while, or do-while loop, as shown in the following [BreakDemo](http://docs.oracle.com/javase/tutorial/java/nutsandbolts/examples/BreakDemo.java" \t "_blank) program:

class BreakDemo {

public static void main(String[] args) {

int[] arrayOfInts =

{ 32, 87, 3, 589,

12, 1076, 2000,

8, 622, 127 };

int searchfor = 12;

int i;

boolean foundIt = false;

for (i = 0; i < arrayOfInts.length; i++) {

if (arrayOfInts[i] == searchfor) {

foundIt = true;

**break;**

}

}

if (foundIt) {

System.out.println("Found " + searchfor + " at index " + i);

} else {

System.out.println(searchfor + " not in the array");

}

}

}

This program searches for the number 12 in an array. The break statement, shown in boldface, terminates the for loop when that value is found. Control flow then transfers to the statement after the for loop. This program's output is:

Found 12 at index 4

An unlabeled break statement terminates the innermost switch, for, while, or do-while statement, but a labeled break terminates an outer statement. The following program, [BreakWithLabelDemo](http://docs.oracle.com/javase/tutorial/java/nutsandbolts/examples/BreakWithLabelDemo.java" \t "_blank), is similar to the previous program, but uses nested for loops to search for a value in a two-dimensional array. When the value is found, a labeled break terminates the outer for loop (labeled "search"):

class BreakWithLabelDemo {

public static void main(String[] args) {

int[][] arrayOfInts = {

{ 32, 87, 3, 589 },

{ 12, 1076, 2000, 8 },

{ 622, 127, 77, 955 }

};

int searchfor = 12;

int i;

int j = 0;

boolean foundIt = false;

search:

for (i = 0; i < arrayOfInts.length; i++) {

for (j = 0; j < arrayOfInts[i].length;

j++) {

if (arrayOfInts[i][j] == searchfor) {

foundIt = true;

break search;

}

}

}

if (foundIt) {

System.out.println("Found " + searchfor + " at " + i + ", " + j);

} else {

System.out.println(searchfor + " not in the array");

}

}

}

This is the output of the program.

Found 12 at 1, 0

The break statement terminates the labeled statement; it does not transfer the flow of control to the label. Control flow is transferred to the statement immediately following the labeled (terminated) statement.

## The continue Statement

The continue statement skips the current iteration of a for, while , or do-while loop. The unlabeled form skips to the end of the innermost loop's body and evaluates theboolean expression that controls the loop. The following program, [ContinueDemo](http://docs.oracle.com/javase/tutorial/java/nutsandbolts/examples/ContinueDemo.java" \t "_blank) , steps through a String, counting the occurences of the letter "p". If the current character is not a p, the continue statement skips the rest of the loop and proceeds to the next character. If it *is* a "p", the program increments the letter count.

class ContinueDemo {

public static void main(String[] args) {

String searchMe = "peter piper picked a " + "peck of pickled peppers";

int max = searchMe.length();

int numPs = 0;

for (int i = 0; i < max; i++) {

// interested only in p's

if (searchMe.charAt(i) != 'p')

continue;

// process p's

numPs++;

}

System.out.println("Found " + numPs + " p's in the string.");

}

}

Here is the output of this program:

Found 9 p's in the string.

To see this effect more clearly, try removing the continue statement and recompiling. When you run the program again, the count will be wrong, saying that it found 35 p's instead of 9.

A labeled continue statement skips the current iteration of an outer loop marked with the given label. The following example program, ContinueWithLabelDemo, uses nested loops to search for a substring within another string. Two nested loops are required: one to iterate over the substring and one to iterate over the string being searched. The following program, [ContinueWithLabelDemo](http://docs.oracle.com/javase/tutorial/java/nutsandbolts/examples/ContinueWithLabelDemo.java" \t "_blank), uses the labeled form of continue to skip an iteration in the outer loop.

class ContinueWithLabelDemo {

public static void main(String[] args) {

String searchMe = "Look for a substring in me";

String substring = "sub";

boolean foundIt = false;

int max = searchMe.length() -

substring.length();

test:

for (int i = 0; i <= max; i++) {

int n = substring.length();

int j = i;

int k = 0;

while (n-- != 0) {

if (searchMe.charAt(j++) != substring.charAt(k++)) {

continue test;

}

}

foundIt = true;

break test;

}

System.out.println(foundIt ? "Found it" : "Didn't find it");

}

}

Here is the output from this program.

Found it

## The return Statement

The last of the branching statements is the return statement. The return statement exits from the current method, and control flow returns to where the method was invoked. The return statement has two forms: one that returns a value, and one that doesn't. To return a value, simply put the value (or an expression that calculates the value) after thereturn keyword.

return ++count;

The data type of the returned value must match the type of the method's declared return value. When a method is declared void, use the form of return that doesn't return a value.

return;

## Print out the multiplication table for 1 to 10

degree of difficulty: 3

Implement a Java-main-method that prints out the multiplication table for all numbers from 1 to 10. Use the tabulator character '\t' to align the values. The output of your method should be as follows:

1 2 3 4 5 6 7 8 9 10

2 4 6 8 10 12 14 16 18 20

3 6 9 12 15 18 21 24 27 30

4 8 12 16 20 24 28 32 36 40

5 10 15 20 25 30 35 40 45 50

6 12 18 24 30 36 42 48 54 60

7 14 21 28 35 42 49 56 63 70

8 16 24 32 40 48 56 64 72 80

9 18 27 36 45 54 63 72 81 90

10 20 30 40 50 60 70 80 90 100

[Solution](http://www.home.hs-karlsruhe.de/~pach0003/informatik_1/aufgaben/en/doc/de/hska/java/exercises/flowcontrol/MultiplicationTable.html)

## Sort three numbers

degree of difficulty: 3

Implement a Java-method with three local int variables a, b, and c that sorts these three values in ascending order by comparing and exchanging their values. At the end of the program a <= b <= c must hold.

You can solve this exercises with three if in minimum. Test your method with all six permutation of three different numbers.

[Solution](http://www.home.hs-karlsruhe.de/~pach0003/informatik_1/aufgaben/en/doc/de/hska/java/exercises/flowcontrol/SortThreeNumbers.html)

## Calculate (german) grades for an examination

degree of difficulty: 3

The maximum points you can get in the written exam to my lecture is 120. At 60 points you have passed the exam (grade 4.0). For every additional 5 points the next better grade. Because of juridictional problems, the grade 4.3 does not exist (it is 4,7, too, in that case). It is possible to get 0.5 points for your solutions in the exam. The following table shows the mapping between points and their grades:

|  |  |
| --- | --- |
| grade | points |
| 5.0 | 0 - 49.5 |
| 4.7 | 50 - 59.5 |
| 4.0 | 60 - 64.5 |
| 3.7 | 65 - 69.5 |
| 3.3 | 70 - 74.5 |
| 3.0 | 75 - 79.5 |
| 2.7 | 80 - 84.5 |
| 2.3 | 85 - 89.5 |
| 2.0 | 90 - 94.5 |
| 1.7 | 95 - 99.5 |
| 1.3 | 100 - 104.5 |
| 1.0 | 105 - 120 |

Implement a Java-function that returns the grade for the points reached in the exam and prints out the grade for all points from 0 to 120 (in steps of 0.5).

[Solution](http://www.home.hs-karlsruhe.de/~pach0003/informatik_1/aufgaben/en/doc/de/hska/java/exercises/flowcontrol/CalculateGrades.html)

## Find the day of the week of a given date

degree of difficulty: 2

Implement a Java-method that prints out the day of the week for a given day (1..31), month (1..12) and year.

The day of the week of dates between March 1900 and February 2100 can be calculated as follows:

First, you have to calculate the total number of days from 1900/1/1 to the given date (see below, for details). Secondly, you divide this number by 7 with integer remainder: this now is the day of the week, with 0 as sunday, 1 as monday, etc.

To calculate the total number of days you have to implement the following steps:

* Subtract 1900 from the given year and multiply the result by 365
* Add the missing leaps years by adding (year - 1900) / 4.
* If the year itself is a leap year and the month is January or February, you have to subtract 1 from the previous result.
* Now add the all days of the year up to the given one to the result (in case of february always 28, because the additional day for a leap year already have been added).

Here some dates with the day of the week for testing your Java programm:

* Easter Sunday: 1916/3/23, 2007/4/8, 2010/4/4
* Ash Wednesday: 2006/3/1, 2007/2/21, 2010/2/17

[Solution](http://www.home.hs-karlsruhe.de/~pach0003/informatik_1/aufgaben/en/doc/de/hska/java/exercises/flowcontrol/DayOfTheWeek.html)

## Winning strategy for Roulette

degree of difficulty 2

A foolproof winning strategy for roulette is to always bet on the same color: If you loose your wager, then you simply double the bet to compensate the loss. The probability for red (or black) is 50% (we ignore the existence of zero).

Implement a Java method, that calculates how much money you approximatly need for playing 1 000 000 times with 1 Euro as the first bet.

Use the expression Math.random() < 0.5 for checking the probability.

How much money do you approximatly need, without loosing it during gambling.

Extend your Java method, such that it calculates from a given seed money the number of rounds you need to play to loose all your money.

[Solution](http://www.home.hs-karlsruhe.de/~pach0003/informatik_1/aufgaben/en/doc/de/hska/java/exercises/flowcontrol/Roulette.html)

## Convert a given number of bytes into a representation with metric units

degree of difficulty 3

A number of bytes given as a int value should be printed out with at most three digits before the decimal comma. The ouput for four different values:

123 Byte are 123.0 Byte

15323 Byte are 15.323 KByte

15323000 Byte are 15.323 MByte

1532300001 Byte are 1.532300001 GByte

Do not use iterations (only if-else).

Topic 1: **Selection Statements: if, if-else, switch**

A loop is a section of code that is executed repeatedly until a stopping condition is met. A typical loop may look like:

while there's more data {

Read a Line of Data

Do Something with the Data

}

There are many different kinds of loops in Java including while, for, and do while loops. They differ primarily in the stopping conditions used.

For loops typically iterate a fixed number of times and then exit. While loops iterate continuously until a particular condition is met. You usually do not know in advance how many times a while loop will loop.

In this case we want to write a loop that will print each of the command line arguments in succession, starting with the first one. We don't know in advance how many arguments there will be, but we can easily find this out before the loop starts using the args.length. Therefore we will write this with a for loop. Here's the code:

Source Code

// This is the Hello program in Java

class Hello {

public static void main (String args[]) {

int i;

/\* Now let's say hello \*/

System.out.print("Hello ");

for (i=0; i < args.length; i = i++) {

System.out.print(args[i]);

System.out.print(" ");

}

System.out.println();

}

}

We begin the code by declaring our variables. In this case we have exactly one variable, the integer i. i

Then we begin the program by saying "Hello" just like before.

Next comes the for loop. The loop begins by initializing the counter variable i to be zero. This happens exactly once at the beginning of the loop. Programming tradition that dates back to Fortran insists that loop indices be named i, j, k, l, m and n in that order.

Next is the test condition. In this case we test that i is less than the number of arguments. When i becomes equal to the number of arguments, (args.length) we exit the loop and go to the first statement after the loop's closing brace. You might think that we should test for i being less than or equal to the number of arguments; but remember that we began counting at zero, not one.

Finally we have the increment step, i++ (i=i+1). This is executed at the end of each iteration of the loop. Without this we'd continue to loop forever since i would always be less than args.length.

Topic 1: **Looping Statements: for, while, do-while** :

# Java Control Flow Statements

[TOC](http://www.wideskills.com/java-tutorial)

* [Java Operators Tutorial](http://www.wideskills.com/java-tutorial/java-operators-tutorial)
* [Introduction To Java Modifiers](http://www.wideskills.com/java-tutorial/introduction-to-java-modifiers)

Java Control statements control the order of execution in a java program, based on data values and conditional logic. There are three main categories of control flow statements;

Selection statements: if, if-else and switch.

Loop statements: while, do-while and for.

 Transfer statements: break, continue, return, try-catch-finally and assert.

We use control statements when we want to change the default sequential order of execution

## SELECTION STATEMENTS

The If Statement

The if statement executes a block of code only if the specified expression is true. If the value is false, then the if block is skipped and execution continues with the rest of the program. You can either have a single statement or a block of code within an if statement. Note that the conditional expression must be a Boolean expression.

The simple if statement has the following syntax:

*if (<conditional expression>)   
       <statement action>*

Below is an example that demonstrates conditional execution based on if statement condition.

public class IfStatementDemo {

public static void main(String[] args) {

int a = 10, b = 20;

if (a &gt; b)

System.out.println("a &gt; b");

if (a &lt; b)

System.out.println("b &gt; a");

}

}

Output

b > a

[**Download**](http://www.wideskills.com/sites/default/files/Zip_Files/if-statement.zip) IfStatementDemo.java

The If-else Statement

The if/else statement is an extension of the if statement. If the statements in the if statement fails, the statements in the else block are executed. You can either have a single statement or a block of code within if-else blocks. Note that the conditional expression must be a Boolean expression.

The if-else statement has the following syntax:

    if (<conditional expression>)  
    <statement action>  
    else  
   <statement action>

Below is an example that demonstrates conditional execution based on if else statement condition.

public class IfElseStatementDemo {

public static void main(String[] args) {

int a = 10, b = 20;

if (a &gt; b) {

System.out.println("a &gt; b");

} else {

System.out.println("b &gt; a");

}

}

}

[**Download**](http://www.wideskills.com/sites/default/files/Zip_Files/if-else-statement.zip)IfElseStatementDemo.java

Switch Case Statement The switch case statement, also called a case statement is a multi-way branch with several choices. A switch is easier to implement than a series of if/else statements. The switch statement begins with a keyword, followed by an expression that equates to a no long integral value. Following the controlling expression is a code block that contains zero or more labeled cases. Each label must equate to an integer constant and each must be unique. When the switch statement executes, it compares the value of the controlling expression to the values of each case label. The program will select the value of the case label that equals the value of the controlling expression and branch down that path to the end of the code block. If none of the case label values match, then none of the codes within the switch statement code block will be executed. Java includes a default label to use in cases where there are no matches. We can have a nested switch within a case block of an outer switch. Its general form is as follows:

*switch (<non-long integral expression>) {  
     case* *label1:* *<statement1>  
     case* *label2:* *<statement2>*        … *case* *labeln:* *<statementn>  
    default: <statement>*    } // end switch

When executing a switch statement, the program falls through to the next case. Therefore, if you want to exit in the middle of the switch statement code block, you must insert a break statement, which causes the program to continue executing after the current code block.

Below is a java example that demonstrates conditional execution based on nested if else statement condition to find the greatest of 3 numbers.

public class SwitchCaseStatementDemo {

public static void main(String[] args) {

int a = 10, b = 20, c = 30;

int status = -1;

if (a &gt; b &amp;&amp; a &gt; c) {

status = 1;

} else if (b &gt; c) {

status = 2;

} else {

status = 3;

}

switch (status) {

case 1:

System.out.println("a is the greatest");

break;

case 2:

System.out.println("b is the greatest");

break;

case 3:

System.out.println("c is the greatest");

break;

default:

System.out.println("Cannot be determined");

}

}

}

Chapter 5:OOP

Topic 1: **Object-oriented programming**    :

# Object-Oriented Programming Concepts

If you've never used an object-oriented programming language before, you'll need to learn a few basic concepts before you can begin writing any code. This lesson will introduce you to objects, classes, inheritance, interfaces, and packages. Each discussion focuses on how these concepts relate to the real world, while simultaneously providing an introduction to the syntax of the Java programming language.

## [What Is an Object?](http://docs.oracle.com/javase/tutorial/java/concepts/object.html)

An object is a software bundle of related state and behavior. Software objects are often used to model the real-world objects that you find in everyday life. This lesson explains how state and behavior are represented within an object, introduces the concept of data encapsulation, and explains the benefits of designing your software in this manner.

## [What Is a Class?](http://docs.oracle.com/javase/tutorial/java/concepts/class.html)

A class is a blueprint or prototype from which objects are created. This section defines a class that models the state and behavior of a real-world object. It intentionally focuses on the basics, showing how even a simple class can cleanly model state and behavior.

## [What Is Inheritance?](http://docs.oracle.com/javase/tutorial/java/concepts/inheritance.html)

Inheritance provides a powerful and natural mechanism for organizing and structuring your software. This section explains how classes inherit state and behavior from their superclasses, and explains how to derive one class from another using the simple syntax provided by the Java programming language.

## [What Is an Interface?](http://docs.oracle.com/javase/tutorial/java/concepts/interface.html)

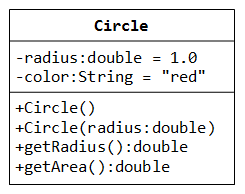
An interface is a contract between a class and the outside world. When a class implements an interface, it promises to provide the behavior published by that interface. This section defines a simple interface and explains the necessary changes for any class that implements it.

## [What Is a Package?](http://docs.oracle.com/javase/tutorial/java/concepts/package.html)

A package is a namespace for organizing classes and interfaces in a logical manner. Placing your code into packages makes large software projects easier to manage. This section explains why this is useful, and introduces you to the Application Programming Interface (API) provided by the Java platform.

### 1.  Exercises on Classes and Instances

#### 1.1  Exercise: The Circle Class



A class called **circle** is designed as shown in the following class diagram. It contains:

* Two private instance variables: radius (of type double) and color (of type String), with default value of 1.0 and "red", respectively.
* Two overloaded constructors;
* Two public methods: getRadius() and getArea().

The source codes for Circle is as follows:

public class **Circle** { // save as "Circle.java"

// private instance variable, not accessible from outside this class

private double radius;

private String color;

// 1st constructor, which sets both radius and color to default

public Circle() {

radius = 1.0;

color = "red";

}

// 2nd constructor with given radius, but color default

public Circle(double r) {

radius = r;

color = "red";

}

// A public method for retrieving the radius

public double getRadius() {

return radius;

}

// A public method for computing the area of circle

public double getArea() {

return radius\*radius\*Math.PI;

}

}

Compile "Circle.java". Can you run the Circle class? Why? This Circle class does not have a main() method. Hence, it cannot be run directly. This Circle class is a “building block” and is meant to be used in another program.

Let us write a test program called TestCircle which uses the Circle class, as follows:

public class **TestCircle** { // save as "TestCircle.java"

public static void main(String[] args) {

// Declare and allocate an instance of class Circle called c1

// with default radius and color

Circle c1 = new Circle();

// Use the dot operator to invoke methods of instance c1.

System.out.println("The circle has radius of "

+ c1.getRadius() + " and area of " + c1.getArea());

// Declare and allocate an instance of class circle called c2

// with the given radius and default color

Circle c2 = new Circle(2.0);

// Use the dot operator to invoke methods of instance c2.

System.out.println("The circle has radius of "

+ c2.getRadius() + " and area of " + c2.getArea());

}

}

Now, run the TestCircle and study the results.

TRY:

1. **Constructor:** Modify the class Circle to include a third constructor for constructing a Circle instance with the given radius and color.
2. // Construtor to construct a new instance of Circle with the given radius and color

public Circle (double r, String c) {......}

Modify the test program TestCircle to construct an instance of Circle using this constructor.

1. **Getter:** Add a getter for variable color for retrieving the color of a Circle instance.
2. // Getter for instance variable color

public String getColor() {......}

Modify the test program to test this method.

1. **public vs. private:** In TestCircle, can you access the instance variable radius directly (e.g., System.out.println(c1.radius)); or assign a new value to radius (e.g.,c1.radius=5.0)? Try it out and explain the error messages.
2. **Setter:** Is there a need to change the values of radius and color of a Circle instance after it is constructed? If so, add two public methods called setters for changing the radius andcolor of a Circle instance as follows:
3. // Setter for instance variable radius
4. public void setRadius(double r) {
5. radius = r;
6. }
8. // Setter for instance variable color

public void setColor(String c) { ...... }

Modify the TestCircle to test these methods, e.g.,

Circle c3 = new Circle(); // construct an instance of Circle

c3.setRadius(5.0); // change radius

c3.setColor(...); // change color

1. **Keyword "this":** Instead of using variable names such as r (for radius) and c (for color) in the methods' arguments, it is better to use variable names radius (for radius) andcolor (for color) and use the special keyword "this" to resolve the conflict between instance variables and methods' arguments. For example,
2. // Instance variable
3. private double radius;
5. // Setter of radius
6. public void setRadius(double radius) {
7. this.radius = radius; // "this.radius" refers to the instance variable
8. // "radius" refers to the method's argument

}

Modify ALL the constructors and setters in the Circle class to use the keyword "this".

1. **Method toString():** Every well-designed Java class should contain a public method called toString() that returns a short description of the instance (in a return type of String). The toString() method can be called explicitly (via instanceName.toString()) just like any other method; or implicitly through println(). If an instance is passed to theprintln(anInstance) method, the toString() method of that instance will be invoked implicitly. For example, include the following toString() methods to the Circle class:
2. public String toString() {
3. return "Circle: radius=" + radius + " color=" + color;

}

Try calling toString() method explicitly, just like any other method:

Circle c1 = new Circle(5.0);

System.out.println(c1.toString()); // explicit call

toString() is called implicitly when an instance is passed to println() method, for example,

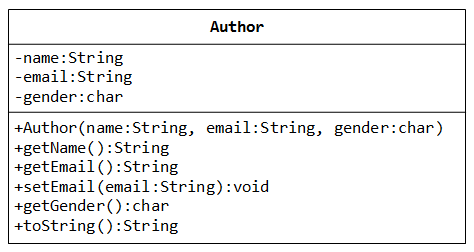
Circle c2 = new Circle(1.2);

System.out.println(c2.toString()); // explicit call

System.out.println(c2); // println() calls toString() implicitly, same as above

System.out.println("Operator '+' invokes toString() too: " + c2); // '+' invokes toString() too

#### 1.2  Exercise: The Author and Book Classes



A class called Author is designed as shown in the class diagram. It contains:

* Three private instance variables: name (String), email (String), and gender (char of either 'm' or 'f');
* One constructor to initialize the name, email and gender with the given values;

public Author (String name, String email, char gender) {......}

(There is no default constructor for Author, as there are no defaults for name, email and gender.)

* public getters/setters: getName(), getEmail(), setEmail(), and getGender();  
  (There are no setters for name and gender, as these attributes cannot be changed.)
* A toString() method that returns "author-name (gender) at email", e.g., "Tan Ah Teck (m) at ahTeck@somewhere.com".

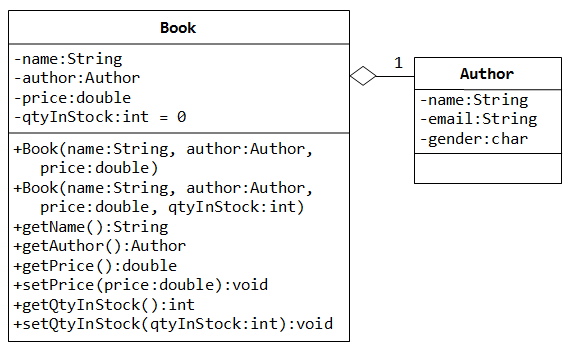
Write the Author class. Also write a test program called TestAuthor to test the constructor and publicmethods. Try changing the email of an author, e.g.,

Author anAuthor = new Author("Tan Ah Teck", "ahteck@somewhere.com", 'm');

System.out.println(anAuthor); // call toString()

anAuthor.setEmail("paul@nowhere.com")

System.out.println(anAuthor);



A class called Book is designed as shown in the class diagram. It contains:

* Four private instance variables: name (String), author (of the class Author you have just created, assume that each book has one and only one author), price (double), andqtyInStock (int);
* Two constructors:
* public Book (String name, Author author, double price) {...}
* public Book (String name, Author author, double price,

int qtyInStock) {...}

* public methods getName(), getAuthor(), getPrice(), setPrice(), getQtyInStock(),setQtyInStock().
* toString() that returns "'book-name' by author-name (gender) at email".  
  (Take note that the Author's toString() method returns "author-name (gender) at email".)

Write the class Book (which uses the Author class written earlier). Also write a test program calledTestBook to test the constructor and public methods in the class Book. Take Note that you have to construct an instance of Author before you can construct an instance of Book. E.g.,

Author anAuthor = new Author(......);

Book aBook = new Book("Java for dummy", anAuthor, 19.95, 1000);

// Use an anonymous instance of Author

Book anotherBook = new Book("more Java for dummy", new Author(......), 29.95, 888);

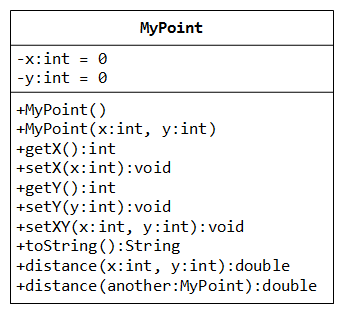
Take note that both Book and Author classes have a variable called name. However, it can be differentiated via the referencing instance. For a Book instance says aBook, aBook.name refers to the name of the book; whereas for an Author's instance say auAuthor, anAuthor.name refers to the name of the author. There is no need (and not recommended) to call the variables bookNameand authorName.

TRY:

1. Printing the name and email of the author from a Book instance. (Hint: aBook.getAuthor().getName(), aBook.getAuthor().getEmail()).
2. Introduce new methods called getAuthorName(), getAuthorEmail(), getAuthorGender() in the Book class to return the name, email and gender of the author of the book. For example,

public String getAuthorName() { ...... }

#### 1.3  Exercise: The MyPoint Class



A class called MyPoint, which models a 2D point with x and y coordinates, is designed as shown in the class diagram. It contains:

* Two instance variables x (int) and y (int).
* A "no-argument" (or "no-arg") constructor that construct a point at (0, 0).
* A constructor that constructs a point with the given x and y coordinates.
* Getter and setter for the instance variables x and y.
* A method setXY() to set both x and y.
* A toString() method that returns a string description of the instance in the format "(x, y)".
* A method called distance(int x, int y) that returns the distance from this point to another point at the given (x, y)coordinates.
* An overloaded distance(MyPoint another) that returns the distance from this point to the given MyPoint instance another.

You are required to:

1. Write the code for the class MyPoint. Also write a test program (called TestMyPoint) to test all the methods defined in the class.  
   Hints:
2. // Overloading method distance()
3. public double distance(int x, int y) { // this version takes two ints as arguments
4. int xDiff = this.x – x;
5. int yDiff = ......
6. return Math.sqrt(xDiff\*xDiff + yDiff\*yDiff);
7. }
9. public double distance(MyPoint another) { // this version takes a MyPoint instance as argument
10. int xDiff = this.x – another.x;
11. .......
12. }
14. // Test program
15. MyPoint p1 = new MyPoint(3, 0);
16. MyPoint p2 = new MyPoint(0, 4);
17. ......
18. // Testing the overloaded method distance()
19. System.out.println(p1.distance(p2)); // which version?
20. System.out.println(p1.distance(5, 6)); // which version?

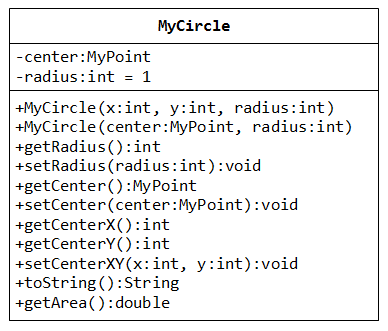
.....

1. Write a program that allocates 10 points in an array of MyPoint, and initializes to (1, 1), (2, 2), ... (10, 10).  
   Hints: You need to allocate the array, as well as each of the ten MyPoint instances.
2. MyPoint[] points = new MyPoint[10]; // Declare and allocate an array of MyPoint
3. for (......) {
4. points[i] = new MyPoint(...); // Allocate each of MyPoint instances

}

Notes: Point is such a common entity that JDK certainly provided for in all flavors.

#### 1.4  Exercise: The MyCircle Class



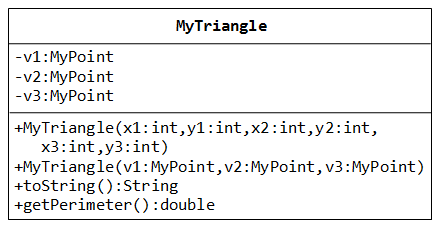
A class called MyCircle, which models a circle with a center (x, y) and a radius, is designed as shown in the class diagram. The MyCircle class uses an instance of MyPoint class (created in the previous exercise) as its center.

The class contains:

* Two private instance variables: center (an instance of MyPoint) and radius (int).
* A constructor that constructs a circle with the given center's (x, y) and radius.
* An overloaded constructor that constructs a MyCircle given a MyPoint instance as center, and radius.
* Various getters and setters.
* A toString() method that returns a string description of this instance in the format "Circle @ (x, y) radius=r".
* A getArea() method that returns the area of the circle in double.

Write the MyCircle class. Also write a test program (called TestMyCircle) to test all the methods defined in the class.

#### 1.5  Exercise: The MyTriangle Class



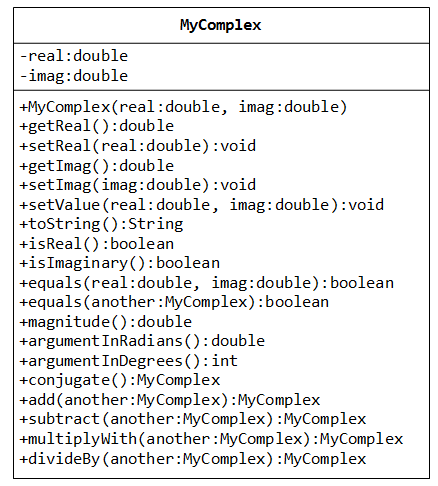
A class called MyTriangle, which models a triangle with 3 vertices, is designed as follows. The MyTriangle class uses three MyPoint instances (created in the earlier exercise) as the three vertices.

The class contains:

* Three private instance variables v1, v2, v3 (instances of MyPoint), for the three vertices.
* A constructor that constructs a MyTriangle with three points v1=(x1, y1), v2=(x2, y2), v3=(x3, y3).
* An overloaded constructor that constructs a MyTriangle given three instances of MyPoint.
* A toString() method that returns a string description of the instance in the format "Triangle @ (x1, y1), (x2, y2), (x3, y3)".
* A getPerimeter() method that returns the length of the perimeter in double. You should use the distance()method of MyPoint to compute the perimeter.
* A method printType(), which prints "equilateral" if all the three sides are equal, "isosceles" if any two of the three sides are equal, or "scalene" if the three sides are different.

Write the MyTriangle class. Also write a test program (called TestMyTriangle) to test all the methods defined in the class.

#### 1.6  Exercise: The MyComplex class



A class called MyComplex, which models complex numbers x+yi, is designed as shown in the class diagram. It contains:

* Two instance variable named real(double) and imag(double) which stores the real and imaginary parts of the complex number respectively.
* A constructor that creates a MyComplex instance with the given real and imaginary values.
* Getters and setters for instance variables real and imag.
* A method setValue() to set the value of the complex number.
* A toString() that returns "(x + yi)" where x and y are the real and imaginary parts respectively.
* Methods isReal() and isImaginary() that returns true if this complex number is real or imaginary, respectively. Hint:

return (imag == 0); // isReal()

* A method equals(double real, double imag) that returns true if this complex number is equal to the given complex number of (real, imag).
* An overloaded equals(MyComplex another) that returns true if this complex number is equal to the givenMyComplex instance another.
* A method magnitude()that returns the magnitude of this complex number.

magnitude(x+yi) = Math.sqrt(x2 + y2)

* Methods argumentInRadians() and argumentInDegrees() that returns the argument of this complex number in radians (in double) and degrees (in int) respectively.

arg(x+yi) = Math.atan2(y, x) (in radians)

Note: The Math library has two arc-tangent methods, Math.atan(double) and Math.atan2(double, double). We commonly use the Math.atan2(y, x) instead of Math.atan(y/x) to avoid division by zero. Read the documentation of Math class in package java.lang.

* A method conjugate() that returns a new MyComplex instance containing the complex conjugate of this instance.

conjugate(x+yi) = x - yi

Hint:

return new MyComplex(real, -imag); // construct a new instance and return the constructed instance

* Methods add(MyComplex another) and subtract(MyComplex another) that adds and subtract this instance with the given MyComplex instance another, and returns a new MyComplexinstance containing the result.
* (a + bi) + (c + di) = (a+c) + (b+d)i

(a + bi) - (c + di) = (a-c) + (b-d)i

* Methods multiplyWith(MyComplex another) and divideBy(MyComplex another) that multiplies and divides this instance with the given MyComplex instance another, keep the result in this instance, and returns this instance.
* (a + bi) \* (c + di) = (ac - bd) + (ad + bc)i

(a + bi) / (c + di) = [(a + bi) \* (c – di)] / (c2 + d2)

Hint:

return this; // return "this" instance

You are required to:

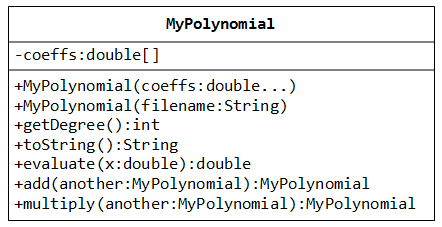
1. Write the MyComplex class.
2. Write a test program to test all the methods defined in the class.
3. Write an application called MyComplexApp that uses the MyComplex class. The application shall prompt the user for two complex numbers, print their values, check for real, imaginary and equality, and carry out all the arithmetic operations.
4. Enter complex number 1 (real and imaginary part): **1.1 2.2**
5. Enter complex number 2 (real and imaginary part): **3.3 4.4**
7. Number 1 is: (1.1 + 2.2i)
8. (1.1 + 2.2i) is NOT a pure real number
9. (1.1 + 2.2i) is NOT a pure imaginary number
11. Number 2 is: (3.3 + 4.4i)
12. (3.3 + 4.4i) is NOT a pure real number
13. (3.3 + 4.4i) is NOT a pure imaginary number
15. (1.1 + 2.2i) is NOT equal to (3.3 + 4.4i)
16. (1.1 + 2.2i) + (3.3 + 4.4i) = (4.4 + 6.6000000000000005i)

(1.1 + 2.2i) - (3.3 + 4.4i) = (-2.1999999999999997 + -2.2i)

Take note that there are a few flaws in the design of this class, which was introduced solely for teaching purpose:

* Comparing doubles in equal() using "==" may produce unexpected outcome. For example, (2.2+4.4)==6.6 returns false. It is common to define a small threshold called EPSILON (set to about 10^-8) for comparing floating point numbers.
* The method add(), subtract(), and conjugate() produce new instances, whereas multiplyWith() and divideBy() modify this instance. There is inconsistency in the design (introduced for teaching purpose).
* Unusual to have both argumentInRadians() and argumentInDegrees().

#### 1.7  Exercise: The MyPolynomial Class



A class called MyPolynomial, which models polynomials of degree-n (see equation), is designed as shown in the class diagram.

ExerciseOOP_MyPolynomialEqn.png

The class contains:

* An instance variable named coeffs, which stores the coefficients of the n-degree polynomial in a double array of size n+1, where c0 is kept at index 0.
* A constructor MyPolynomial(coeffs:double...) that takes a variable number of doubles to initialize the coeffs array, where the first argument corresponds to c0. The three dots is known as varargs (variable number of arguments), which is a new feature introduced in JDK 1.5. It accepts an array or a sequence of comma-separated arguments. The compiler automatically packs the comma-separated arguments in an array. The three dots can only be used for the last argument of the method.  
  Hints:
* public class MyPolynomial {
* private double[] coeffs;
* public MyPolynomial(double... coeffs) { // varargs
* this.coeffs = coeffs; // varargs is treated as array
* }
* ......
* }
* // Test program
* // Can invoke with a variable number of arguments
* MyPolynomial p1 = new MyPolynomial(1.1, 2.2, 3.3);
* MyPolynomial p1 = new MyPolynomial(1.1, 2.2, 3.3, 4.4, 5.5);
* // Can also invoke with an array
* Double coeffs = {1.2, 3.4, 5.6, 7.8}

MyPolynomial p2 = new MyPolynomial(coeffs);

* Another constructor that takes coefficients from a file (of the given filename), having this format:
* Degree-n(int)
* c0(double)
* c1(double)
* ......
* ......
* cn-1(double)
* cn(double)
* (end-of-file)

Hints:

public MyPolynomial(String filename) {

Scanner in = null;

try {

in = new Scanner(new File(filename)); // open file

} catch (FileNotFoundException e) {

e.printStackTrace();

}

int degree = in.nextInt(); // read the degree

coeffs = new double[degree+1]; // allocate the array

for (int i=0; i<coeffs.length; ++i) {

coeffs[i] = in.nextDouble();

}

}

* A method getDegree() that returns the degree of this polynomial.
* A method toString() that returns "cnx^n+cn-1x^(n-1)+...+c1x+c0".
* A method evaluate(double x) that evaluate the polynomial for the given x, by substituting the given x into the polynomial expression.
* Methods add() and multiply() that adds and multiplies this polynomial with the given MyPolynomial instance another, and returns a new MyPolynomial instance that contains the result.

Write the MyPolynomial class. Also write a test program (called TestMyPolynomial) to test all the methods defined in the class.

Question: Do you need to keep the degree of the polynomial as an instance variable in the MyPolynomial class in Java? How about C/C++? Why?

#### 1.8  Exercise: Using JDK's BigInteger Class

Recall that primitive integer type byte, short, int and long represent 8-, 16-, 32-, and 64-bit signed integers, respectively. You cannot use them for integers bigger than 64 bits. Java API provides a class called BigInteger in a package called java.math. Study the API of the BigInteger class (Java API ⇒ From "Packages", choose "java.math" " From "classes", choose "BigInteger" " Study the constructors (choose "CONSTR") on how to construct a BigInteger instance, and the public methods available (choose "METHOD"). Look for methods for adding and multiplying two BigIntegers.

Write a program called TestBigInteger that:

1. adds "11111111111111111111111111111111111111111111111111111111111111" to "22222222222222222222222222222222222222222222222222" and prints the result.
2. multiplies the above two number and prints the result.

Hints:

import java.math.BigInteger

public class **TestBigInteger** {

public static void main(String[] args) {

BigInteger i1 = new BigInteger(...);

BigInteger i2 = new BigInteger(...);

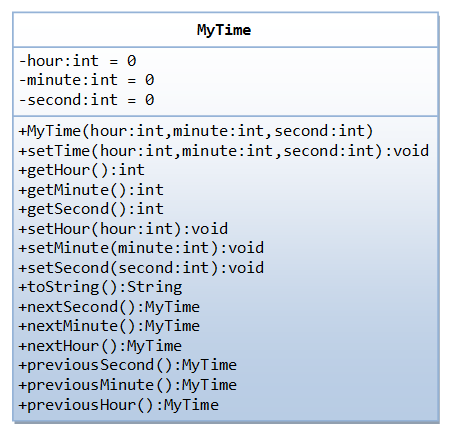
System.out.println(i1.add(i2));

.......

}

}

#### 1.9  Exercise: The MyTime Class



A class called MyTime, which models a time instance, is designed as shown in the class diagram.

It contains the following private instance variables:

* hour: between 0 to 23.
* minute: between 0 to 59.
* Second: between 0 to 59.

The constructor shall invoke the setTime() method (to be described later) to set the instance variable.

It contains the following public methods:

* setTime(int hour, int minute, int second): It shall check if the given hour, minute and second are valid before setting the instance variables.  
  (Advanced: Otherwise, it shall throw an IllegalArgumentException with the message "Invalid hour, minute, or second!".)
* Setters setHour(int hour), setMinute(int minute), setSecond(int second): It shall check if the parameters are valid, similar to the above.
* Getters getHour(), getMinute(), getSecond().
* toString(): returns "HH:MM:SS".
* nextSecond(): Update this instance to the next second and return this instance. Take note that thenextSecond() of 23:59:59 is 00:00:00.
* nextMinute(), nextHour(), previousSecond(), previousMinute(), previousHour(): similar to the above.

Write the code for the MyTime class. Also write a test program (called TestMyTime) to test all the methods defined in the MyTime class.

#### 1.10  Exercise: The MyDate Class



A class called MyDate, which models a date instance, is defined as shown in the class diagram.

The MyDate class contains the following private instance variables:

* year (int): Between 1 to 9999.
* month (int): Between 1 (Jan) to 12 (Dec).
* day (int): Between 1 to 28|29|30|31, where the last day depends on the month and whether it is a leap year for Feb (28|29).

It also contains the following private static variables (drawn with underlined in the class diagram):

* strMonths (String[]), strDays (String[]), and dayInMonths (int[]): static variables, initialized as shown, which are used in the methods.

The MyDate class has the following public static methods (drawn with underlined in the class diagram):

* isLeapYear(int year): returns true if the given year is a leap year. A year is a leap year if it is divisible by 4 but not by 100, or it is divisible by 400.
* isValidDate(int year, int month, int day): returns true if the given year, month, and day constitute a valid date. Assume that year is between 1 and 9999, month is between 1 (Jan) to 12 (Dec) and day shall be between 1 and 28|29|30|31 depending on the month and whether it is a leap year on Feb.
* getDayOfWeek(int year, int month, int day): returns the day of the week, where 0 for Sun, 1 for Mon, ..., 6 for Sat, for the given date. Assume that the date is valid. Read the [earlier exercise on how to determine the day of the week](http://www.ntu.edu.sg/home/ehchua/programming/java/J2a_BasicsExercises.html#dateutil) (or Wiki "Determination of the day of the week").

The MyDate class has one constructor, which takes 3 parameters: year, month and day. It shall invoke setDate()method (to be described later) to set the instance variables.

The MyDate class has the following public methods:

* setDate(int year, int month, int day): It shall invoke the static method isValidDate() to verify that the given year, month and day constitute a valid date.  
  (Advanced: Otherwise, it shall throw an IllegalArgumentException with the message "Invalid year, month, or day!".)
* setYear(int year): It shall verify that the given year is between 1 and 9999.  
  (Advanced: Otherwise, it shall throw an IllegalArgumentException with the message "Invalid year!".)
* setMonth(int month): It shall verify that the given month is between 1 and 12.  
  (Advanced: Otherwise, it shall throw an IllegalArgumentException with the message "Invalid month!".)
* setDay(int day): It shall verify that the given day is between 1 and dayMax, where dayMax depends on the month and whether it is a leap year for Feb.  
  (Advanced: Otherwise, it shall throw an IllegalArgumentException with the message "Invalid month!".)
* getYear(), getMonth(), getDay(): return the value for the year, month and day, respectively.
* toString(): returns a date string in the format "xxxday d mmm yyyy", e.g., "Tuesday 14 Feb 2012".
* nextDay(): update this instance to the next day and return this instance. Take note that nextDay() for 31 Dec 2000 shall be 1 Jan 2001.
* nextMonth(): update this instance to the next month and return this instance. Take note that nextMonth() for 31 Oct 2012 shall be 30 Nov 2012.
* nextYear(): update this instance to the next year and return this instance. Take note that nextYear() for 29 Feb 2012 shall be 28 Feb 2013.  
  (Advanced: throw an IllegalStateException with the message "Year out of range!" if year > 9999.)
* previousDay(), previousMonth(), previousYear(): similar to the above.

Write the code for the MyDate class.

Use the following test statements to test the MyDate class:

MyDate d1 = new MyDate(2012, 2, 28);

System.out.println(d1); // Tuesday 28 Feb 2012

System.out.println(d1.nextDay()); // Wednesday 29 Feb 2012

System.out.println(d1.nextDay()); // Thursday 1 Mar 2012

System.out.println(d1.nextMonth()); // Sunday 1 Apr 2012

System.out.println(d1.nextYear()); // Monday 1 Apr 2013

MyDate d2 = new MyDate(2012, 1, 2);

System.out.println(d2); // Monday 2 Jan 2012

System.out.println(d2.previousDay()); // Sunday 1 Jan 2012

System.out.println(d2.previousDay()); // Saturday 31 Dec 2011

System.out.println(d2.previousMonth()); // Wednesday 30 Nov 2011

System.out.println(d2.previousYear()); // Tuesday 30 Nov 2010

MyDate d3 = new MyDate(2012, 2, 29);

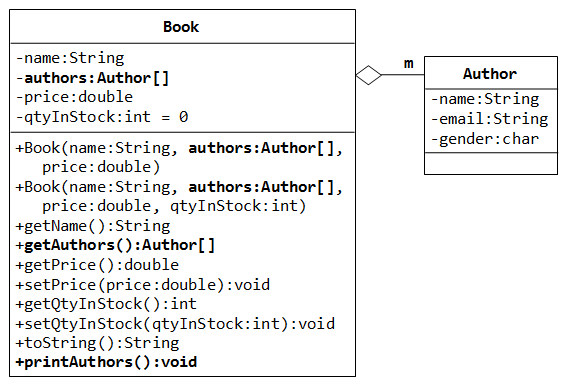
System.out.println(d3.previousYear()); // Monday 28 Feb 2011

// MyDate d4 = new MyDate(2099, 11, 31); // Invalid year, month, or day!

// MyDate d5 = new MyDate(2011, 2, 29); // Invalid year, month, or day!

Write a test program that tests the nextDay() in a loop, by printing the dates from 28 Dec 2011 to 2 Mar 2012.

#### 1.11  Exercise: Book and Author Classes Again - An Array of Objects as an Instance Variable



In the [earlier exercise](http://www.ntu.edu.sg/home/ehchua/programming/java/J3f_OOPExercises.html#bookauthor), a book is written by one and only one author. In reality, a book can be written by one or more author. Modify the Book class to support one or more authors by changing the instance variable authors to an Author array. Reuse the Author class written earlier.

Notes:

* The constructors take an array of Author (i.e., Author[]), instead of an Author instance.
* The toString() method shall return "book-name by n authors", where n is the number of authors.
* A new method printAuthors() to print the names of all the authors.

You are required to:

1. Write the code for the Book class. You shall re-use the Author class written earlier.
2. Write a test program (called TestBook) to test the Book class.

Hints:

// Declare and allocate an array of Authors

Author[] authors = new Author[2];

authors[0] = new Author("Tan Ah Teck", "AhTeck@somewhere.com", 'm');

authors[1] = new Author("Paul Tan", "Paul@nowhere.com", 'm');

// Declare and allocate a Book instance

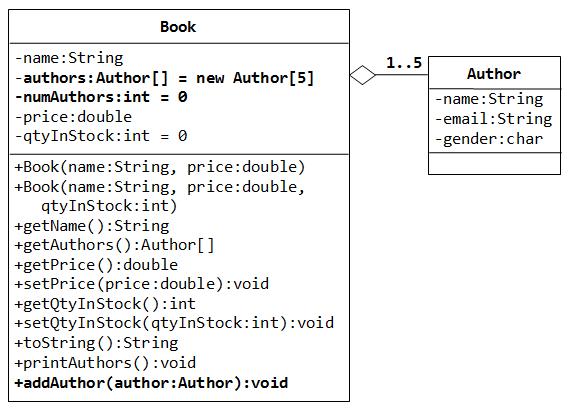
Book javaDummy = new Book("Java for Dummy", authors, 19.99, 99);

System.out.println(javaDummy); // toString()

System.out.print("The authors are: ");

javaDummy.printAuthors();

#### 1.12  Exercise: Book and Author Classes Once More - A Fixed-length Array of Objects as an Instance Variable



In the above exercise, the number of authors cannot be changed once a Book instance is constructed. Suppose that we wish to allow the user to add more authors (which is really unusual but presented here for academic purpose).

We shall remove the authors from the constructors, and add a new method called addAuthor()to add the given Author instance to this Book.

We also need to pre-allocate an Author array, with a fixed length (says 5 - a book is written by 1 to 5 authors), and use another instance variable numAuthors (int) to keep track of the actual number of authors.

You are required to:

1. Modify your Book class to support this new requirement.  
   Hints:
2. public class Book {
3. // private instance variable
4. private Author[] authors = new Author[5]; // declare and allocate the array
5. // BUT not the element's instance
6. private int numAuthors = 0;
7. ......
8. ......
9. public void addAuthor(Author author) {
10. authors[numAuthors] = author;
11. ++numAuthors;
12. }
13. }
15. // Test program
16. Book javaDummy = new Book("Java for Dummy", 19.99, 99);
17. System.out.println(javaDummy); // toString()
18. System.out.print("The authors are: ");
19. javaDummy.printAuthors();
21. javaDummy.addAuthor(new Author("Tan Ah Teck", "AhTeck@somewhere.com", 'm'));
22. javaDummy.addAuthor(new Author("Paul Tan", "Paul@nowhere.com", 'm'));
23. System.out.println(javaDummy); // toString()
24. System.out.print("The authors are: ");

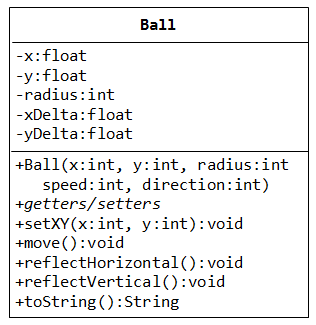
javaDummy.printAuthors();

1. Try writing a method called removeAuthorByName(authorName), that remove the author from this Book instance if authorName is present. The method shall return true if it succeeds.

boolean removeAuthorByName(String authorName)

Advanced Note: Instead of using a fixed-length array in this case, it is better to be a dynamically allocated array (e.g., ArrayList), which does not have a fixed length.

#### 1.13  Exercise: Bouncing Balls - Ball and Container Classes

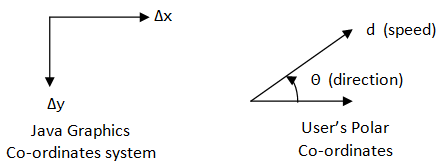


A class called Ball is designed as shown in the class diagram.

The Ball class contains the following private instance variables:

* x, y and radius, which represent the ball's center (x, y) co-ordinates and the radius, respectively.
* xDelta (Δx) and yDelta (Δy), which represent the displacement (movement) per step, in the x and y direction respectively.

The Ball class contains the following public methods:

* A constructor which accepts x, y, radius, speed, and direction as arguments. For user friendliness, user specifies speed (in pixels per step) and direction (in degrees in the range of (-180°, 180°]). For the internal operations, the speed and direction are to be converted to (Δx, Δy) in the internal representation. Note that the y-axis of the Java graphics coordinate system is inverted, i.e., the origin (0, 0) is located at the top-left corner.  
  
* Δx = d × cos(θ)

Δy = -d × sin(θ)

* Getter and setter for all the instance variables.
* A method move() which move the ball by one step.
* x += Δx

y += Δy

* reflectHorizontal() which reflects the ball horizontally (i.e., hitting a vertical wall)
* Δx = -Δx

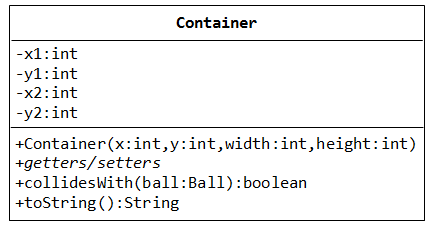
Δy no changes

* reflectVertical() (the ball hits a horizontal wall).
* Δx no changes

Δy = -Δy

* toString() which prints the message "Ball at (x, y) of velocity (Δx, Δy)".

Write the Ball class. Also write a test program to test all the methods defined in the class.



A class called Container, which represents the enclosing box for the ball, is designed as shown in the class diagram. It contains:

* Instance variables (x1, y1) and (x2, y2) which denote the top-left and bottom-right corners of the rectangular box.
* A constructor which accepts (x, y) of the top-left corner, width and height as argument, and converts them into the internal representation (i.e., x2=x1+width-1). Width and height is used in the argument for safer operation (there is no need to check the validity of x2>x1 etc.).
* A toString() method that returns "Container at (x1,y1) to (x2, y2)".
* A boolean method called collidesWith(Ball), which check if the given Ball is outside the bounds of the container box. If so, it invokes the Ball's reflectHorizontal() and/or reflectVertical() to change the movement direction of the ball, and returns true.
* public boolean collidesWith(Ball ball) {
* if (ball.getX() - ball.getRadius() <= this.x1 ||
* ball.getX() - ball.getRadius() >= this.x2) {
* ball.reflectHorizontal();
* return true;
* }
* ......

}

Use the following statements to test your program:

Ball ball = new Ball(50, 50, 5, 10, 30);

Container box = new Container(0, 0, 100, 100);

for (int step = 0; step < 100; ++step) {

ball.move();

box.collidesWith(ball);

System.out.println(ball); // manual check the position of the ball

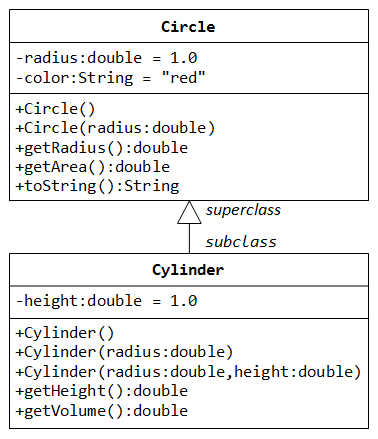
}

#### 1.14  Exercise: Ball and Player

[TODO]

### 2.  Exercises on Inheritance

#### 2.1  Exercise: The Circle and Cylinder Classes



In this exercise, a subclass called Cylinder is derived from the superclass Circle as shown in the class diagram (where an an arrow pointing up from the subclass to its superclass). Study how the subclass Cylinder invokes the superclass' constructors (viasuper() and super(radius)) and inherits the variables and methods from the superclass Circle.

You can reuse the Circle class that you have created in the previous exercise. Make sure that you keep "Circle.class" in the same directory.

public class **Cylinder extends Circle** { //save as "Cylinder.java"

private double height; // private variable

// Constructor with default color, radius and height

public Cylinder() {

super(); // call superclass no-arg constructor Circle()

height = 1.0;

}

// Constructor with default radius, color but given height

public Cylinder(double height) {

super(); // call superclass no-arg constructor Circle()

this.height = height;

}

// Constructor with default color, but given radius, height

public Cylinder(double radius, double height) {

super(radius); // call superclass constructor Circle(r)

this.height = height;

}

// A public method for retrieving the height

public double getHeight() {

return height;

}

// A public method for computing the volume of cylinder

// use superclass method getArea() to get the base area

public double getVolume() {

return getArea()\*height;

}

}

Write a test program (says TestCylinder) to test the Cylinder class created, as follow:

public class **TestCylinder** { // save as "TestCylinder.java"

public static void main (String[] args) {

// Declare and allocate a new instance of cylinder

// with default color, radius, and height

Cylinder c1 = new Cylinder();

System.out.println("Cylinder:"

+ " radius=" + c1.getRadius()

+ " height=" + c1.getHeight()

+ " base area=" + c1.getArea()

+ " volume=" + c1.getVolume());

// Declare and allocate a new instance of cylinder

// specifying height, with default color and radius

Cylinder c2 = new Cylinder(10.0);

System.out.println("Cylinder:"

+ " radius=" + c2.getRadius()

+ " height=" + c2.getHeight()

+ " base area=" + c2.getArea()

+ " volume=" + c2.getVolume());

// Declare and allocate a new instance of cylinder

// specifying radius and height, with default color

Cylinder c3 = new Cylinder(2.0, 10.0);

System.out.println("Cylinder:"

+ " radius=" + c3.getRadius()

+ " height=" + c3.getHeight()

+ " base area=" + c3.getArea()

+ " volume=" + c3.getVolume());

}

}

**Method Overriding and "Super":** The subclass Cylinder inherits getArea() method from its superclass Circle. Try overriding the getArea() method in the subclass Cylinder to compute the surface area (=2π×radius×height + 2×base-area) of the cylinder instead of base area. That is, if getArea() is called by a Circle instance, it returns the area. If getArea() is called by a Cylinder instance, it returns the surface area of the cylinder.

If you override the getArea() in the subclass Cylinder, the getVolume() no longer works. This is because the getVolume() uses the overridden getArea() method found in the same class. (Java runtime will search the superclass only if it cannot locate the method in this class). Fix the getVolume().

Hints: After overridding the getArea() in subclass Cylinder, you can choose to invoke the getArea() of the superclass Circle by calling super.getArea().

TRY:

Provide a toString() method to the Cylinder class, which overrides the toString() inherited from the superclass Circle, e.g.,

@Override

public String toString() { // in Cylinder class

return "Cylinder: subclass of " + super.toString() // use Circle's toString()

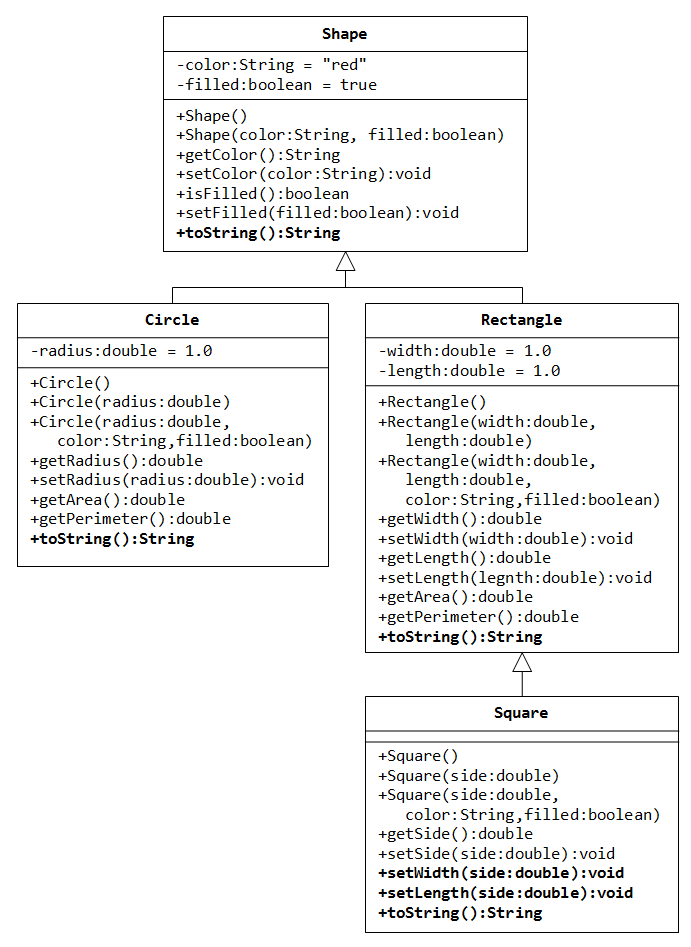
+ " height=" + height;

}

Try out the toString() method in TestCylinder.

Note: @Override is known as annotation (introduced in JDK 1.5), which asks compiler to check whether there is such a method in the superclass to be overridden. This helps greatly if you misspell the name of the toString(). If @Override is not used and toString() is misspelled as ToString(), it will be treated as a new method in the subclass, instead of overriding the superclass. If @Override is used, the compiler will signal an error. @Override annotation is optional, but certainly nice to have.

#### 2.2  Exercise: Superclass Shape and its subclasses Circle, Rectangle and Square



Write a superclass called Shape (as shown in the class diagram), which contains:

* Two instance variables color (String) and filled (boolean).
* Two constructors: a no-arg (no-argument) constructor that initializes the color to "green" and filled to true, and a constructor that initializes the color and filled to the given values.
* Getter and setter for all the instance variables. By convention, the getter for a boolean variable xxx is called isXXX() (instead of getXxx() for all the other types).
* A toString() method that returns "A Shape with color of xxx and filled/Not filled".

Write a test program to test all the methods defined in Shape.

Write two subclasses of Shape called Circle and Rectangle, as shown in the class diagram.

The Circle class contains:

* An instance variable radius (double).
* Three constructors as shown. The no-arg constructor initializes the radius to 1.0.
* Getter and setter for the instance variable radius.
* Methods getArea() and getPerimeter().
* Override the toString() method inherited, to return "A Circle with radius=xxx, which is a subclass of yyy", where yyy is the output of the toString() method from the superclass.

The Rectangle class contains:

* Two instance variables width (double) and length (double).
* Three constructors as shown. The no-arg constructor initializes the width and length to 1.0.
* Getter and setter for all the instance variables.
* Methods getArea() and getPerimeter().
* Override the toString() method inherited, to return "A Rectangle with width=xxx and length=zzz, which is a subclass of yyy", where yyy is the output of the toString()method from the superclass.

Write a class called Square, as a subclass of Rectangle. Convince yourself that Square can be modeled as a subclass of Rectangle. Square has no instance variable, but inherits the instance variables width and length from its superclass Rectangle.

* Provide the appropriate constructors (as shown in the class diagram). Hint:
* public Square(double side) {
* super(side, side); // Call superclass Rectangle(double, double)

}

* Override the toString() method to return "A Square with side=xxx, which is a subclass of yyy", where yyy is the output of the toString() method from the superclass.
* Do you need to override the getArea() and getPerimeter()? Try them out.
* Override the setLength() and setWidth() to change both the width and length, so as to maintain the square geometry.

### 3.  Exercises on Composition vs Inheritance

They are two ways to reuse a class in your applications: composition and inheritance.

#### 3.1  Exercise: The Point and Line Classes

Let us begin with composition with the statement "a line composes of two points".

Complete the definition of the following two classes: Point and Line. The class Line composes 2 instances of class Point, representing the beginning and ending points of the line. Also write test classes for Point and Line (says TestPoint and TestLine).

public class **Point** {

// Private variables

private int x; // x co-ordinate

private int y; // y co-ordinate

// Constructor

public Point (int x, int y) {......}

// Public methods

public String toString() {

return "Point: (" + x + "," + y + ")";

}

public int getX() {......}

public int getY() {......}

public void setX(int x) {......}

public void setY(int y) {......}

public void setXY(int x, int y) {......}

}

public class **TestPoint** {

public static void main(String[] args) {

Point p1 = new Point(10, 20); // Construct a Point

System.out.println(p1);

// Try setting p1 to (100, 10).

......

}

}

public class **Line** {

// A line composes of two points (as instance variables)

private Point begin; // beginning point

private Point end; // ending point

// Constructors

public Line (Point begin, Point end) { // caller to construct the Points

this.begin = begin;

......

}

public Line (int beginX, int beginY, int endX, int endY) {

begin = new Point(beginX, beginY); // construct the Points here

......

}

// Public methods

public String toString() { ...... }

public Point getBegin() { ...... }

public Point getEnd() { ...... }

public void setBegin(......) { ...... }

public void setEnd(......) { ...... }

public int getBeginX() { ...... }

public int getBeginY() { ...... }

public int getEndX() { ...... }

public int getEndY() { ...... }

public void setBeginX(......) { ...... }

public void setBeginY(......) { ...... }

public void setBeginXY(......) { ...... }

public void setEndX(......) { ...... }

public void setEndY(......) { ...... }

public void setEndXY(......) { ...... }

public int getLength() { ...... } // Length of the line

// Math.sqrt(xDiff\*xDiff + yDiff\*yDiff)

public double getGradient() { ...... } // Gradient in radians

// Math.atan2(yDiff, xDiff)

}

public class **TestLine** {

public static void main(String[] args) {

Line l1 = new Line(0, 0, 3, 4);

System.out.println(l1);

Point p1 = new Point(...);

Point p2 = new Point(...);

Line l2 = new Line(p1, p2);

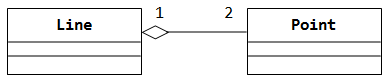
System.out.println(l2);

...

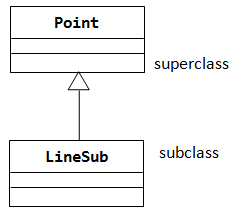
}

}

The class diagram for composition is as follows (where a diamond-hollow-head arrow pointing to its constituents):



Instead of composition, we can design a Line class using inheritance. Instead of "a line composes of two points", we can say that "a line is a point extended by another point", as shown in the following class diagram:



Let's re-design the Line class (called LineSub) as a subclass of class Point. LineSub inherits the starting point from its superclass Point, and adds an ending point. Complete the class definition. Write a testing class called TestLineSub to test LineSub.

public class **LineSub extends Point** {

// A line needs two points: begin and end.

// The begin point is inherited from its superclass Point.

// Private variables

Point end; // Ending point

// Constructors

public LineSub (int beginX, int beginY, int endX, int endY) {

super(beginX, beginY); // construct the begin Point

this.end = new Point(endX, endY); // construct the end Point

}

public LineSub (Point begin, Point end) { // caller to construct the Points

super(begin.getX(), begin.getY()); // need to reconstruct the begin Point

this.end = end;

}

// Public methods

// Inherits methods getX() and getY() from superclass Point

public String toString() { ... }

public Point getBegin() { ... }

public Point getEnd() { ... }

public void setBegin(...) { ... }

public void setEnd(...) { ... }

public int getBeginX() { ... }

public int getBeginY() { ... }

public int getEndX() { ... }

public int getEndY() { ... }

public void setBeginX(...) { ... }

public void setBeginY(...) { ... }

public void setBeginXY(...) { ... }

public void setEndX(...) { ... }

public void setEndY(...) { ... }

public void setEndXY(...) { ... }

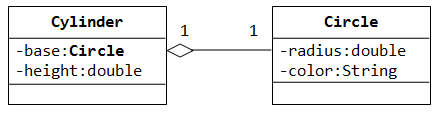
public int getLength() { ... } // Length of the line

public double getGradient() { ... } // Gradient in radians

}

Summary: There are two approaches that you can design a line, composition or inheritance. "A line composes two points" or "A line is a point extended with another point"”. Compare theLine and LineSub designs: Line uses composition and LineSub uses inheritance. Which design is better?

#### 3.2  Exercise: Circle and Cylinder using Composition



Try rewriting the Circle-Cylinder of the previous exercise using composition (as shown in the class diagram) instead of inheritance. That is, "a cylinder is composed of a base circle and a height".

public class **Cylinder** {

private Circle base; // Base circle, an instance of Circle class

private double height;

// Constructor with default color, radius and height

public Cylinder() {

base = new Circle(); // Call the constructor to construct the Circle

height = 1.0;

}

......

}

Which design (inheritance or composition) is better?

### 4.  Exercises on Polymorphism, Abstract Classes and Interfaces

#### 4.1  Exercise: Abstract Superclass Shape and Its Concrete Subclasses

Rewrite the superclass Shape and its subclasses Circle, Rectangle and Square, as shown in the class diagram.



In this exercise, Shape shall be defined as an abstract class, which contains:

* Two protected instance variables color(String) and filled(boolean). The protected variables can be accessed by its subclasses and classes in the same package. They are denoted with a '#' sign in the class diagram.
* Getter and setter for all the instance variables, and toString().
* Two abstract methods getArea() and getPerimeter() (shown in italics in the class diagram).

The subclasses Circle and Rectangle shall override the abstract methods getArea() and getPerimeter() and provide the proper implementation. They also override the toString().

Write a test class to test these statements involving polymorphism and explain the outputs. Some statements may trigger compilation errors. Explain the errors, if any.

Shape s1 = new Circle(5.5, "RED", false); // Upcast Circle to Shape

System.out.println(s1); // which version?

System.out.println(s1.getArea()); // which version?

System.out.println(s1.getPerimeter()); // which version?

System.out.println(s1.getColor());

System.out.println(s1.isFilled());

System.out.println(s1.getRadius());

Circle c1 = (Circle)s1; // Downcast back to Circle

System.out.println(c1);

System.out.println(c1.getArea());

System.out.println(c1.getPerimeter());

System.out.println(c1.getColor());

System.out.println(c1.isFilled());

System.out.println(c1.getRadius());

Shape s2 = new Shape();

Shape s3 = new Rectangle(1.0, 2.0, "RED", false); // Upcast

System.out.println(s3);

System.out.println(s3.getArea());

System.out.println(s3.getPerimeter());

System.out.println(s3.getColor());

System.out.println(s3.getLength());

Rectangle r1 = (Rectangle)s3; // downcast

System.out.println(r1);

System.out.println(r1.getArea());

System.out.println(r1.getColor());

System.out.println(r1.getLength());

Shape s4 = new Square(6.6); // Upcast

System.out.println(s4);

System.out.println(s4.getArea());

System.out.println(s4.getColor());

System.out.println(s4.getSide());

// Take note that we downcast Shape s4 to Rectangle,

// which is a superclass of Square, instead of Square

Rectangle r2 = (Rectangle)s4;

System.out.println(r2);

System.out.println(r2.getArea());

System.out.println(r2.getColor());

System.out.println(r2.getSide());

System.out.println(r2.getLength());

// Downcast Rectangle r2 to Square

Square sq1 = (Square)r2;

System.out.println(sq1);

System.out.println(sq1.getArea());

System.out.println(sq1.getColor());

System.out.println(sq1.getSide());

System.out.println(sq1.getLength());

What is the usage of the abstract method and abstract class?

#### 4.2  Exercise: Polymorphism

Examine the following codes and draw the class diagram.

abstract public class **Animal** {

abstract public void greeting();

}

public class **Cat extends Animal** {

@Override

public void greeting() {

System.out.println("Meow!");

}

}

public class **Dog extends Animal** {

@Override

public void greeting() {

System.out.println("Woof!");

}

public void greeting(Dog another) {

System.out.println("Woooooooooof!");

}

}

public class **BigDog extends Dog** {

@Override

public void greeting() {

System.out.println("Woow!");

}

@Override

public void greeting(Dog another) {

System.out.println("Woooooowwwww!");

}

}

Explain the outputs (or error) for the following test program.

public class **TestAnimal** {

public static void main(String[] args) {

// Using the subclasses

Cat cat1 = new Cat();

cat1.greeting();

Dog dog1 = new Dog();

dog1.greeting();

BigDog bigDog1 = new BigDog();

bigDog1.greeting();

// Using Polymorphism

Animal animal1 = new Cat();

animal1.greeting();

Animal animal2 = new Dog();

animal2.greeting();

Animal animal3 = new BigDog();

animal3.greeting();

Animal animal4 = new Animal();

// Downcast

Dog dog2 = (Dog)animal2;

BigDog bigDog2 = (BigDog)animal3;

Dog dog3 = (Dog)animal3;

Cat cat2 = (Cat)animal2;

dog2.greeting(dog3);

dog3.greeting(dog2);

dog2.greeting(bigDog2);

bigDog2.greeting(dog2);

bigDog2.greeting(bigDog1);

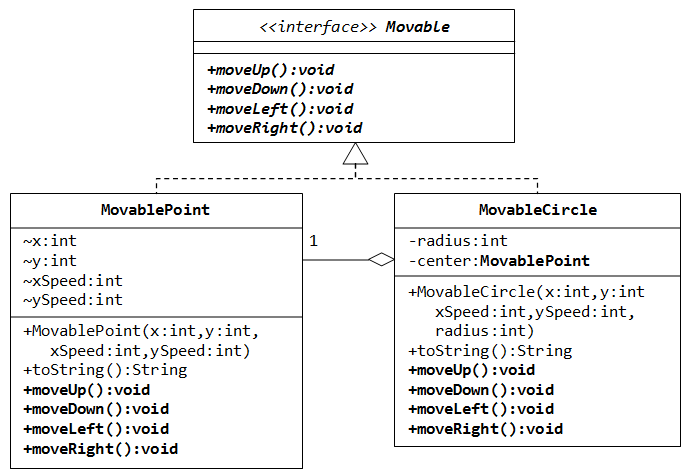
}

}

#### 4.3  Exercise: Interface Movable and its implementations MovablePoint and MovableCircle

Suppose that we have a set of objects with some common behaviors: they could move up, down, left or right. The exact behaviors (such as how to move and how far to move) depend on the objects themselves. One common way to model these common behaviors is to define an interface called Movable, with abstract methods moveUp(), moveDown(), moveLeft() andmoveRight(). The classes that implement the Movable interface will provide actual implementation to these abstract methods.

Let's write two concrete classes - MovablePoint and MovableCircle - that implement the Movable interface.



The code for the interface Movable is straight forward.

public interface **Movable** { // saved as "Movable.java"

public void moveUp();

......

}

For the MovablePoint class, declare the instance variable x, y, xSpeed and ySpeed with package access as shown with '~' in the class diagram (i.e., classes in the same package can access these variables directly). For the MovableCircle class, use a MovablePoint to represent its center (which contains four variable x, y, xSpeed and ySpeed). In other words, the MovableCirclecomposes a MovablePoint, and its radius.

public class **MovablePoint implements Movable** { // saved as "MovablePoint.java"

// instance variables

int x, y, xSpeed, ySpeed; // package access

// Constructor

public MovablePoint(int x, int y, int xSpeed, int ySpeed) {

this.x = x;

......

}

......

// Implement abstract methods declared in the interface Movable

@Override

public void moveUp() {

y -= ySpeed; // y-axis pointing down for 2D graphics

}

......

}

public class **MovableCircle implements Movable** { // saved as "MovableCircle.java"

// instance variables

private MovablePoint center; // can use center.x, center.y directly

// because they are package accessible

private int radius;

// Constructor

public MovableCircle(int x, int y, int xSpeed, int ySpeed, int radius) {

// Call the MovablePoint's constructor to allocate the center instance.

center = new MovablePoint(x, y, xSpeed, ySpeed);

......

}

......

// Implement abstract methods declared in the interface Movable

@Override

public void moveUp() {

center.y -= center.ySpeed;

}

......

}

Write a test program and try out these statements:

Movable m1 = new MovablePoint(5, 6, 10); // upcast

System.out.println(m1);

m1.moveLeft();

System.out.println(m1);

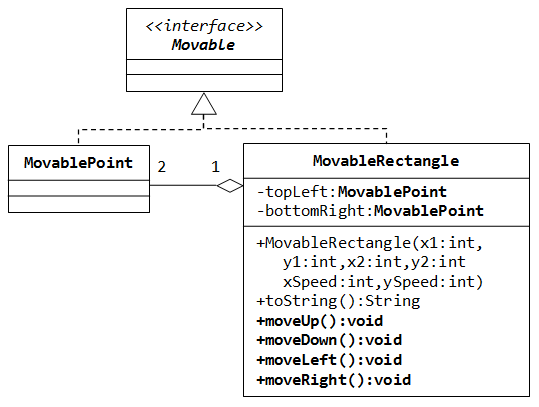
Movable m2 = new MovableCircle(2, 1, 2, 20); // upcast

System.out.println(m2);

m2.moveRight();

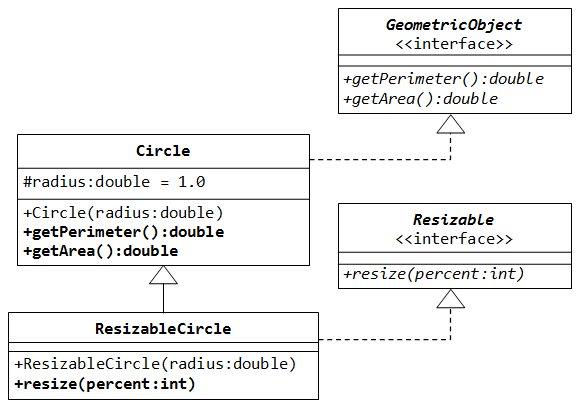
System.out.println(m2);

Write a new class called MovableRectangle, which composes two MovablePoints (representing the top-left and bottom-right corners) and implementing the Movable Interface. Make sure that the two points has the same speed.



What is the difference between an interface and an abstract class?

#### 4.4  Exercise: Interfaces GeometricObject and Resizable



1. Write the interface called GeometricObject, which declares two abstract methods: getParameter() and getArea(), as specified in the class diagram.  
   Hints:
2. public interface **GeometricObject** {
3. public double getPerimeter();
4. ......

}

1. Write the implementation class Circle, with a protected variable radius, which implements the interface GeometricObject.  
   Hints:
2. public class **Circle** implements GeometricObject {
3. // Private variable
4. ......
5. // Constructor
6. ......
7. // Implement methods defined in the interface GeometricObject
8. @Override
9. public double getPerimeter() { ...... }
10. ......

}

1. Write a test program called TestCircle to test the methods defined in Circle.
2. The class ResizableCircle is defined as a subclass of the class Circle, which also implements an interface called Resizable, as shown in class diagram. The interface Resizabledeclares an abstract method resize(), which modifies the dimension (such as radius) by the given percentage. Write the interface Resizable and the class ResizableCircle.  
   Hints:
3. public interface Resizable {
4. public double resize(...);

}

public class ResizableCircle extends Circle implements Resizeable {

// Constructor

public ResizableCircle(double radius) {

super(...);

}

// Implement methods defined in the interface Resizable

@Override

public double resize(int percent) { ...... }

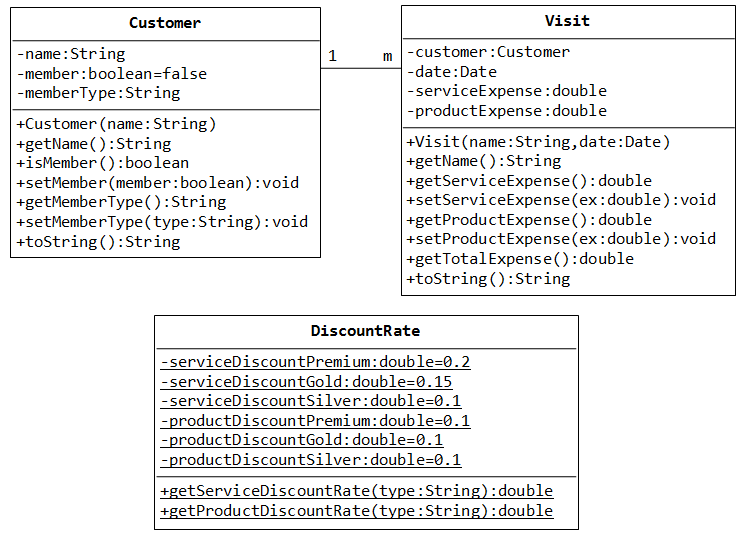
}

1. Write a test program called TestResizableCircle to test the methods defined in ResizableCircle.

### 5.  More Exercises on OOP

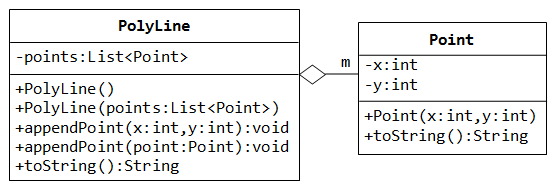
#### 5.1  Exercise: The Discount System

You are asked to write a discount system for a beauty saloon, which provides services and sells beauty products. It offers 3 types of memberships: Premium, Gold and Silver. Premium, gold and silver members receive a discount of 20%, 15%, and 10%, respectively, for all services provided. Customers without membership receive no discount. All members receives a flat 10% discount on products purchased (this might change in future). Your system shall consist of three classes: Customer, Discount and Visit, as shown in the class diagram. It shall compute the total bill if a customer purchases $x of products and $y of services, for a visit. Also write a test program to exercise all the classes.



The class DiscountRate contains only static variables and methods (underlined in the class diagram).

#### 5.2  Exercise: Polyline of Points with ArrayList



A polyline is a line with segments formed by points. Let's use the ArrayList (dynamically allocated array) to keep the points, but upcast to List in the instance variable. (Take note that array is of fixed-length, and you need to set the initial length).

public class **Point** {

private int x;

private int y;

public Point(int x, int y) { ...... }

public String toString() { ...... }

}

import java.util.\*;

public class **PolyLine** {

private List<Point> points = new ArrayList<Point>();

// Allocate an ArrayList of Points and upcast to List

public PolyLine() { } // default constructor

public PolyLine(List<Point> points) {

this.points = points;

}

// Append a point at (x, y) to the end of this polyline

public void appendPoint(int x, int y) {

Point newPoint = new Point(x, y);

points.add(newPoint);

}

// Append a point instance to the end of this polyline

public void appendPoint(Point point) {

points.add(point);

}

// return (x1,y1)(x2,y2)(x3,y3)....

public String toString() {

StringBuilder sb = new StringBuilder();

for (Point aPoint : points) {

sb.append(aPoint.toString());

}

return sb.toString();

}

}

public class **TestPolyLine** {

public static void main(String[] args) {

PolyLine l1 = new PolyLine();

System.out.println(l1); // empty

l1.appendPoint(new Point(1, 1));

l1.appendPoint(2, 2);

l1.appendPoint(3, 3);

System.out.println(l1); // (1,1)(2,2)(3,3)

}

}

### 6.  Exercises on Data Structure and Algorithm

#### 6.1  Exercise: MyIntStack

A stack is a first-in-last-out queue. Write a program calld MyIntStack, which uses an array to store the contents, restricted to int.

Write a test program.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30 | public class MyIntStack {  private int[] contents;  private int tos; // Top of the stack    // constructors  public MyIntStack(int capacity) {  contents = new int[capacity];  tos = -1;  }    public void push(int element) {  contents[++tos] = element;  }    public int pop() {  return contents[tos--];  }    public int peek() {  return contents[tos];  }    public boolean isEmpty() {  return tos < 0;  }    public boolean isFull() {  return tos == contents.length - 1;  }  } |

Try:

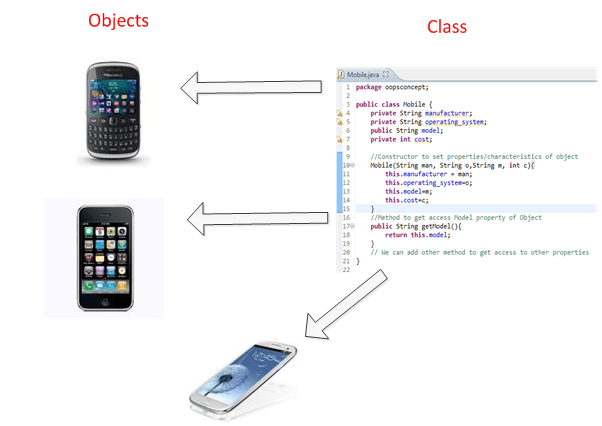
1. Modify the push() method to throw an IllegalStateException if the stack is full.
2. Modify the push() to return true if the operation is sucessful, or false otherwise.
3. Modify the push() to increase the capacity by reallocating another array, if the stack is full.

Topic 1: **Features of OOPs:- Encapsulation, Abstraction, Inheritance, Polymorphism**    :

## Introduction

This tutorial will help you to understand about Java OOP’S concepts with examples. Let’s discuss about what are the features of Object Oriented Programming. Writing object-oriented programs involves creating classes, creating objects from those classes, and creating applications, which are stand-alone executable programs that use those objects.

A class is a template, blueprint,or contract that defines what an object’s data fields and methods will be. An object is an instance of a class. You can create many instances of a class. A Java class uses variables to define data fields and methods to define actions. Additionally,a class provides methods of a special type, known as constructors, which are invoked to create a new object. A constructor can perform any action, but constructors are designed to perform initializing actions, such as initializing the data fields of objects.



Objects are made up of attributes and methods. Attributes are the characteristics that define an object; the values contained in attributes differentiate objects of the same class from one another. To understand this better let’s take example of Mobile as object. Mobile has characteristics like model, manufacturer, cost, operating system etc. So if we create “Samsung” mobile object and “IPhone” mobile object we can distinguish them from characteristics. The values of the attributes of an object are also referred to as the object’s state.

There are three main features of OOPS.

**1)** Encapsulation

**2)** Inheritance

**3)** Polymorphism

Let’s we discuss about the about features in details.

## Encapsulation

Encapsulation means putting together all the variables (instance variables) and the methods into a single unit called Class. It also means hiding data and methods within an Object. Encapsulation provides the security that keeps data and methods safe from inadvertent changes. Programmers sometimes refer to encapsulation as using a “black box,” or a device that you can use without regard to the internal mechanisms. A programmer can access and use the methods and data contained in the black box but cannot change them. Below example shows Mobile class with properties, which can be set once while creating object using constructor arguments. Properties can be accessed using getXXX() methods which are having public access modifiers.

[view plaincopy to clipboardprint?](http://www.w3resource.com/java-tutorial/java-object-oriented-programming.php)

1. **package** oopsconcept;
2. **public** **class** Mobile {
3. **private** String manufacturer;
4. **private** String operating\_system;
5. **public** String model;
6. **private** **int** cost;
7. //Constructor to set properties/characteristics of object
8. Mobile(String man, String o,String m, **int** c){
9. **this**.manufacturer = man;
10. **this**.operating\_system=o;
11. **this**.model=m;
12. **this**.cost=c;
13. }
14. //Method to get access Model property of Object
15. **public** String getModel(){
16. **return** **this**.model;
17. }
18. // We can add other method to get access to other properties
19. }

## Inheritance

An important feature of object-oriented programs is inheritance—the ability to create classes that share the attributes and methods of existing classes, but with more specific features. Inheritance is mainly used for code reusability. So you are making use of already written class and further extending on that. That why we discussed about the code reusability the concept. In general one line definition we can tell that deriving a new class from existing class, it’s called as Inheritance. You can look into the following example for inheritance concept. Here we have Mobile class extended by other specific class like Android and Blackberry.

[view plaincopy to clipboardprint?](http://www.w3resource.com/java-tutorial/java-object-oriented-programming.php)

1. **package** oopsconcept;
2. **public** **class** Android **extends** Mobile{
3. //Constructor to set properties/characteristics of object
4. Android(String man, String o,String m, **int** c){
5. **super**(man, o, m, c);
6. }
7. //Method to get access Model property of Object
8. **public** String getModel(){
9. **return** "This is Android Mobile- " + model;
10. }
11. }

[view plaincopy to clipboardprint?](http://www.w3resource.com/java-tutorial/java-object-oriented-programming.php)

1. **package** oopsconcept;
2. **public** **class** Blackberry **extends** Mobile{
3. //Constructor to set properties/characteristics of object
4. Blackberry(String man, String o,String m, **int** c){
5. **super**(man, o, m, c);
6. }
7. **public** String getModel(){
8. **return** "This is Blackberry-"+ model;
9. }
10. }

## Polymorphism

In Core Java Polymorphism is one of easy concept to understand. Polymorphism definition is that Poly means many and morphos means forms. It describes the feature of languages that allows the same word or symbol to be interpreted correctly in different situations based on the context. There are two types of Polymorphism available in Java. For example, in English the verb “run” means different things if you use it with “a footrace,” a “business,” or “a computer.” You understand the meaning of “run” based on the other words used with it. Object-oriented programs are written so that the methods having same name works differently in different context. Java provides two ways to implement polymorphism.

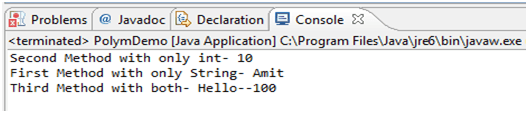
## Static Polymorphism (compile time polymorphism/ Method overloading):

The ability to execute different method implementations by altering the argument used with the method name is known as method overloading. In below program we have three print methods each with different arguments. When you properly overload a method, you can call it providing different argument lists, and the appropriate version of the method executes.

[view plaincopy to clipboardprint?](http://www.w3resource.com/java-tutorial/java-object-oriented-programming.php)

1. **package** oopsconcept;
2. **class** Overloadsample {
3. **public** **void** print(String s){
4. System.out.println("First Method with only String- "+ s);
5. }
6. **public** **void** print (**int** i){
7. System.out.println("Second Method with only int- "+ i);
8. }
9. **public** **void** print (String s, **int** i){
10. System.out.println("Third Method with both- "+ s + "--" + i);
11. }
12. }
13. **public** **class** PolymDemo {
14. **public** **static** **void** main(String[] args) {
15. Overloadsample obj = **new** Overloadsample();
16. obj.print(10);
17. obj.print("Amit");
18. obj.print("Hello", 100);
19. }
20. }

## Output :



## Dynamic Polymorphism (run time polymorphism/ Method Overriding)

When you create a subclass by extending an existing class, the new subclass contains data and methods that were defined in the original superclass. In other words, any child class object has all the attributes of its parent. Sometimes, however, the superclass data fields and methods are not entirely appropriate for the subclass objects; in these cases, you want to override the parent class members. Let’s take example used in inheritance explanation.

[view plaincopy to clipboardprint?](http://www.w3resource.com/java-tutorial/java-object-oriented-programming.php)

1. **package** oopsconcept;
2. **public** **class** OverridingDemo {
3. **public** **static** **void** main(String[] args) {
4. //Creating Object of SuperClass and calling getModel Method
5. Mobile m = **new** Mobile("Nokia", "Win8", "Lumia",10000);
6. System.out.println(m.getModel());
7. //Creating Object of Sublcass and calling getModel Method
8. Android a = **new** Android("Samsung", "Android", "Grand",30000);
9. System.out.println(a.getModel());
10. //Creating Object of Sublcass and calling getModel Method
11. Blackberry b = **new** Blackberry("BlackB", "RIM", "Curve",20000);
12. System.out.println(b.getModel());
13. }
14. }

## Abstraction

All programming languages provide abstractions. It can be argued that the complexity of the problems you’re able to solve is directly related to the kind and quality of abstraction. An essential element of object-oriented programming is abstraction. Humans manage complexity through abstraction. When you drive your car you do not have to be concerned with the exact internal working of your car(unless you are a mechanic). What you are concerned with is interacting with your car via its interfaces like steering wheel, brake pedal, accelerator pedal etc. Various manufacturers  of car has different implementation of car working but its basic interface has not changed (i.e. you still use steering wheel, brake pedal, accelerator pedal etc to interact with your car). Hence the knowledge you have of your car is abstract.

A powerful way to manage abstraction is through the use of hierarchical classifications. This allows you to layer the semantics of complex systems, breaking them into more manageable pieces. From the outside, the car is a single object. Once inside, you see that the car consists of several subsystems: steering, brakes, sound system, seat belts, heating, cellular phone, and so on. In turn, each of these subsystems is made up of more specialized units. For instance, the sound system consists of a radio, a CD player, and/or a tape player. The point is that you manage the complexity of the car (or any other complex system)through the use of hierarchical abstractions.

An abstract class is something which is incomplete and you can not create instance of abstract class. If you want to use it you need to make it complete or concrete by extending it. A class is called concrete if it does not contain any abstract method and implements all abstract method inherited from abstract class or interface it has implemented or extended. By the way Java has concept of abstract classes, abstract method but a variable can not be abstract in Java.

Lets take an example of Java Abstract Class called Vehicle. When I am creating a class called Vehicle, I know there should be methods like start() and Stop() but don't know start and stop mechanism of every vehicle since they could have different start and stop mechanism e.g some can be started by kick or some can be by pressing buttons.

Advantage of Abstraction is if there is new type of vehicle introduced we might just need to add one class which extends Vehicle Abstract class and implement specific methods.  Interface of start and stop method would be same.

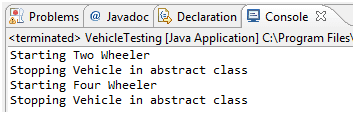
[view plaincopy to clipboardprint?](http://www.w3resource.com/java-tutorial/java-object-oriented-programming.php)

1. **package** oopsconcept;
2. **public** **abstract** **class** VehicleAbstract {
3. **public** **abstract** **void** start();
4. **public** **void** stop(){
5. System.out.println("Stopping Vehicle in abstract class");
6. }
7. }
8. **class** TwoWheeler **extends** VehicleAbstract{
9. @Override
10. **public** **void** start() {
11. System.out.println("Starting Two Wheeler");
12. }
13. }
14. **class** FourWheeler **extends** VehicleAbstract{
15. @Override
16. **public** **void** start() {
17. System.out.println("Starting Four Wheeler");
18. }
19. }

[view plaincopy to clipboardprint?](http://www.w3resource.com/java-tutorial/java-object-oriented-programming.php)

1. **package** oopsconcept;
2. **public** **class** VehicleTesting {
3. **public** **static** **void** main(String[] args) {
4. VehicleAbstract my2Wheeler = **new** TwoWheeler();
5. VehicleAbstract my4Wheeler = **new** FourWheeler();
6. my2Wheeler.start();
7. my2Wheeler.stop();
8. my4Wheeler.start();
9. my4Wheeler.stop();
10. }
11. }

## Output :



## Summary

* An object is an instance of a class.
* Encapsulation provides the security that keeps data and methods safe from inadvertent changes.
* Inheritance is parent-child relationship of class which is mainly used for code reusability.
* Polymorphism definition is that Poly means many and morphos means forms.
* Using abstraction one can simulate real world objects.
* Abstraction provides advantage of code reuse
* Abstraction enables program open for extension

## Project: Circle Class

This small project tests the ability to create a class and then use it to create an object that does something! Circle is a class with **property** radius and **methods** setRadius(), showDiameter(), showArea(). Use double precision and 3.14 for π. The math class constant Math.PI should be used once the [**class libraries**](http://home.cogeco.ca/~ve3ll/jatutor5.htm) tutorial is covered.

Since this is a small project use only one file, say doCircle.java that contains both the main (or driver) class and the Circle class. The driver creates an instance of the Circle class and then displays the diameter and area. Test with radius=3.0 The diameter should read as 6.0 and the area as 28.25999 Once it is working, try to factor it into two files (Circle and doCircle) and compile separately, (the Circle class file first). As an enhancement, try adding a [**command line argument**](http://home.cogeco.ca/~ve3ll/jatutor2.htm#cl) to set the radius.

HELP Me! [Add message as to what is wanted] Get Source Code

## Arrays of Objects

As with arrays of primitive types, arrays of objects allow much more efficient methods of access. Note in this example that once the array of Animals has been structured, it can be used to store objects of any subclass of Animal. By making the method speak() abstract, it can be defined for each subclass and any usage will be polymorphic (ie. adapted to the appropriate object type at runtime). It now becomes very easy to rehearse the speak() method for each object by object indexing.



## Casting Objects

One of the difficulties of using a superclass array to hold many instances of subclass objects is that one can only access properties and methods that are in the superclass (ie. common to all). By **casting** an individual instance to its subclass form, one can refer to any property or method. But first take care to make sure the cast is valid by using the **instanceof** operator. Then perform the cast. As an example using the above Animal class:



Casts to subclass can be done implicitly but explicit casts are recommended. Casts to superclass must be done explicitly. Casts cannot be made between sibling classes.

|  |  |
| --- | --- |
| * [**Tutorials - Table of Contents**](http://home.cogeco.ca/~ve3ll/jatutor0.htm) * [**Tutorial 1 - Structure / Syntax**](http://home.cogeco.ca/~ve3ll/jatutor1.htm) * [**Tutorial 2 - Control Flow**](http://home.cogeco.ca/~ve3ll/jatutor2.htm) * [**Tutorial 3 - Encapsulation & Classes**](http://home.cogeco.ca/~ve3ll/jatutor3.htm) * [**Tutorial 4 - Inheritance & Polymorphism**](http://home.cogeco.ca/~ve3ll/jatutor4.htm) * [**Tutorial 5 - Java Class Libraries**](http://home.cogeco.ca/~ve3ll/jatutor5.htm) * [**Tutorial 6 - String Manipulation**](http://home.cogeco.ca/~ve3ll/jatutor6.htm) * [**Tutorial 7 - Collections, Exceptions, IO**](http://home.cogeco.ca/~ve3ll/jatutor7.htm) * [**Tutorial 8 - File IO**](http://home.cogeco.ca/~ve3ll/jatutor8.htm) * [**Tutorial 9 - GUI Swing Widgets**](http://home.cogeco.ca/~ve3ll/jatutor9.htm) | * [**Tutorial 10 - Intermediate Swing**](http://home.cogeco.ca/~ve3ll/jatutora.htm) * [**Tutorial 11 - Advanced Swing**](http://home.cogeco.ca/~ve3ll/jatutorb.htm) * [**Tutorial 12 - UI Design/Implementation**](http://home.cogeco.ca/~ve3ll/jatutorc.htm) * [**Tutorial 13 - GUI Views and Studies**](http://home.cogeco.ca/~ve3ll/jatutord.htm) * [**Tutorial 14 - Threads & Serialization**](http://home.cogeco.ca/~ve3ll/jatutore.htm) * [**Tutorial 15 - Java Networking**](http://home.cogeco.ca/~ve3ll/jatutorf.htm) * [**Tutorial 16 - Graphics and Imaging**](http://home.cogeco.ca/~ve3ll/jatutorg.htm) * [**Tutorial 17 - Testing Applications**](http://home.cogeco.ca/~ve3ll/jatutorh.htm) * [**Tutorial 18 - Apps Deployment**](http://home.cogeco.ca/~ve3ll/jatutori.htm) * [**Appendices**](http://home.cogeco.ca/~ve3ll/jatutorx.htm) |

Topic 1: **Class, Objects, Methods & Properties**    :

### 1.  Why OOP?

Suppose that you want to assemble your own PC, you go to a hardware store and pick up a motherboard, a processor, some RAMs, a hard disk, a casing, a power supply, and put them together.  You turn on the power, and the PC runs.  You need not worry whether the CPU is 1-core or 6-core; the motherboard is a 4-layer or 6-layer; the hard disk has 4 plates or 6 plates, 3 inches or 5 inches in diameter; the RAM is made in Japan or Korea, and so on. You simply put the hardware components together and expect the machine to run.  Of course, you have to make sure that you have the correctinterfaces, i.e., you pick an IDE hard disk rather than a SCSI hard disk, if your motherboard supports only IDE; you have to select RAMs with the correct speed rating, and so on.  Nevertheless, it is not difficult to set up a machine from hardware components.

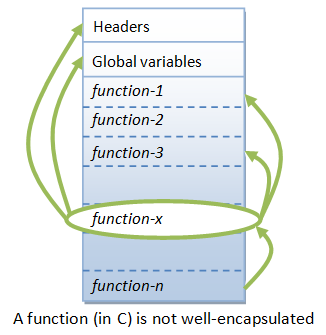
Similarly, a car is assembled from parts and components, such as chassis, doors, engine, wheels, brake and transmission. The components are reusable, e.g., a wheel can be used in many cars (of the same specifications).

Hardware, such as computers and cars, are assembled from parts, which are reusable components.

How about software?  Can you "assemble" a software application by picking a routine here, a routine there, and expect the program to run?  The answer is obviously no!  Unlike hardware, it is very difficult to "assemble" an application from software components.  Since the advent of computer 70 years ago, we have written tons and tons of programs.  However, for each new application, we have to re-invent the wheels and write the program from scratch.

Why re-invent the wheels? Why rewriting codes?

##### Traditional Procedural-Oriented languages



Traditional procedural-oriented programming languages (such as C, Fortran, Cobol and Pascal) suffer some notable drawbacks in creating reusable software components:

1. The procedural-oriented programs are made up of functions. Functions are less reusable. It is very difficult to copy a function from one program and reuse in another program because the function is likely to reference the global variables and other functions. In other words, functions are not well-encapsulated as a self-contained reusable unit.
2. The procedural languages are not suitable of high-level abstraction for solving real life problems. For example, C programs uses constructs such as if-else, for-loop, array, method, pointer, which are low-level and hard to abstract real problems such as a Customer Relationship Management (CRM) system or a computer soccer game.

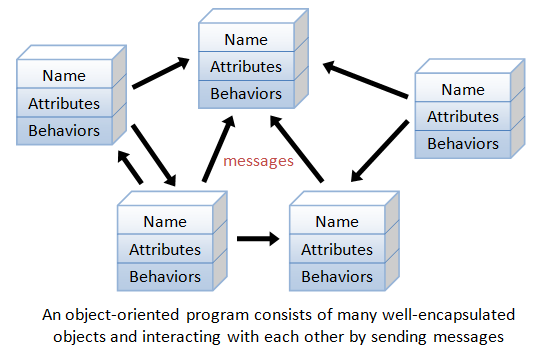
In brief, the traditional procedural-languages separate the data structures and algorithms of the software entities.

In the early 1970s, the US Department of Defense (DoD) commissioned a task force to investigate why its IT budget always went out of control; but without much to show for. The findings are:

1. 80% of the budget went to the software (with the remaining 20% to the hardware).
2. More than 80% of the software budget went to maintenance (only the remaining 20% for new software development).
3. Hardware components could be applied to various products, and their integrity normally did not affect other products. (Hardware can share and reuse! Hardware faults are isolated!)
4. Software procedures were often non-sharable and not reusable. Software faults could affect other programs running in computers.

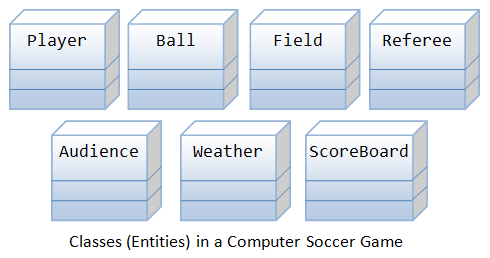
The task force proposed to make software behave like hardware OBJECT. Subsequently, DoD replaces over 450 computer languages, which were then used to build DoD systems, with an object-oriented language called Ada.

##### Object-Oriented Programming Languages



Object-oriented programming (OOP) languages are designed to overcome these problems.

1. The basic unit of OOP is a class, which encapsulates both the static properties and dynamic operations within a "box", and specifies the public interface for using these boxes. Since classes are well-encapsulated, it is easier to reuse these classes. In other words, OOP combines the data structures and algorithms of a software entity inside the same box.
2. OOP languages permit higher level of abstraction for solving real-life problems. The traditional procedural language (such as C and Pascal) forces you to think in terms of the structure of the computer (e.g. memory bits and bytes, array, decision, loop) rather than thinking in terms of the problem you are trying to solve. The OOP languages (such as Java, C++ and C#) let you think in the problem space, and use software objects to represent and abstract entities of the problem space to solve the problem.



As an example, suppose you wish to write a computer soccer games (which I consider as a complex application). It is quite difficult to model the game in procedural-oriented languages. But using OOP languages, you can easily model the program accordingly to the "real things" appear in the soccer games.

* Player: attributes include name, number, location in the field, and etc; operations include run, jump, kick-the-ball, and etc.
* Ball:
* Reference:
* Field:
* Audience:
* Weather:

Most importantly, some of these classes (such as Ball and Audience) can be reused in another application, e.g., computer basketball game, with little or no modification.

##### Benefits of OOP

The procedural-oriented languages focus on procedures, with function as the basic unit. You need to first figure out all the functions and then think about how to represent data.

The object-oriented languages focus on components that the user perceives, with objects as the basic unit. You figure out all the objects by putting all the data and operations that describe the user's interaction with the data.

Object-Oriented technology has many benefits:

* Ease in software design as you could think in the problem space rather than the machine's bits and bytes. You are dealing with high-level concepts and abstractions. Ease in design leads to more productive software development.
* Ease in software maintenance: object-oriented software are easier to understand, therefore easier to test, debug, and maintain.
* Reusable software: you don't need to keep re-inventing the wheels and re-write the same functions for different situations. The fastest and safest way of developing a new application is to reuse existing codes - fully tested and proven codes.

### 2.  OOP in Java

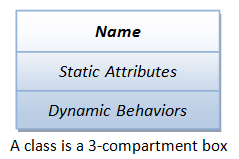
#### 2.1  Class & Instances

In Java, a class is a definition of objects of the same kind. In other words, a class is a blueprint, template, or prototype that defines and describes the static attributes and dynamic behaviorscommon to all objects of the same kind.

An instance is a realization of a particular item of a class. In other words, an instance is an instantiation of a class. All the instances of a class have similar properties, as described in the class definition. For example, you can define a class called "Student" and create three instances of the class "Student" for "Peter", "Paul" and "Pauline".

The term "object" usually refers to instance. But it is often used loosely, which may refer to a class or an instance.

#### 2.2  A Class is a 3-Compartment Box encapsulating Data and Operations

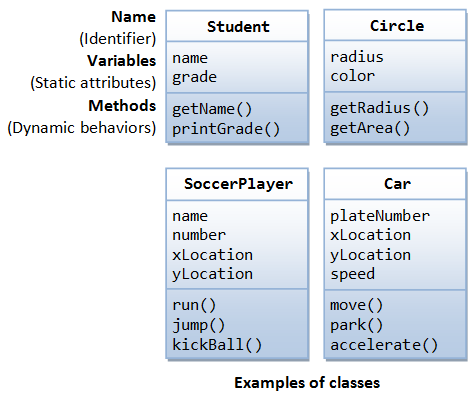


A class can be visualized as a three-compartment box, as illustrated:

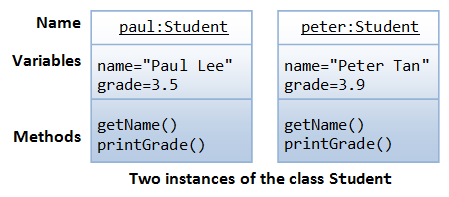
1. Name (or identity): identifies the class.
2. Variables (or attribute, state, field): contains the static attributes of the class.
3. Methods (or behaviors, function, operation): contains the dynamic behaviors of the class.

In other words, a class encapsulates the static attributes (data) and dynamic behaviors (operations that operate on the data) in a box.

The followings figure shows a few examples of classes:



The following figure shows two instances of the class Student, identified as "paul" and "peter".



**Unified Modeling Language (UML) Class and Instance Diagrams:** The above class diagrams are drawn according to the UML notations. A class is represented as a 3-compartment box, containing name, variables, and methods, respectively. Class name is shown in bold and centralized. An instance (object) is also represented as a 3-compartment box, with instance name shown as instanceName:Classname and underlined.

##### Brief Summary

1. A class is a programmer-defined, abstract, self-contained, reusable software entity that mimics a real-world thing.
2. A class is a 3-compartment box containing the name, variables and the methods.
3. A class encapsulates the data structures (in variables) and algorithms (methods). The values of the variables constitute its state. The methods constitute its behaviors.
4. An instance is an instantiation (or realization) of a particular item of a class.

#### 2.3  Class Definition

In Java, we use the keyword class to define a class. For examples:

public **class** **Circle** { // class name

double radius; // variables

String color;

double getRadius() {...} // methods

double getArea() {...}

}

public **class** **SoccerPlayer** { // class name

int number; // variables

String name;

int x, y;

void run() {...} // methods

void kickBall() {...}

}

The syntax for class definition in Java is:

[AccessControlModifier] **class** ClassName **{**

// class body contains definition of variables and methods

...

**}**

We shall explain the access control modifier, such as public and private, later.

**Class Naming Convention:** A class name shall be a noun or a noun phrase made up of several words. All the words shall be initial-capitalized (camel-case). Use a singular noun for class name. Choose a meaningful and self-descriptive classname. For examples, SoccerPlayer, HttpProxyServer, FileInputStream, PrintStream and SocketFactory.

#### 2.4  Creating Instances of a Class

To create an instance of a class, you have to:

1. Declare an instance identifier (instance name) of a particular class.
2. Construct the instance (i.e., allocate storage for the instance and initialize the instance) using the "new" operator.

For examples, suppose that we have a class called Circle, we can create instances of Circle as follows:

// Declare 3 instances of the class Circle, c1, c2, and c3

Circle c1, c2, c3;

// Allocate and construct the instances via new operator

c1 = **new** Circle();

c2 = **new** Circle(2.0);

c3 = **new** Circle(3.0, "red");

// You can declare and construct in the same statement

Circle c4 = **new** Circle();

#### 2.5  Dot Operator

The variables and methods belonging to a class are formally called member variables and member methods. To reference a member variable or method, you must:

1. first identify the instance you are interested in, and then
2. Use the dot operator (.) to reference the member (variable or method).

For example, suppose that we have a class called Circle, with two variables (radius and color) and two methods (getRadius() and getArea()). We have created three instances of the classCircle, namely, c1, c2 and c3. To invoke the method getArea(), you must first identity the instance of interest, says c2, then use the dot operator, in the form of c2.getArea(), to invoke thegetArea() method of instance c2.

For example,

// Declare and construct instances c1 and c2 of the class Circle

Circle c1 = new Circle ();

Circle c2 = new Circle ();

// Invoke member methods for the instance c1 via dot operator

System.out.println(c1.getArea());

System.out.println(c1.getRadius());

// Reference member variables for instance c2 via dot operator

c2.radius = 5.0;

c2.color = "blue";

Calling getArea() without identifying the instance is meaningless, as the radius is unknown (there could be many instances of Circle - each maintaining its own radius).

In general, suppose there is a class called *AClass* with a member variable called *aVariable* and a member method called aMethod(). An instance called *anInstance* is constructed for *AClass*. You use *anInstance.aVariable* and anInstance.aMethod().

#### 2.6  Member Variables

A member variable has a name (or identifier) and a type; and holds a value of that particular type (as descried in the earlier chapter). A member variable can also be an instance of a certain class (to be discussed later).

**Variable Naming Convention:** A variable name shall be a noun or a noun phrase made up of several words. The first word is in lowercase and the rest of the words are initial-capitalized (camel-case), e.g., fontSize, roomNumber, xMax, yMin and xTopLeft. Take note that variable name begins with an lowercase, while class name begins with an uppercase.

The formal syntax for variable definition in Java is:

[AccessControlModifier] type variableName [**=** initialValue]**;**

[AccessControlModifier] type variableName-1 [**=** initialValue-1] [**,** type variableName-2 [**=** initialValue-2]] ... **;**

For example,

private double radius;

public int length = 1, width = 1;

#### 2.7  Member Methods

A method (as described in the earlier chapter):

1. receives parameters from the caller,
2. performs the operations defined in the method body, and
3. returns a piece of result (or void) to the caller.

The syntax for method declaration in Java is as follows:

[AccessControlModifier] returnType methodName **(**[argumentList]**) {**

// method body or implementation

......

**}**

For examples:

public double getArea() {

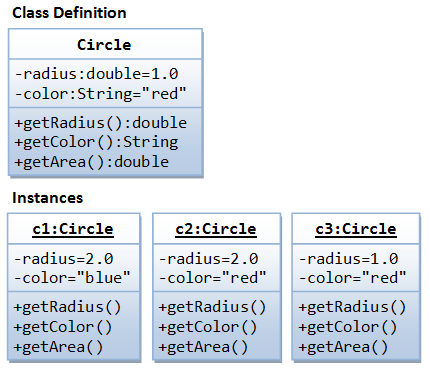
return radius\*radius\*Math.PI;

}

**Method Naming Convention:** A method name shall be a verb, or a verb phrase made up of several words. The first word is in lowercase and the rest of the words are initial-capitalized (camel-case). For example, getRadius(), getParameterValues().

Take note that variable name is a noun (denoting a static attribute), while method name is a verb (denoting an action). They have the same naming convention. Nevertheless, you can easily distinguish them from the context. Methods take arguments in parentheses (possibly zero argument with empty parentheses), but variables do not. In this writing, methods are denoted with a pair of parentheses, e.g., println(), getArea() for clarity.

#### 2.8  Putting them together: An OOP Example



A class called Circle is to be defined as illustrated in the class diagram. It contains two variables: radius (of typedouble) and color (of type String); and three methods: getRadius(), getColor(), and getArea().

Three instances of Circles called c1, c2, and c3 shall then be constructed with their respective data members, as shown in the instance diagrams.

The source codes for Circle.java is as follows:

##### Circle.java

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31 | // Define the Circle class  public class **Circle** { // Save as "Circle.java"  // Private variables  private double radius;  private String color;    // Constructors (overloaded)  public Circle() { // 1st Constructor  radius = 1.0;  color = "red";  }  public Circle(double r) { // 2nd Constructor  radius = r;  color = "red";  }  public Circle(double r, String c) { // 3rd Constructor  radius = r;  color = c;  }    // Public methods  public double getRadius() {  return radius;  }  public String getColor() {  return color;  }  public double getArea() {  return radius\*radius\*Math.PI;  }  } |

Compile "Circle.java" into "Circle.class".

Notice that the Circle class does not have a main() method. Hence, it is NOT a standalone program and you cannot run the Circle class by itself. The Circle class is meant to be a building block and used in other programs.

##### TestCircle.java

We shall now write another class called TestCircle, which uses the Circle class. The TestCircle class has a main() method and can be executed.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22 | // Test driver program for the Circle class  public class **TestCircle** { // Save as "TestCircle.java"  public static void main(String[] args) { // Execution entry point  // Construct an instance of the Circle class called c1  Circle c1 = new Circle(2.0, "blue"); // Use 3rd constructor  System.out.println("Radius is " + c1.getRadius() // use dot operator to invoke member methods  + " Color is " + c1.getColor()  + " Area is " + c1.getArea());    // Construct another instance of the Circle class called c2  Circle c2 = new Circle(2.0); // Use 2nd constructor  System.out.println("Radius is " + c2.getRadius()  + " Color is " + c2.getColor()  + " Area is " + c2.getArea());    // Construct yet another instance of the Circle class called c3  Circle c3 = new Circle(); // Use 1st constructor  System.out.println("Radius is " + c3.getRadius()  + " Color is " + c3.getColor()  + " Area is " + c3.getArea());  }  } |

Compile TestCircle.java into TestCircle.class.

Run the TestCircle and study the output:

Radius is 2.0 Color is blue Area is 12.566370614359172

Radius is 2.0 Color is red Area is 12.566370614359172

Radius is 1.0 Color is red Area is 3.141592653589793

#### 2.9  Constructors

A constructor is a special method that has the same method name as the class name. In the above Circle class, we define three overloaded versions of constructor "Circle(....)". A constructor is used to construct and initialize all the member variables. To create a new instance of a class, you need to use a special "new" operator followed by a call to one of the constructors. For example,

Circle c1 = new Circle();

Circle c2 = new Circle(2.0);

Circle c3 = new Circle(3.0, "red");

A constructor is different from an ordinary method in the following aspects:

* The name of the constructor method is the same as the class name, and by classname convention, begins with an uppercase.
* Constructor has no return type (or implicitly returns void). Hence, no return statement is allowed inside the constructor's body.
* Constructor can only be invoked via the "new" operator. It can only be used once to initialize the instance constructed. You cannot call the constructor afterwards.
* Constructors are not inherited (to be explained later).

A constructor with no parameter is called the default constructor, which initializes the member variables to their default value. For example, the Circle() in the above example.

#### 2.10  Method Overloading

Method overloading means that the same method name can have different implementations (versions). However, the different implementations must be distinguishable by their argument list (either the number of arguments, or the type of arguments, or their order).

**Example:** The method average() has 3 versions, with different argument lists. The caller can invoke the chosen version by supplying the matching arguments.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22 | // Example to illustrate Method Overloading  public class **TestMethodOverloading** {  public static int average(int n1, int n2) { // A  return (n1+n2)/2;  }    public static double average(double n1, double n2) { // B  return (n1+n2)/2;  }    public static int average(int n1, int n2, int n3) { // C  return (n1+n2+n3)/3;  }  public static void main(String[] args) {  System.out.println(average(1, 2)); // Use A  System.out.println(average(1.0, 2.0)); // Use B  System.out.println(average(1, 2, 3)); // Use C  System.out.println(average(1.0, 2)); // Use B - int 2 implicitly casted to double 2.0  // average(1, 2, 3, 4); // Compilation Error - No matching method  }  } |

##### Overloading Circle Class' Constructor

The above Circle class has three versions of constructors differentiated by their parameter list, as followed:

Circle()

Circle(double r)

Circle(double r, String c)

Depending on the actual argument list used when invoking the method, the matching constructor will be invoked. If your argument list does not match any one of the methods, you will get a compilation error.

#### 2.11  public vs. private - Access Control Modifiers

An access control modifier can be used to control the visibility of a class, or a member variable or a member method within a class. We begin with the following two access control modifiers:

1. public: The class/variable/method is accessible and available to all the other objects in the system.
2. private: The class/variable/method is accessible and available within this class only.

For example, in the above Circle definition, the member variable radius is declared private. As the result, radius is accessible inside the Circle class, but NOT inside the TestCircle class. In other words, you cannot use "c1.radius" to refer to c1's radius in TestCircle. Try inserting the statement "System.out.println(c1.radius);" in TestCircle and observe the error message. Try changing radius to public, and re-run the statement.

On the other hand, the method getRadius() is declared public in the Circle class. Hence, it can be invoked in the TestCircle class.

**UML Notation:** In UML notation, public members are denoted with a "+", while private members with a "-" in the class diagram.

More access control modifiers will be discussed later.

#### 2.12  Information Hiding and Encapsulation

A class encapsulates the name, static attributes and dynamic behaviors into a "3-compartment box". Once a class is defined, you can seal up the "box" and put the "box" on the shelve for others to use and reuse. Anyone can pick up the "box" and use it in their application. This cannot be done in the traditional procedural-oriented language like C, as the static attributes (or variables) are scattered over the entire program and header files. You cannot "cut" out a portion of C program, plug into another program and expect the program to run without extensive changes.

Member variables of a class are typically hidden from the outside word (i.e., the other classes), with private access control modifier. Access to the member variables are provided via publicassessor methods, e.g., getRadius() and getColor().

This follows the principle of information hiding. That is, objects communicate with each others using well-defined interfaces (public methods). Objects are not allowed to know the implementation details of others. The implementation details are hidden or encapsulated within the class. Information hiding facilitates reuse of the class.

**Rule of Thumb:** Do not make any variable public, unless you have a good reason.

#### 2.13  Getters and Setters

To allow other classes to read the value of a private variable says xxx, you shall provide a get method (or getter or accessor method) called getXxx(). A get method need not expose the data in raw format. It can process the data and limit the view of the data others will see. get methods cannot modify the variable.

To allow other classes to modify the value of a private variable says xxx, you shall provide a set method (or setter or mutator method) called setXxx(). A set method could provide data validation (such as range checking), and transform the raw data into the internal representation.

For example, in our Circle class, the variables radius and color are declared private. That is to say, they are only available within the Circle class and not visible in any other classes - including TestCircle class. You cannot access the private variables radius and color from the TestCircle class directly - via says c1.radius or c1.color. The Circle class provides two public accessor methods, namely, getRadius() and getColor(). These methods are declared public. The class TestCircle can invoke these public accessor methods to retrieve the radiusand color of a Circle object, via says c1.getRadius() and c1.getColor().

There is no way you can change the radius or color of a Circle object, after it is constructed in the TestCircle class. You cannot issue statements such as c1.radius = 5.0 to change theradius of instance c1, as radius is declared as private in the Circle class and is not visible to other classes including TestCircle.

If the designer of the Circle class permits the change the radius and color after a Circle object is constructed, he has to provide the appropriate set methods (or setters or mutator methods), e.g.,

// Setter for color

public void setColor(String c) {

color = c;

}

// Setter for radius

public void setRadius(double r) {

radius = r;

}

With proper implementation of information hiding, the designer of a class has full control of what the user of the class can and cannot do.

#### 2.14  Keyword "this"

You can use keyword "this" to refer to thisinstance inside a class definition.

One of the main usage of keyword this is to resolve ambiguity.

public class Circle {

double radius; // Member variable called "radius"

public Circle(double radius) { // Method's argument also called "radius"

this.radius = radius;

// "this.radius" refers to this instance's member variable

// "radius" resolved to the method's argument.

}

...

}

In the above codes, there are two identifiers called radius - a member variable of the class and the method's argument. This causes naming conflict. To avoid the naming conflict, you could name the method's argument r instead of radius. However, radius is more approximate and meaningful in this context. Java provides a keyword called this to resolve this naming conflict. "this.radius" refers to the member variable; while "radius" resolves to the method's argument.

Using the keyword "this", the constructor, getter and setter methods for a private variable called xxx of type T are as follows:

public class Aaa {

// A private variable named xxx of type T

private T xxx;

// Constructor

public Aaa(T xxx) {

this.xxx = xxx;

}

// A getter for variable xxx of type T receives no argument and return a value of type T

public T getXxx() {

return xxx;

}

// A setter for variable xxx of type T receives a parameter of type T and return void

public void setXxx(T xxx) {

this.xxx = xxx;

}

}

For a boolean variable xxx, the getter shall be named isXxx(), instead of getXxx(), as follows:

// Private boolean variable

private boolean xxx;

// Getter

public boolean isXxx() {

return xxx;

}

// Setter

public void setXxx(boolean xxx) {

this.xxx = xxx;

}

Notes:

* this.varName refers to *varName* of this instance; this.methodName(...) invokes methodName(...) of this instance.
* In a constructor, we can use this(...) to call another constructor of this class.
* Inside a method, we can use the statement "return this" to return this instance to the caller.

#### 2.15  Method toString()

Every well-designed Java class should have a public method called toString() that returns a string description of the object. You can invoke the toString() method explicitly by callinganInstanceName.toString() or implicitly via println() or String concatenation operator '+'. Running println(anInstance) with an object argument invokes the toString() method of that instance implicitly.

For example, if the following toString() method is included in our Circle class:

// Return a short String description of this instance

public String toString() {

return "Circle with radius = " + radius + " and color of " + color;

}

In your TestCircle class, you can get a short text descriptor of a Circle object by:

Circle c1 = new Circle();

System.out.println(c1.toString()); // explicitly calling toString()

System.out.println(c1); // implicit call to c1.toString()

System.out.println("c1 is: " + c1); // '+' invokes c1.toString() to get a String before concatenation

The signature of toString() is:

public String toString() { ...... }

#### 2.16  Constants (final)

Constants are variables defined with the modifier final. A final variable can only be assigned once and its value cannot be modified once assigned. For example,

public final double X\_REFERENCE = 1.234;

private final int MAX\_ID = 9999;

MAX\_ID = 10000; // error: cannot assign a value to final variable MAX\_ID

// You need to initialize a final member variable during declaration

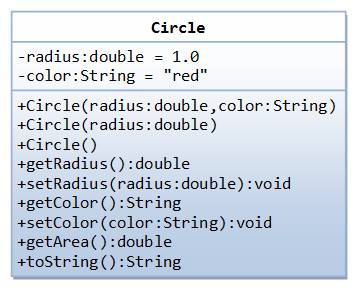
private final int SIZE; // error: variable SIZE might not have been initialized

**Constant Naming Convention:** A constant name is a noun, or a noun phrase made up of several words. All words are in uppercase separated by underscores '\_', for examples,X\_REFERENCE, MAX\_INTEGER and MIN\_VALUE.

Advanced Notes:

1. A final primitive variable cannot be re-assigned a new value.
2. A final instance cannot be re-assigned a new address.
3. A final class cannot be sub-classed (or extended).
4. A final method cannot be overridden.

#### 2.17  Putting Them Together in the Revised Circle Class



##### Circle.java

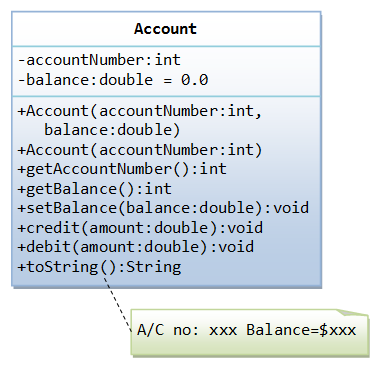
|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42  43  44  45  46  47  48 | // The Circle class definition  public class Circle { // Save as "Circle.java"  // Public constants  public static final double DEFAULT\_RADIUS = 8.8;  public static final String DEFAULT\_COLOR = "red";    // Private variables  private double radius;  private String color;    // Constructors (overloaded)  public Circle() { // 1st Constructor  radius = DEFAULT\_RADIUS;  color = DEFAULT\_COLOR;  }  public Circle(double radius) { // 2nd Constructor  this.radius = radius;  color = DEFAULT\_COLOR;  }  public Circle(double radius, String color) { // 3rd Constructor  this.radius = radius;  this.color = color;  }    // Public getter and setter for private variables  public double getRadius() {  return radius;  }  public void setRadius(double radius) {  this.radius = radius;  }  public String getColor() {  return color;  }  public void setColor(String color) {  this.color = color;  }    // toString() to provide a short description of this instance  public String toString() {  return "Circle with radius = " + radius + " and color of " + color;  }    // Public methods  public double getArea() {  return radius\*radius\*Math.PI;  }  } |

##### TRY

Write a test driver to test ALL the public methods in the Circle class.

### 3.  More Examples on Classes

#### 3.1  Example: The Account Class



A class called Account, which models a bank account, is designed as shown in the class diagram. It contains:

* Two private variables: accountNumber (int) and balance (double), which maintains the current account balance.
* Public methods credit() and debit(), which adds or subtracts the given amount from the balance, respectively. Thedebit() method shall print "amount withdrawn exceeds the current balance!" if amount is more than balance.
* A toString(), which returns "A/C no: xxx Balance=xxx" (e.g., A/C no: 991234 Balance=$88.88), with balance rounded to two decimal places.

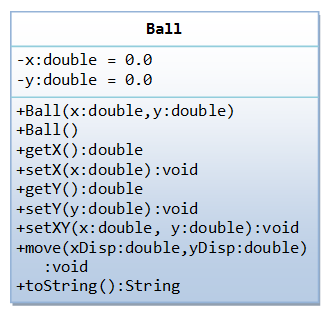
##### Account.java

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42  43  44  45  46 | /\* The Account class Definition \*/  public class Account {  // Private variables  private int accountNumber;  private double balance;    // Constructors  public Account(int accountNumber, double balance) {  this.accountNumber = accountNumber;  this.balance = balance;  }    public Account(int accountNumber) {  this.accountNumber = accountNumber;  balance = 0;  }    // Public getters and setters for private variables  public int getAccountNumber() {  return accountNumber;  }    public double getBalance() {  return balance;  }    public void setBalance(double balance) {  this.balance = balance;  }    public void credit(double amount) {  balance += amount;  }    public void debit(double amount) {  if (balance < amount) {  System.out.println("amount withdrawn exceeds the current balance!");  } else {  balance -= amount;  }  }    public String toString() {  return String.format("A/C no: %d Balance=%.2f", accountNumber, balance);  }  } |

##### TestAccount.java

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15 | public class TestAccount {  public static void main(String[] args) {  Account a1 = new Account(1234, 99.99);  System.out.println(a1);  a1.credit(10.001);  a1.debit(5);  System.out.println(a1);  System.out.println(a1.getBalance());  a1.setBalance(0);  System.out.println(a1);    Account a2 = new Account(8888);  System.out.println(a2);  }  } |

#### 3.2  Example: The Ball class



A Ball class models a moving ball, designed as shown in the class diagram, contains the following members:

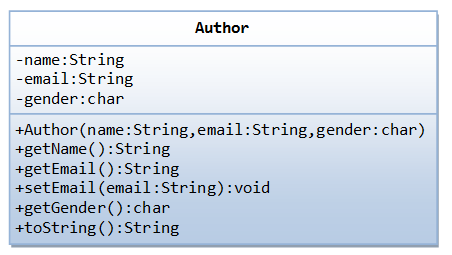
* Two private variables x, y, which maintain the position of the ball.
* Constructors, public getters and setters for the private variables.
* A method setXY(), which sets the position of the ball and setXYSpeed() to set the speed of the ball.
* A method move(), which increases x and y by the given xDisp and yDisp, respectively.
* A toString(), which returns "Ball @ (x,y)".

##### Ball.java

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42  43 | // Define the class Ball  public class Ball {  // Private variables  private double x, y; // x and y location    // Constructors  public Ball(double x, double y) {  this.x = x;  this.y = y;  }  public Ball() {  x = 0.0;  y = 0.0;  }    // Public getters and setters for private variables x and y  public double getX() {  return x;  }  public void setX(double x) {  this.x = x;  }  public double getY() {  return y;  }  public void setY(double y) {  this.y = y;  }    public void setXY(double x, double y) {  this.x = x;  this.y = y;  }    public void move(double xDisp, double yDisp) {  x += xDisp;  y += yDisp;  }    public String toString() {  return "Ball @ (" + x + "," + y + ")";  }  } |

#### 3.3  The Author and Book Classes

##### Let's start with the Author class



A class called Author is defined as shown in the class diagram. It contains:

* Three private member variables: name (String), email (String), and gender (char of either 'm', 'f', or 'u'for unknown - you might also use a boolean variable called male having value of true or false).
* One constructor to initialize the name, email and gender with the given values.  
  (There is no default constructor for Author, as there are no defaults for name, email and gender.)
* Public getters/setters: getName(), getEmail(), setEmail(), and getGender().  
  (There are no setters for name and gender, as these attributes cannot be changed.)
* A toString() method that returns "author-name (gender) at email", e.g., "Tan Ah Teck (m) at ahTeck@somewhere.com".

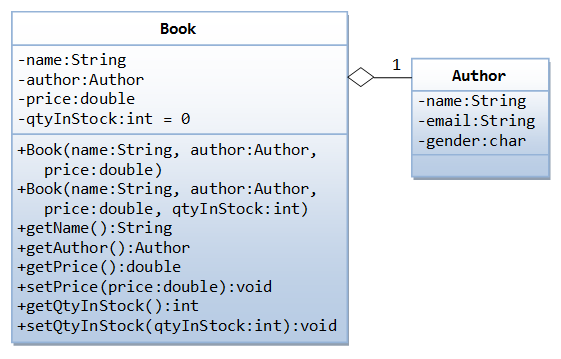
##### Author.java

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36 | // The Author class definition  public class Author {  // Private variables  private String name;  private String email;  private char gender;    // Constructor  public Author(String name, String email, char gender) {  this.name = name;  this.email = email;  this.gender = gender;  }    // Public getters and setters for private variables  public String getName() {  return name;  }    public String getEmail() {  return email;  }    public char getGender() {  return gender;  }    public void setEmail(String email) {  this.email = email;  }    // toString() to describe itself  public String toString() {  return name + " (" + gender + ") at " + email;  }  } |

##### A Test Driver Program: TestAuthor.java

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9 | // A test driver for the Author class  public class TestAuthor {  public static void main(String[] args) {  Author teck = new Author("Tan Ah Teck", "teck@somewhere.com", 'm');  System.out.println(teck); // toString()  teck.setEmail("teck@nowhere.com");  System.out.println(teck); // toString()  }  } |

##### A Book is written by an Author - Using an "Object" Member Variable



Let's design a Book class. Assume that a book is written by one and only one author. The Book class (as shown in the class diagram) contains the following members:

* Four private member variables: name (String), author (an instance of the class Author you have just created, assume that each book has one and only one author), price (double), andqtyInStock (int).
* Two overloaded constructors.
* Getters/Setters: getName(), getAuthor(), getPrice(), setPrice(), getQtyInStock(),setQtyInStock().
* A toString() that returns "'book-name' by author-name (gender) at email".  
  You should reuse the Author's toString() method, which returns "author-name (gender) at email".

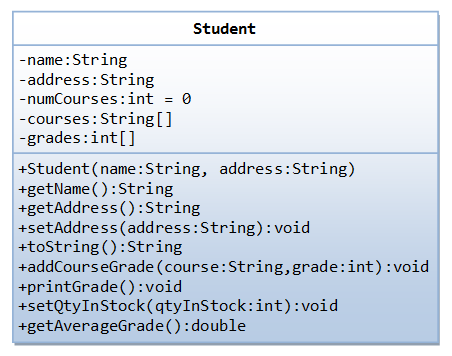
##### Book.java

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42  43  44  45  46  47  48  49  50  51  52  53 | // The Book class definition  public class Book {  // Private variables  private String name;  private Author author;  private double price;  private int qtyInStock;    // Constructors  public Book(String name, Author author, double price) {  this.name = name;  this.author = author;  this.price = price;  this.qtyInStock = 0; // Not given, set to the default value  }    public Book(String name, Author author, double price, int qtyInStock) {  this.name = name;  this.author = author;  this.price = price;  this.qtyInStock = qtyInStock;  }    // Getters and Setters  public String getName() {  return name;  }    public Author getAuthor() {  return author; // return member author, which is an instance of class Author  }    public double getPrice() {  return price;  }    public void setPrice(double price) {  this.price = price;  }    public int getQtyInStock() {  return qtyInStock;  }    public void setQtyInStock(int qtyInStock) {  this.qtyInStock = qtyInStock;  }    // toString() t describe itself  public String toString() {  return "'" + name + "' by " + author; // author.toString()  }  } |

##### A Test Driver Program - TestBook.java

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15 | // A test driver program for the Book class  public class TestBook {  public static void main(String[] args) {  Author teck = new Author("Tan Ah Teck", "teck@somewhere.com", 'm');  System.out.println(teck); // toString()    Book dummyBook = new Book("Java for dummies", teck, 9.99, 88);  System.out.println(dummyBook); // toString()    Book moreDummyBook = new Book("Java for more dummies",  new Author("Peter Lee", "peter@nowhere.com", 'm'), // anonymous instance of Author  19.99, 8);  System.out.println(moreDummyBook); // toString()  }  } |

#### 3.4  The Student Class



Suppose that our application requires us to model a group of students. A student has a name and address. We are required to keep track of the courses taken by each student, together with the grades (between 0 and 100) for each of the courses. A student shall not take more than 30 courses for the entire program. We are required to print all course grades, and also the overall average grade.

We can design the Student class as shown in the class diagram. The class contains:

* Private member variables name (String), address (String), numCourses, course and grades. The numCoursesmaintains the number of courses taken by the student so far. The courses and grades are two parallel arrays, storing the courses taken (e.g., {"IM101", "IM102", "IM103"}) and their respective grades (e.g. {89, 56, 98}).
* A constructor that constructs an instance with the given name and Address. It also constructs the courses andgrades arrays and set the numCourses to 0.
* Getters for name and address; setter for address. No setter is defined for name as it shall not be changed.
* A toString(), which prints "name(address)".
* A method addCourseGrade(course, grade), which appends the given course and grade into the coursesand grades arrays, respectively, and increments numCourses.
* A method printGrades(), which prints "name(address) course1:grade1, course2:grade2,...". You should reuse the toString() here.
* A method getAverageGrade(), which returns the average grade of all the courses taken.

**UML Notations:** In UML notations, a variable is written as [+|-]varName:type; a method is written as [+|-]methodName(arg1:type1, arg2:type2,...):returnType, where '+'denotes public and '-' denotes private.

##### Student.java

The source code for Student.java is as follows:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42  43  44  45  46  47  48  49  50  51  52  53  54  55  56  57  58  59  60  61  62  63  64  65 | // The student class definition  public class Student {  // Private member variables  private String name;  private String address;  // Courses taken and grades for the courses are kept in 2 arrays of the same length  private String[] courses;  private int[] grades; // A grade is between 0 to 100  private int numCourses; // Number of courses taken so far  private static final int MAX\_COURSES = 30; // maximum courses    // Constructor  public Student(String name, String address) {  this.name = name;  this.address = address;  courses = new String[MAX\_COURSES]; // allocate arrays  grades = new int[MAX\_COURSES];  numCourses = 0; // no courses so far  }    // Public getter for private variable name  public String getName() {  return name;  }    // Public getter for private variable address  public String getAddress() {  return address;  }    // Public setter for private variable address  public void setAddress(String address) {  this.address = address;  }    // Describe itself  public String toString() {  return name + "(" + address + ")";  }    // Add a course and grade  public void addCourseGrade(String course, int grade) {  courses[numCourses] = course;  grades[numCourses] = grade;  ++numCourses;  }    // Print all courses taken and their grades  public void printGrades() {  System.out.print(this); // toString()  for (int i = 0; i < numCourses; ++i) {  System.out.print(" " + courses[i] + ":" + grades[i]);  }  System.out.println();  }    // Compute the average grade  public double getAverageGrade() {  int sum = 0;  for (int i = 0; i < numCourses; ++i) {  sum += grades[i];  }  return (double)sum/numCourses;  }  } |

##### TestStudent.java

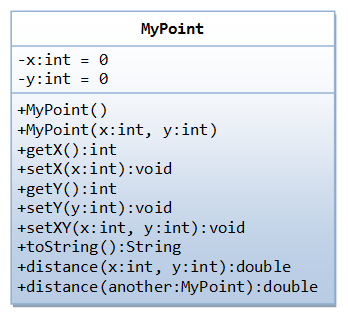
Let us write a test program to create a student named "Tan Ah Teck", who has taken 3 courses, "IM101", "IM102" and "IM103" with grades of 89, 57, and 96 respectively. We shall print all the course grades, and the average grade.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11 | // A test driver program for the Student class  public class TestStudent {  public static void main(String[] args) {  Student ahTeck = new Student("Tan Ah Teck", "1 Happy Ave");  ahTeck.addCourseGrade("IM101", 89);  ahTeck.addCourseGrade("IM102", 57);  ahTeck.addCourseGrade("IM103", 96);  ahTeck.printGrades();  System.out.printf("The average grade is %.2f", ahTeck.getAverageGrade());  }  } |

Tan Ah Teck(1 Happy Ave) IM101:89 IM102:57 IM103:96

The average grade is 80.67

#### 3.5  The MyPoint and MyCircle class



A class called MyPoint, which models a 2D point with x and y coordinates, is designed as shown in the class diagram. It contains:

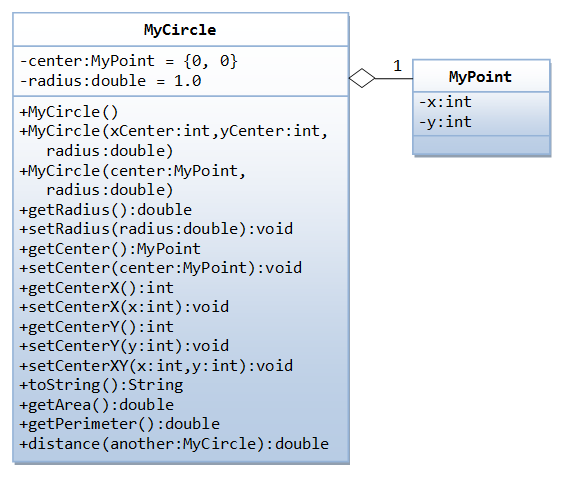
* Two private member variables x (int) and y (int).
* A default (no-arg) constructor that construct a point at (0,0).
* A constructor that constructs a point with the given x and y coordinates.
* Getter and setter for the private variables x and y.
* A method setXY() to set both x and y.
* A toString() method that returns a string description of the instance in the format "(x,y)".
* A method distance(int x, int y), which returns the distance from this point to another point at the given (x,y) coordinates in double.
* An overloaded distance(MyPoint another) method, which returns the distance from this point to the given MyPoint instance called another.

##### MyPoint.java

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42  43  44  45  46  47  48  49  50  51  52  53  54  55 | // The MyPoint class definition  public class MyPoint {  // Private member variables  private int x;  private int y;    // Constructors  public MyPoint() {  x = 0;  y = 0;  }    public MyPoint(int x, int y) {  this.x = x;  this.y = y;  }    // Getters and Setters  public int getX() {  return x;  }    public int getY() {  return y;  }    public void setX(int x) {  this.x = x;  }    public void setY(int y) {  this.y = y;  }    public void setXY(int x, int y) {  this.x = x;  this.y = y;  }    public String toString() {  return "(" + x + "," + y + ")";  }    public double distance(int x, int y) {  int xDiff = this.x - x;  int yDiff = this.y - y;  return Math.sqrt(xDiff\*xDiff + yDiff\*yDiff);  }    public double distance(MyPoint another) {  int xDiff = this.x - another.x;  int yDiff = this.y - another.y;  return Math.sqrt(xDiff\*xDiff + yDiff\*yDiff);  }  } |

##### Test Driver Program: TestMyPoint.java

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16 | // A test driver for MyPoint class  public class TestMyPoint {  public static void main(String[] args) {  MyPoint p1 = new MyPoint();  System.out.println(p1);  p1.setXY(0, 3);  System.out.println(p1);    MyPoint p2 = new MyPoint(4, 0);  System.out.println(p2);    // Test distance() methods  System.out.println(p1.distance(4, 0));  System.out.println(p1.distance(p2));  }  } |



A class called MyCircle is designed as shown in the class diagram. It contains:

* Two private member variables: a radius (double) and a center (an instance of MyPoint, which we created earlier).
* A default (no-arg) constructor that construct a Circle at (0,0) with radius of 1.0.
* A constructor that constructs a Circle with the given xCenter, yCenter and radius.
* A constructor that constructs a Circle with the given instance of MyPoint as center; andradius.
* Getter and setter for the private variables center and radius.
* Methods getCenterX(), setCenterX(), setCenterXY(), etc.
* A toString() method that returns a string description of the instance in the format "center=(x,y) radius=r".
* A distance(MyCircle another) method that returns the distance from this Circle to the given MyCircle instance called another.

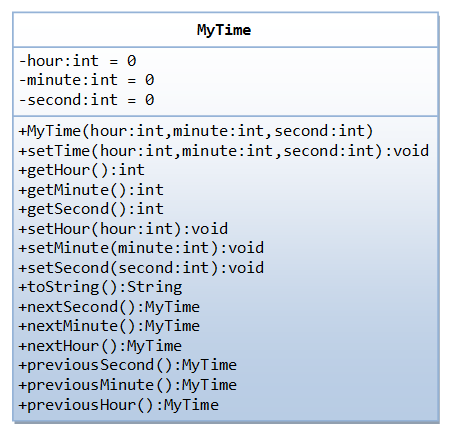
##### MyCircle.java

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42  43  44  45  46  47  48  49  50  51  52  53  54  55  56  57  58  59  60  61  62  63  64  65  66  67  68  69  70  71  72  73  74  75  76 | // The MyCircle class definition  public class MyCircle {  // Private member variables  private MyPoint center; // Declare instance center  private double radius;    // Constructors  public MyCircle() {  center = new MyPoint(); // Construct instance at (0,0)  radius = 1.0;  }    public MyCircle(int xCenter, int yCenter, double radius) {  center = new MyPoint(xCenter, yCenter); // Construct instance  this.radius = radius;  }    public MyCircle(MyPoint center, double radius) {  this.center = center;  this.radius = radius;  }    // Getters and Setters  public double getRadius() {  return radius;  }    public void setRadius(double radius) {  this.radius = radius;  }    public MyPoint getCenter() {  return center;  }    public void setCenter(MyPoint center) {  this.center = center;  }    public int getCenterX() {  return center.getX();  }    public void setCenterX(int x) {  center.setX(x);  }    public int getCenterY() {  return center.getY();  }    public void setCenterY(int y) {  center.setY(y);  }    public void setCenterXY(int x, int y) {  center.setX(x);  center.setY(y);  }    public String toString() {  return "center=" + center + " radius=" + radius;  }    public double getArea() {  return Math.PI \* radius \* radius;  }    public double getPerimeter() {  return 2.0 \* Math.PI \* radius;  }    public double distance(MyCircle another) {  return center.distance(another.center); // use distance() of MyPoint  }  } |

##### TestMyCircle.java

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16 | // A test driver for MyCircle class  public class TestMyCircle {  public static void main(String[] args) {  MyCircle c1 = new MyCircle();  System.out.println(c1);  c1.setCenterXY(0, 3);  System.out.println(c1);    MyPoint p1 = new MyPoint(4, 0);  MyCircle c2 = new MyCircle(p1, 9);  System.out.println(c2);    // Test distance() methods  System.out.println(c1.distance(c2));  }  } |

#### 3.6  The MyTime class



A class called MyTime, which models a time instance, is designed as shown in the class diagram.

It contains the following private instance variables:

* hour: between 00 to 23.
* minute: between 00 to 59.
* Second: between 00 to 59.

The constructor shall invoke the setTime() method (to be described later) to set the instance variable.

It contains the following public methods:

* setTime(int hour, int minute, int second): It shall check if the given hour, minute and second are valid before setting the instance variables.  
  (Advanced: Otherwise, it shall throw an IllegalArgumentException with the message "Invalid hour, minute, or second!".)
* Setters setHour(int hour), setMinute(int minute), setSecond(int second): It shall check if the parameters are valid, similar to the above.
* Getters getHour(), getMinute(), getSecond().
* toString(): returns "HH:MM:SS".
* nextSecond(): Update this instance to the next second and return this instance. Take note that thenextSecond() of 23:59:59 is 00:00:00.
* nextMinute(), nextHour(), previousSecond(), previousMinute(), previousHour(): similar to the above.

##### MyTime.java

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42  43  44  45  46  47  48  49  50  51  52  53  54  55  56  57  58  59  60  61  62  63  64  65  66  67  68  69  70  71  72  73  74  75  76  77  78 | // The MyTime class definition  public class MyTime { // "MyTime.java"  // Private member variables  private int hour; // 0-23  private int minute; // 0-59  private int second; // 0-59    // Constructor  public MyTime(int hour, int minute, int second) {  setTime(hour, minute, second);  }    void setTime(int hour, int minute, int second) {  setHour(hour);  setMinute(minute);  setSecond(second);  }    // Setters which validates input with exception handling  void setHour(int hour) {  if (hour >= 0 && hour <= 23) {  this.hour = hour;  } else {  throw new IllegalArgumentException("Invalid hour!");  }  }    void setMinute(int minute) {  if (minute >= 0 && minute <= 59) {  this.minute = minute;  } else {  throw new IllegalArgumentException("Invalid minute!");  }  }    void setSecond(int second) {  if (second >= 0 && second <= 59) {  this.second = second;  } else {  throw new IllegalArgumentException("Invalid second!");  }  }    // Getters  public int getHour() {  return hour;  }    public int getMinute() {  return minute;  }    public int getSecond() {  return second;  }    // Return description in the format "hh:mm:ss" with leading zeros  public String toString() {  return String.format("%02d:%02d:%02d", hour, minute, second);  }    // Increment this instance to the next second, return this instance  public MyTime nextSecond() {  ++second;  if (second == 60) {  second = 0;  ++minute;  }  if (minute == 60) {  minute = 0;  ++hour;  }  if (hour == 24) {  hour = 0;  }  return this; // Return this instance, to support cascaded operation  }  } |

##### Exception Handling

What to do if an invalid hour, minute or second was given in the constructor or setter? Print an error message? Abruptly terminate the program? Continue operation by setting the parameter to its default? This is a hard decision.

In Java, instead of printing an error message, you can throw an so-called Exception object (such as IllegalArgumentException) to the caller, and let the caller handles the exception. For example,

void setHour(int hour) {

if (hour >= 0 && hour <= 23) {

this.hour = hour;

} else {

throw new IllegalArgumentException("Invalid hour!");

}

}

The caller can use the try-catch construct to handle the exception (in the test driver below). For example,

try {

MyTime t3 = new MyTime(12, 69, 69);

// skip remaining statements in try, goto catch

System.out.println(t1);

} catch (IllegalArgumentException ex) {

ex.printStackTrace();

} // after catch, continue next statement

The statements in the try-clause will be executed. If all the statements are successful, the catch-clause is ignored. However, if one of the statement in the try-clause throws anIllegalArgumentException, the rest of try-clause will be skipped, and the execution transferred to the catch-clause. The program always continues to the next statement after the try-catch.

##### A Test Driver Class: TestMyTime.java

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23 | // A test driver program for MyTime  public class TestMyTime {  public static void main(String[] args) {  MyTime t1 = new MyTime(23, 59, 58);  System.out.println(t1);  System.out.println(t1.nextSecond());  System.out.println(t1.nextSecond().nextSecond().nextSecond());    // MyTime t2 = new MyTime(12, 69, 69); // abrupt termination  // NOT continue to next statement    // Handling exception gracefully  try {  MyTime t3 = new MyTime(12, 69, 69);  // skip remaining statements in try, goto catch  System.out.println(t1);  } catch (IllegalArgumentException ex) {  ex.printStackTrace();  } // after try or catch, continue next statement    System.out.println("Continue after exception!");  }  } |

Topic 1 **Java docs**    :

**Java™ Platform, Standard Edition 6  
API Specification**

This document is the API specification for version 6 of the Java™ Platform, Standard Edition.

**See:**   
          [**Description**](http://docs.oracle.com/javase/6/docs/api/overview-summary.html#overview_description)

|  |  |
| --- | --- |
| **Packages** | |
| [**java.applet**](http://docs.oracle.com/javase/6/docs/api/java/applet/package-summary.html) | Provides the classes necessary to create an applet and the classes an applet uses to communicate with its applet context. |
| [**java.awt**](http://docs.oracle.com/javase/6/docs/api/java/awt/package-summary.html) | Contains all of the classes for creating user interfaces and for painting graphics and images. |
| [**java.awt.color**](http://docs.oracle.com/javase/6/docs/api/java/awt/color/package-summary.html) | Provides classes for color spaces. |
| [**java.awt.datatransfer**](http://docs.oracle.com/javase/6/docs/api/java/awt/datatransfer/package-summary.html) | Provides interfaces and classes for transferring data between and within applications. |
| [**java.awt.dnd**](http://docs.oracle.com/javase/6/docs/api/java/awt/dnd/package-summary.html) | Drag and Drop is a direct manipulation gesture found in many Graphical User Interface systems that provides a mechanism to transfer information between two entities logically associated with presentation elements in the GUI. |
| [**java.awt.event**](http://docs.oracle.com/javase/6/docs/api/java/awt/event/package-summary.html) | Provides interfaces and classes for dealing with different types of events fired by AWT components. |
| [**java.awt.font**](http://docs.oracle.com/javase/6/docs/api/java/awt/font/package-summary.html) | Provides classes and interface relating to fonts. |
| [**java.awt.geom**](http://docs.oracle.com/javase/6/docs/api/java/awt/geom/package-summary.html) | Provides the Java 2D classes for defining and performing operations on objects related to two-dimensional geometry. |
| [**java.awt.im**](http://docs.oracle.com/javase/6/docs/api/java/awt/im/package-summary.html) | Provides classes and interfaces for the input method framework. |
| [**java.awt.im.spi**](http://docs.oracle.com/javase/6/docs/api/java/awt/im/spi/package-summary.html) | Provides interfaces that enable the development of input methods that can be used with any Java runtime environment. |
| [**java.awt.image**](http://docs.oracle.com/javase/6/docs/api/java/awt/image/package-summary.html) | Provides classes for creating and modifying images. |
| [**java.awt.image.renderable**](http://docs.oracle.com/javase/6/docs/api/java/awt/image/renderable/package-summary.html) | Provides classes and interfaces for producing rendering-independent images. |
| [**java.awt.print**](http://docs.oracle.com/javase/6/docs/api/java/awt/print/package-summary.html) | Provides classes and interfaces for a general printing API. |
| [**java.beans**](http://docs.oracle.com/javase/6/docs/api/java/beans/package-summary.html) | Contains classes related to developing *beans* -- components based on the JavaBeansTMarchitecture. |
| [**java.beans.beancontext**](http://docs.oracle.com/javase/6/docs/api/java/beans/beancontext/package-summary.html) | Provides classes and interfaces relating to bean context. |
| [**java.io**](http://docs.oracle.com/javase/6/docs/api/java/io/package-summary.html) | Provides for system input and output through data streams, serialization and the file system. |
| [**java.lang**](http://docs.oracle.com/javase/6/docs/api/java/lang/package-summary.html) | Provides classes that are fundamental to the design of the Java programming language. |
| [**java.lang.annotation**](http://docs.oracle.com/javase/6/docs/api/java/lang/annotation/package-summary.html) | Provides library support for the Java programming language annotation facility. |
| [**java.lang.instrument**](http://docs.oracle.com/javase/6/docs/api/java/lang/instrument/package-summary.html) | Provides services that allow Java programming language agents to instrument programs running on the JVM. |
| [**java.lang.management**](http://docs.oracle.com/javase/6/docs/api/java/lang/management/package-summary.html) | Provides the management interface for monitoring and management of the Java virtual machine as well as the operating system on which the Java virtual machine is running. |
| [**java.lang.ref**](http://docs.oracle.com/javase/6/docs/api/java/lang/ref/package-summary.html) | Provides reference-object classes, which support a limited degree of interaction with the garbage collector. |
| [**java.lang.reflect**](http://docs.oracle.com/javase/6/docs/api/java/lang/reflect/package-summary.html) | Provides classes and interfaces for obtaining reflective information about classes and objects. |
| [**java.math**](http://docs.oracle.com/javase/6/docs/api/java/math/package-summary.html) | Provides classes for performing arbitrary-precision integer arithmetic (BigInteger) and arbitrary-precision decimal arithmetic (BigDecimal). |
| [**java.net**](http://docs.oracle.com/javase/6/docs/api/java/net/package-summary.html) | Provides the classes for implementing networking applications. |
| [**java.nio**](http://docs.oracle.com/javase/6/docs/api/java/nio/package-summary.html) | Defines buffers, which are containers for data, and provides an overview of the other NIO packages. |
| [**java.nio.channels**](http://docs.oracle.com/javase/6/docs/api/java/nio/channels/package-summary.html) | Defines channels, which represent connections to entities that are capable of performing I/O operations, such as files and sockets; defines selectors, for multiplexed, non-blocking I/O operations. |
| [**java.nio.channels.spi**](http://docs.oracle.com/javase/6/docs/api/java/nio/channels/spi/package-summary.html) | Service-provider classes for the [java.nio.channels](http://docs.oracle.com/javase/6/docs/api/java/nio/channels/package-summary.html) package. |
| [**java.nio.charset**](http://docs.oracle.com/javase/6/docs/api/java/nio/charset/package-summary.html) | Defines charsets, decoders, and encoders, for translating between bytes and Unicode characters. |
| [**java.nio.charset.spi**](http://docs.oracle.com/javase/6/docs/api/java/nio/charset/spi/package-summary.html) | Service-provider classes for the [java.nio.charset](http://docs.oracle.com/javase/6/docs/api/java/nio/charset/package-summary.html) package. |
| [**java.rmi**](http://docs.oracle.com/javase/6/docs/api/java/rmi/package-summary.html) | Provides the RMI package. |
| [**java.rmi.activation**](http://docs.oracle.com/javase/6/docs/api/java/rmi/activation/package-summary.html) | Provides support for RMI Object Activation. |
| [**java.rmi.dgc**](http://docs.oracle.com/javase/6/docs/api/java/rmi/dgc/package-summary.html) | Provides classes and interface for RMI distributed garbage-collection (DGC). |
| [**java.rmi.registry**](http://docs.oracle.com/javase/6/docs/api/java/rmi/registry/package-summary.html) | Provides a class and two interfaces for the RMI registry. |
| [**java.rmi.server**](http://docs.oracle.com/javase/6/docs/api/java/rmi/server/package-summary.html) | Provides classes and interfaces for supporting the server side of RMI. |
| [**java.security**](http://docs.oracle.com/javase/6/docs/api/java/security/package-summary.html) | Provides the classes and interfaces for the security framework. |
| [**java.security.acl**](http://docs.oracle.com/javase/6/docs/api/java/security/acl/package-summary.html) | The classes and interfaces in this package have been superseded by classes in the java.security package. |
| [**java.security.cert**](http://docs.oracle.com/javase/6/docs/api/java/security/cert/package-summary.html) | Provides classes and interfaces for parsing and managing certificates, certificate revocation lists (CRLs), and certification paths. |
| [**java.security.interfaces**](http://docs.oracle.com/javase/6/docs/api/java/security/interfaces/package-summary.html) | Provides interfaces for generating RSA (Rivest, Shamir and Adleman AsymmetricCipher algorithm) keys as defined in the RSA Laboratory Technical Note PKCS#1, and DSA (Digital Signature Algorithm) keys as defined in NIST's FIPS-186. |
| [**java.security.spec**](http://docs.oracle.com/javase/6/docs/api/java/security/spec/package-summary.html) | Provides classes and interfaces for key specifications and algorithm parameter specifications. |
| [**java.sql**](http://docs.oracle.com/javase/6/docs/api/java/sql/package-summary.html) | Provides the API for accessing and processing data stored in a data source (usually a relational database) using the JavaTM programming language. |
| [**java.text**](http://docs.oracle.com/javase/6/docs/api/java/text/package-summary.html) | Provides classes and interfaces for handling text, dates, numbers, and messages in a manner independent of natural languages. |
| [**java.text.spi**](http://docs.oracle.com/javase/6/docs/api/java/text/spi/package-summary.html) | Service provider classes for the classes in the java.text package. |
| [**java.util**](http://docs.oracle.com/javase/6/docs/api/java/util/package-summary.html) | Contains the collections framework, legacy collection classes, event model, date and time facilities, internationalization, and miscellaneous utility classes (a string tokenizer, a random-number generator, and a bit array). |
| [**java.util.concurrent**](http://docs.oracle.com/javase/6/docs/api/java/util/concurrent/package-summary.html) | Utility classes commonly useful in concurrent programming. |
| [**java.util.concurrent.atomic**](http://docs.oracle.com/javase/6/docs/api/java/util/concurrent/atomic/package-summary.html) | A small toolkit of classes that support lock-free thread-safe programming on single variables. |
| [**java.util.concurrent.locks**](http://docs.oracle.com/javase/6/docs/api/java/util/concurrent/locks/package-summary.html) | Interfaces and classes providing a framework for locking and waiting for conditions that is distinct from built-in synchronization and monitors. |
| [**java.util.jar**](http://docs.oracle.com/javase/6/docs/api/java/util/jar/package-summary.html) | Provides classes for reading and writing the JAR (Java ARchive) file format, which is based on the standard ZIP file format with an optional manifest file. |
| [**java.util.logging**](http://docs.oracle.com/javase/6/docs/api/java/util/logging/package-summary.html) | Provides the classes and interfaces of the JavaTM 2 platform's core logging facilities. |
| [**java.util.prefs**](http://docs.oracle.com/javase/6/docs/api/java/util/prefs/package-summary.html) | This package allows applications to store and retrieve user and system preference and configuration data. |
| [**java.util.regex**](http://docs.oracle.com/javase/6/docs/api/java/util/regex/package-summary.html) | Classes for matching character sequences against patterns specified by regular expressions. |
| [**java.util.spi**](http://docs.oracle.com/javase/6/docs/api/java/util/spi/package-summary.html) | Service provider classes for the classes in the java.util package. |
| [**java.util.zip**](http://docs.oracle.com/javase/6/docs/api/java/util/zip/package-summary.html) | Provides classes for reading and writing the standard ZIP and GZIP file formats. |
| [**javax.accessibility**](http://docs.oracle.com/javase/6/docs/api/javax/accessibility/package-summary.html) | Defines a contract between user-interface components and an assistive technology that provides access to those components. |
| [**javax.activation**](http://docs.oracle.com/javase/6/docs/api/javax/activation/package-summary.html) |  |
| [**javax.activity**](http://docs.oracle.com/javase/6/docs/api/javax/activity/package-summary.html) | Contains Activity service related exceptions thrown by the ORB machinery during unmarshalling. |
| [**javax.annotation**](http://docs.oracle.com/javase/6/docs/api/javax/annotation/package-summary.html) |  |
| [**javax.annotation.processing**](http://docs.oracle.com/javase/6/docs/api/javax/annotation/processing/package-summary.html) | Facilities for declaring annotation processors and for allowing annotation processors to communicate with an annotation processing tool environment. |
| [**javax.crypto**](http://docs.oracle.com/javase/6/docs/api/javax/crypto/package-summary.html) | Provides the classes and interfaces for cryptographic operations. |
| [**javax.crypto.interfaces**](http://docs.oracle.com/javase/6/docs/api/javax/crypto/interfaces/package-summary.html) | Provides interfaces for Diffie-Hellman keys as defined in RSA Laboratories' PKCS #3. |
| [**javax.crypto.spec**](http://docs.oracle.com/javase/6/docs/api/javax/crypto/spec/package-summary.html) | Provides classes and interfaces for key specifications and algorithm parameter specifications. |
| [**javax.imageio**](http://docs.oracle.com/javase/6/docs/api/javax/imageio/package-summary.html) | The main package of the Java Image I/O API. |
| [**javax.imageio.event**](http://docs.oracle.com/javase/6/docs/api/javax/imageio/event/package-summary.html) | A package of the Java Image I/O API dealing with synchronous notification of events during the reading and writing of images. |
| [**javax.imageio.metadata**](http://docs.oracle.com/javase/6/docs/api/javax/imageio/metadata/package-summary.html) | A package of the Java Image I/O API dealing with reading and writing metadata. |
| [**javax.imageio.plugins.bmp**](http://docs.oracle.com/javase/6/docs/api/javax/imageio/plugins/bmp/package-summary.html) | Package containing the public classes used by the built-in BMP plug-in. |
| [**javax.imageio.plugins.jpeg**](http://docs.oracle.com/javase/6/docs/api/javax/imageio/plugins/jpeg/package-summary.html) | Classes supporting the built-in JPEG plug-in. |
| [**javax.imageio.spi**](http://docs.oracle.com/javase/6/docs/api/javax/imageio/spi/package-summary.html) | A package of the Java Image I/O API containing the plug-in interfaces for readers, writers, transcoders, and streams, and a runtime registry. |
| [**javax.imageio.stream**](http://docs.oracle.com/javase/6/docs/api/javax/imageio/stream/package-summary.html) | A package of the Java Image I/O API dealing with low-level I/O from files and streams. |
| [**javax.jws**](http://docs.oracle.com/javase/6/docs/api/javax/jws/package-summary.html) |  |
| [**javax.jws.soap**](http://docs.oracle.com/javase/6/docs/api/javax/jws/soap/package-summary.html) |  |
| [**javax.lang.model**](http://docs.oracle.com/javase/6/docs/api/javax/lang/model/package-summary.html) | Classes and hierarchies of packages used to model the Java programming language. |
| [**javax.lang.model.element**](http://docs.oracle.com/javase/6/docs/api/javax/lang/model/element/package-summary.html) | Interfaces used to model elements of the Java programming language. |
| [**javax.lang.model.type**](http://docs.oracle.com/javase/6/docs/api/javax/lang/model/type/package-summary.html) | Interfaces used to model Java programming language types. |
| [**javax.lang.model.util**](http://docs.oracle.com/javase/6/docs/api/javax/lang/model/util/package-summary.html) | Utilities to assist in the processing of [program elements](http://docs.oracle.com/javase/6/docs/api/javax/lang/model/element/package-summary.html) and [types](http://docs.oracle.com/javase/6/docs/api/javax/lang/model/type/package-summary.html). |
| [**javax.management**](http://docs.oracle.com/javase/6/docs/api/javax/management/package-summary.html) | Provides the core classes for the Java Management Extensions. |
| [**javax.management.loading**](http://docs.oracle.com/javase/6/docs/api/javax/management/loading/package-summary.html) | Provides the classes which implement advanced dynamic loading. |
| [**javax.management.modelmbean**](http://docs.oracle.com/javase/6/docs/api/javax/management/modelmbean/package-summary.html) | Provides the definition of the ModelMBean classes. |
| [**javax.management.monitor**](http://docs.oracle.com/javase/6/docs/api/javax/management/monitor/package-summary.html) | Provides the definition of the monitor classes. |
| [**javax.management.openmbean**](http://docs.oracle.com/javase/6/docs/api/javax/management/openmbean/package-summary.html) | Provides the open data types and Open MBean descriptor classes. |
| [**javax.management.relation**](http://docs.oracle.com/javase/6/docs/api/javax/management/relation/package-summary.html) | Provides the definition of the Relation Service. |
| [**javax.management.remote**](http://docs.oracle.com/javase/6/docs/api/javax/management/remote/package-summary.html) | Interfaces for remote access to JMX MBean servers. |
| [**javax.management.remote.rmi**](http://docs.oracle.com/javase/6/docs/api/javax/management/remote/rmi/package-summary.html) | The RMI connector is a connector for the JMX Remote API that uses RMI to transmit client requests to a remote MBean server. |
| [**javax.management.timer**](http://docs.oracle.com/javase/6/docs/api/javax/management/timer/package-summary.html) | Provides the definition of the Timer MBean. |
| [**javax.naming**](http://docs.oracle.com/javase/6/docs/api/javax/naming/package-summary.html) | Provides the classes and interfaces for accessing naming services. |
| [**javax.naming.directory**](http://docs.oracle.com/javase/6/docs/api/javax/naming/directory/package-summary.html) | Extends the javax.naming package to provide functionality for accessing directory services. |
| [**javax.naming.event**](http://docs.oracle.com/javase/6/docs/api/javax/naming/event/package-summary.html) | Provides support for event notification when accessing naming and directory services. |
| [**javax.naming.ldap**](http://docs.oracle.com/javase/6/docs/api/javax/naming/ldap/package-summary.html) | Provides support for LDAPv3 extended operations and controls. |
| [**javax.naming.spi**](http://docs.oracle.com/javase/6/docs/api/javax/naming/spi/package-summary.html) | Provides the means for dynamically plugging in support for accessing naming and directory services through the javax.naming and related packages. |
| [**javax.net**](http://docs.oracle.com/javase/6/docs/api/javax/net/package-summary.html) | Provides classes for networking applications. |
| [**javax.net.ssl**](http://docs.oracle.com/javase/6/docs/api/javax/net/ssl/package-summary.html) | Provides classes for the secure socket package. |
| [**javax.print**](http://docs.oracle.com/javase/6/docs/api/javax/print/package-summary.html) | Provides the principal classes and interfaces for the JavaTM Print Service API. |
| [**javax.print.attribute**](http://docs.oracle.com/javase/6/docs/api/javax/print/attribute/package-summary.html) | Provides classes and interfaces that describe the types of JavaTM Print Service attributes and how they can be collected into attribute sets. |
| [**javax.print.attribute.standard**](http://docs.oracle.com/javase/6/docs/api/javax/print/attribute/standard/package-summary.html) | Package javax.print.attribute.standard contains classes for specific printing attributes. |
| [**javax.print.event**](http://docs.oracle.com/javase/6/docs/api/javax/print/event/package-summary.html) | Package javax.print.event contains event classes and listener interfaces. |
| [**javax.rmi**](http://docs.oracle.com/javase/6/docs/api/javax/rmi/package-summary.html) | Contains user APIs for RMI-IIOP. |
| [**javax.rmi.CORBA**](http://docs.oracle.com/javase/6/docs/api/javax/rmi/CORBA/package-summary.html) | Contains portability APIs for RMI-IIOP. |
| [**javax.rmi.ssl**](http://docs.oracle.com/javase/6/docs/api/javax/rmi/ssl/package-summary.html) | Provides implementations of [RMIClientSocketFactory](http://docs.oracle.com/javase/6/docs/api/java/rmi/server/RMIClientSocketFactory.html" \o "interface in java.rmi.server) and [RMIServerSocketFactory](http://docs.oracle.com/javase/6/docs/api/java/rmi/server/RMIServerSocketFactory.html" \o "interface in java.rmi.server) over the Secure Sockets Layer (SSL) or Transport Layer Security (TLS) protocols. |
| [**javax.script**](http://docs.oracle.com/javase/6/docs/api/javax/script/package-summary.html) | The scripting API consists of interfaces and classes that define Java TM Scripting Engines and provides a framework for their use in Java applications. |
| [**javax.security.auth**](http://docs.oracle.com/javase/6/docs/api/javax/security/auth/package-summary.html) | This package provides a framework for authentication and authorization. |
| [**javax.security.auth.callback**](http://docs.oracle.com/javase/6/docs/api/javax/security/auth/callback/package-summary.html) | This package provides the classes necessary for services to interact with applications in order to retrieve information (authentication data including usernames or passwords, for example) or to display information (error and warning messages, for example). |
| [**javax.security.auth.kerberos**](http://docs.oracle.com/javase/6/docs/api/javax/security/auth/kerberos/package-summary.html) | This package contains utility classes related to the Kerberos network authentication protocol. |
| [**javax.security.auth.login**](http://docs.oracle.com/javase/6/docs/api/javax/security/auth/login/package-summary.html) | This package provides a pluggable authentication framework. |
| [**javax.security.auth.spi**](http://docs.oracle.com/javase/6/docs/api/javax/security/auth/spi/package-summary.html) | This package provides the interface to be used for implementing pluggable authentication modules. |
| [**javax.security.auth.x500**](http://docs.oracle.com/javase/6/docs/api/javax/security/auth/x500/package-summary.html) | This package contains the classes that should be used to store X500 Principal and X500 Private Crendentials in a *Subject*. |
| [**javax.security.cert**](http://docs.oracle.com/javase/6/docs/api/javax/security/cert/package-summary.html) | Provides classes for public key certificates. |
| [**javax.security.sasl**](http://docs.oracle.com/javase/6/docs/api/javax/security/sasl/package-summary.html) | Contains class and interfaces for supporting SASL. |
| [**javax.sound.midi**](http://docs.oracle.com/javase/6/docs/api/javax/sound/midi/package-summary.html) | Provides interfaces and classes for I/O, sequencing, and synthesis of MIDI (Musical Instrument Digital Interface) data. |
| [**javax.sound.midi.spi**](http://docs.oracle.com/javase/6/docs/api/javax/sound/midi/spi/package-summary.html) | Supplies interfaces for service providers to implement when offering new MIDI devices, MIDI file readers and writers, or sound bank readers. |
| [**javax.sound.sampled**](http://docs.oracle.com/javase/6/docs/api/javax/sound/sampled/package-summary.html) | Provides interfaces and classes for capture, processing, and playback of sampled audio data. |
| [**javax.sound.sampled.spi**](http://docs.oracle.com/javase/6/docs/api/javax/sound/sampled/spi/package-summary.html) | Supplies abstract classes for service providers to subclass when offering new audio devices, sound file readers and writers, or audio format converters. |
| [**javax.sql**](http://docs.oracle.com/javase/6/docs/api/javax/sql/package-summary.html) | Provides the API for server side data source access and processing from the JavaTMprogramming language. |
| [**javax.sql.rowset**](http://docs.oracle.com/javase/6/docs/api/javax/sql/rowset/package-summary.html) | Standard interfaces and base classes for JDBC RowSet implementations. |
| [**javax.sql.rowset.serial**](http://docs.oracle.com/javase/6/docs/api/javax/sql/rowset/serial/package-summary.html) | Provides utility classes to allow serializable mappings between SQL types and data types in the Java programming language. |
| [**javax.sql.rowset.spi**](http://docs.oracle.com/javase/6/docs/api/javax/sql/rowset/spi/package-summary.html) | The standard classes and interfaces that a third party vendor has to use in its implementation of a synchronization provider. |
| [**javax.swing**](http://docs.oracle.com/javase/6/docs/api/javax/swing/package-summary.html) | Provides a set of "lightweight" (all-Java language) components that, to the maximum degree possible, work the same on all platforms. |
| [**javax.swing.border**](http://docs.oracle.com/javase/6/docs/api/javax/swing/border/package-summary.html) | Provides classes and interface for drawing specialized borders around a Swing component. |
| [**javax.swing.colorchooser**](http://docs.oracle.com/javase/6/docs/api/javax/swing/colorchooser/package-summary.html) | Contains classes and interfaces used by the JColorChooser component. |
| [**javax.swing.event**](http://docs.oracle.com/javase/6/docs/api/javax/swing/event/package-summary.html) | Provides for events fired by Swing components. |
| [**javax.swing.filechooser**](http://docs.oracle.com/javase/6/docs/api/javax/swing/filechooser/package-summary.html) | Contains classes and interfaces used by the JFileChooser component. |
| [**javax.swing.plaf**](http://docs.oracle.com/javase/6/docs/api/javax/swing/plaf/package-summary.html) | Provides one interface and many abstract classes that Swing uses to provide its pluggable look-and-feel capabilities. |
| [**javax.swing.plaf.basic**](http://docs.oracle.com/javase/6/docs/api/javax/swing/plaf/basic/package-summary.html) | Provides user interface objects built according to the Basic look and feel. |
| [**javax.swing.plaf.metal**](http://docs.oracle.com/javase/6/docs/api/javax/swing/plaf/metal/package-summary.html) | Provides user interface objects built according to the Java look and feel (once codenamed*Metal*), which is the default look and feel. |
| [**javax.swing.plaf.multi**](http://docs.oracle.com/javase/6/docs/api/javax/swing/plaf/multi/package-summary.html) | Provides user interface objects that combine two or more look and feels. |
| [**javax.swing.plaf.synth**](http://docs.oracle.com/javase/6/docs/api/javax/swing/plaf/synth/package-summary.html) | Synth is a skinnable look and feel in which all painting is delegated. |
| [**javax.swing.table**](http://docs.oracle.com/javase/6/docs/api/javax/swing/table/package-summary.html) | Provides classes and interfaces for dealing with javax.swing.JTable. |
| [**javax.swing.text**](http://docs.oracle.com/javase/6/docs/api/javax/swing/text/package-summary.html) | Provides classes and interfaces that deal with editable and noneditable text components. |
| [**javax.swing.text.html**](http://docs.oracle.com/javase/6/docs/api/javax/swing/text/html/package-summary.html) | Provides the class HTMLEditorKit and supporting classes for creating HTML text editors. |
| [**javax.swing.text.html.parser**](http://docs.oracle.com/javase/6/docs/api/javax/swing/text/html/parser/package-summary.html) | Provides the default HTML parser, along with support classes. |
| [**javax.swing.text.rtf**](http://docs.oracle.com/javase/6/docs/api/javax/swing/text/rtf/package-summary.html) | Provides a class (RTFEditorKit) for creating Rich-Text-Format text editors. |
| [**javax.swing.tree**](http://docs.oracle.com/javase/6/docs/api/javax/swing/tree/package-summary.html) | Provides classes and interfaces for dealing with javax.swing.JTree. |
| [**javax.swing.undo**](http://docs.oracle.com/javase/6/docs/api/javax/swing/undo/package-summary.html) | Allows developers to provide support for undo/redo in applications such as text editors. |
| [**javax.tools**](http://docs.oracle.com/javase/6/docs/api/javax/tools/package-summary.html) | Provides interfaces for tools which can be invoked from a program, for example, compilers. |
| [**javax.transaction**](http://docs.oracle.com/javase/6/docs/api/javax/transaction/package-summary.html) | Contains three exceptions thrown by the ORB machinery during unmarshalling. |
| [**javax.transaction.xa**](http://docs.oracle.com/javase/6/docs/api/javax/transaction/xa/package-summary.html) | Provides the API that defines the contract between the transaction manager and the resource manager, which allows the transaction manager to enlist and delist resource objects (supplied by the resource manager driver) in JTA transactions. |
| [**javax.xml**](http://docs.oracle.com/javase/6/docs/api/javax/xml/package-summary.html) | Defines core XML constants and functionality from the XML specifications. |
| [**javax.xml.bind**](http://docs.oracle.com/javase/6/docs/api/javax/xml/bind/package-summary.html) | Provides a runtime binding framework for client applications including unmarshalling, marshalling, and validation capabilities. |
| [**javax.xml.bind.annotation**](http://docs.oracle.com/javase/6/docs/api/javax/xml/bind/annotation/package-summary.html) | Defines annotations for customizing Java program elements to XML Schema mapping. |
| [**javax.xml.bind.annotation.adapters**](http://docs.oracle.com/javase/6/docs/api/javax/xml/bind/annotation/adapters/package-summary.html) | [XmlAdapter](http://docs.oracle.com/javase/6/docs/api/javax/xml/bind/annotation/adapters/XmlAdapter.html) and its spec-defined sub-classes to allow arbitrary Java classes to be used with JAXB. |
| [**javax.xml.bind.attachment**](http://docs.oracle.com/javase/6/docs/api/javax/xml/bind/attachment/package-summary.html) | This package is implemented by a MIME-based package processor that enables the interpretation and creation of optimized binary data within an MIME-based package format. |
| [**javax.xml.bind.helpers**](http://docs.oracle.com/javase/6/docs/api/javax/xml/bind/helpers/package-summary.html) | **JAXB Provider Use Only:** Provides partial default implementations for some of thejavax.xml.bind interfaces. |
| [**javax.xml.bind.util**](http://docs.oracle.com/javase/6/docs/api/javax/xml/bind/util/package-summary.html) | Useful client utility classes. |
| [**javax.xml.crypto**](http://docs.oracle.com/javase/6/docs/api/javax/xml/crypto/package-summary.html) | Common classes for XML cryptography. |
| [**javax.xml.crypto.dom**](http://docs.oracle.com/javase/6/docs/api/javax/xml/crypto/dom/package-summary.html) | DOM-specific classes for the [javax.xml.crypto](http://docs.oracle.com/javase/6/docs/api/javax/xml/crypto/package-summary.html) package. |
| [**javax.xml.crypto.dsig**](http://docs.oracle.com/javase/6/docs/api/javax/xml/crypto/dsig/package-summary.html) | Classes for generating and validating XML digital signatures. |
| [**javax.xml.crypto.dsig.dom**](http://docs.oracle.com/javase/6/docs/api/javax/xml/crypto/dsig/dom/package-summary.html) | DOM-specific classes for the [javax.xml.crypto.dsig](http://docs.oracle.com/javase/6/docs/api/javax/xml/crypto/dsig/package-summary.html) package. |
| [**javax.xml.crypto.dsig.keyinfo**](http://docs.oracle.com/javase/6/docs/api/javax/xml/crypto/dsig/keyinfo/package-summary.html) | Classes for parsing and processing [KeyInfo](http://docs.oracle.com/javase/6/docs/api/javax/xml/crypto/dsig/keyinfo/KeyInfo.html" \o "interface in javax.xml.crypto.dsig.keyinfo) elements and structures. |
| [**javax.xml.crypto.dsig.spec**](http://docs.oracle.com/javase/6/docs/api/javax/xml/crypto/dsig/spec/package-summary.html) | Parameter classes for XML digital signatures. |
| [**javax.xml.datatype**](http://docs.oracle.com/javase/6/docs/api/javax/xml/datatype/package-summary.html) | XML/Java Type Mappings. |
| [**javax.xml.namespace**](http://docs.oracle.com/javase/6/docs/api/javax/xml/namespace/package-summary.html) | XML Namespace processing. |
| [**javax.xml.parsers**](http://docs.oracle.com/javase/6/docs/api/javax/xml/parsers/package-summary.html) | Provides classes allowing the processing of XML documents. |
| [**javax.xml.soap**](http://docs.oracle.com/javase/6/docs/api/javax/xml/soap/package-summary.html) | Provides the API for creating and building SOAP messages. |
| [**javax.xml.stream**](http://docs.oracle.com/javase/6/docs/api/javax/xml/stream/package-summary.html) |  |
| [**javax.xml.stream.events**](http://docs.oracle.com/javase/6/docs/api/javax/xml/stream/events/package-summary.html) |  |
| [**javax.xml.stream.util**](http://docs.oracle.com/javase/6/docs/api/javax/xml/stream/util/package-summary.html) |  |
| [**javax.xml.transform**](http://docs.oracle.com/javase/6/docs/api/javax/xml/transform/package-summary.html) | This package defines the generic APIs for processing transformation instructions, and performing a transformation from source to result. |
| [**javax.xml.transform.dom**](http://docs.oracle.com/javase/6/docs/api/javax/xml/transform/dom/package-summary.html) | This package implements DOM-specific transformation APIs. |
| [**javax.xml.transform.sax**](http://docs.oracle.com/javase/6/docs/api/javax/xml/transform/sax/package-summary.html) | This package implements SAX2-specific transformation APIs. |
| [**javax.xml.transform.stax**](http://docs.oracle.com/javase/6/docs/api/javax/xml/transform/stax/package-summary.html) | Provides for StAX-specific transformation APIs. |
| [**javax.xml.transform.stream**](http://docs.oracle.com/javase/6/docs/api/javax/xml/transform/stream/package-summary.html) | This package implements stream- and URI- specific transformation APIs. |
| [**javax.xml.validation**](http://docs.oracle.com/javase/6/docs/api/javax/xml/validation/package-summary.html) | This package provides an API for validation of XML documents. |
| [**javax.xml.ws**](http://docs.oracle.com/javase/6/docs/api/javax/xml/ws/package-summary.html) | This package contains the core JAX-WS APIs. |
| [**javax.xml.ws.handler**](http://docs.oracle.com/javase/6/docs/api/javax/xml/ws/handler/package-summary.html) | This package defines APIs for message handlers. |
| [**javax.xml.ws.handler.soap**](http://docs.oracle.com/javase/6/docs/api/javax/xml/ws/handler/soap/package-summary.html) | This package defines APIs for SOAP message handlers. |
| [**javax.xml.ws.http**](http://docs.oracle.com/javase/6/docs/api/javax/xml/ws/http/package-summary.html) | This package defines APIs specific to the HTTP binding. |
| [**javax.xml.ws.soap**](http://docs.oracle.com/javase/6/docs/api/javax/xml/ws/soap/package-summary.html) | This package defines APIs specific to the SOAP binding. |
| [**javax.xml.ws.spi**](http://docs.oracle.com/javase/6/docs/api/javax/xml/ws/spi/package-summary.html) | This package defines SPIs for JAX-WS. |
| [**javax.xml.ws.wsaddressing**](http://docs.oracle.com/javase/6/docs/api/javax/xml/ws/wsaddressing/package-summary.html) | This package defines APIs related to WS-Addressing. |
| [**javax.xml.xpath**](http://docs.oracle.com/javase/6/docs/api/javax/xml/xpath/package-summary.html) | This package provides an *object-model neutral* API for the evaluation of XPath expressions and access to the evaluation environment. |
| [**org.ietf.jgss**](http://docs.oracle.com/javase/6/docs/api/org/ietf/jgss/package-summary.html) | This package presents a framework that allows application developers to make use of security services like authentication, data integrity and data confidentiality from a variety of underlying security mechanisms like Kerberos, using a unified API. |
| [**org.omg.CORBA**](http://docs.oracle.com/javase/6/docs/api/org/omg/CORBA/package-summary.html) | Provides the mapping of the OMG CORBA APIs to the JavaTM programming language, including the class ORB, which is implemented so that a programmer can use it as a fully-functional Object Request Broker (ORB). |
| [**org.omg.CORBA\_2\_3**](http://docs.oracle.com/javase/6/docs/api/org/omg/CORBA_2_3/package-summary.html) | The CORBA\_2\_3 package defines additions to existing CORBA interfaces in the Java[tm] Standard Edition 6.   These changes occurred in recent revisions to the CORBA API defined by the OMG.  The new methods were added to  interfaces derived from the corresponding interfaces in the CORBA package.  This provides backward compatibility and avoids breaking the JCK tests. |
| [**org.omg.CORBA\_2\_3.portable**](http://docs.oracle.com/javase/6/docs/api/org/omg/CORBA_2_3/portable/package-summary.html) | Provides methods for the input and output of value types, and contains other updates to theorg/omg/CORBA/portable package. |
| [**org.omg.CORBA.DynAnyPackage**](http://docs.oracle.com/javase/6/docs/api/org/omg/CORBA/DynAnyPackage/package-summary.html) | Provides the exceptions used with the DynAny interface (InvalidValue, Invalid, InvalidSeq, andTypeMismatch). |
| [**org.omg.CORBA.ORBPackage**](http://docs.oracle.com/javase/6/docs/api/org/omg/CORBA/ORBPackage/package-summary.html) | Provides the exception InvalidName, which is thrown by the methodORB.resolve\_initial\_references and the exception InconsistentTypeCode, which is thrown by the Dynamic Any creation methods in the ORB class. |
| [**org.omg.CORBA.portable**](http://docs.oracle.com/javase/6/docs/api/org/omg/CORBA/portable/package-summary.html) | Provides a portability layer, that is, a set of ORB APIs that makes it possible for code generated by one vendor to run on another vendor's ORB. |
| [**org.omg.CORBA.TypeCodePackage**](http://docs.oracle.com/javase/6/docs/api/org/omg/CORBA/TypeCodePackage/package-summary.html) | Provides the user-defined exceptions BadKind and Bounds, which are thrown by methods in in the class TypeCode. |
| [**org.omg.CosNaming**](http://docs.oracle.com/javase/6/docs/api/org/omg/CosNaming/package-summary.html) | Provides a naming service for Java IDL. |
| [**org.omg.CosNaming.NamingContextExtPackage**](http://docs.oracle.com/javase/6/docs/api/org/omg/CosNaming/NamingContextExtPackage/package-summary.html) | This package contains the following classes, which are used inorg.omg.CosNaming.NamingContextExt: |
| [**org.omg.CosNaming.NamingContextPackage**](http://docs.oracle.com/javase/6/docs/api/org/omg/CosNaming/NamingContextPackage/package-summary.html) | This package contains Exception classes for the org.omg.CosNaming package. |
| [**org.omg.Dynamic**](http://docs.oracle.com/javase/6/docs/api/org/omg/Dynamic/package-summary.html) | This package contains the Dynamic module specified in the OMG Portable Interceptor specification, <http://cgi.omg.org/cgi-bin/doc?ptc/2000-08-06>, section 21.9. |
| [**org.omg.DynamicAny**](http://docs.oracle.com/javase/6/docs/api/org/omg/DynamicAny/package-summary.html) | Provides classes and interfaces that enable traversal of the data value associated with an any at runtime, and extraction of the primitive constituents of the data value. |
| [**org.omg.DynamicAny.DynAnyFactoryPackage**](http://docs.oracle.com/javase/6/docs/api/org/omg/DynamicAny/DynAnyFactoryPackage/package-summary.html) | This package contains classes and exceptions from the DynAnyFactory interface of the DynamicAnymodule specified in the OMG *The Common Object Request Broker: Architecture and Specification*, <http://cgi.omg.org/cgi-bin/doc?formal/99-10-07>, section 9.2.2. |
| [**org.omg.DynamicAny.DynAnyPackage**](http://docs.oracle.com/javase/6/docs/api/org/omg/DynamicAny/DynAnyPackage/package-summary.html) | This package contains classes and exceptions from the DynAny interface of the DynamicAnymodule specified in the OMG *The Common Object Request Broker: Architecture and Specification*, <http://cgi.omg.org/cgi-bin/doc?formal/99-10-07>, section 9.2. |
| [**org.omg.IOP**](http://docs.oracle.com/javase/6/docs/api/org/omg/IOP/package-summary.html) | This package contains the IOP module specified in the OMG document *The Common Object Request Broker: Architecture and Specification*, <http://cgi.omg.org/cgi-bin/doc?formal/99-10-07>, section 13.6. |
| [**org.omg.IOP.CodecFactoryPackage**](http://docs.oracle.com/javase/6/docs/api/org/omg/IOP/CodecFactoryPackage/package-summary.html) | This package contains the exceptions specified in the IOP::CodeFactory interface (as part of the Portable Interceptors spec). |
| [**org.omg.IOP.CodecPackage**](http://docs.oracle.com/javase/6/docs/api/org/omg/IOP/CodecPackage/package-summary.html) | This package is generated from the IOP::Codec IDL interface definition. |
| [**org.omg.Messaging**](http://docs.oracle.com/javase/6/docs/api/org/omg/Messaging/package-summary.html) | This package contains the Messaging module specified in the OMG CORBA Messaging specification, <http://cgi.omg.org/cgi-bin/doc?formal/99-10-07>. |
| [**org.omg.PortableInterceptor**](http://docs.oracle.com/javase/6/docs/api/org/omg/PortableInterceptor/package-summary.html) | Provides a mechanism to register ORB hooks through which ORB services can intercept the normal flow of execution of the ORB. |
| [**org.omg.PortableInterceptor.ORBInitInfoPackage**](http://docs.oracle.com/javase/6/docs/api/org/omg/PortableInterceptor/ORBInitInfoPackage/package-summary.html) | This package contains the exceptions and typedefs from the ORBInitInfo local interface of thePortableInterceptor module specified in the OMG Portable Interceptor specification,<http://cgi.omg.org/cgi-bin/doc?ptc/2000-08-06>, section 21.7.2. |
| [**org.omg.PortableServer**](http://docs.oracle.com/javase/6/docs/api/org/omg/PortableServer/package-summary.html) | Provides classes and interfaces for making the server side of your applications portable across multivendor ORBs. |
| [**org.omg.PortableServer.CurrentPackage**](http://docs.oracle.com/javase/6/docs/api/org/omg/PortableServer/CurrentPackage/package-summary.html) | Provides method implementations with access to the identity of the object on which the method was invoked. |
| [**org.omg.PortableServer.POAManagerPackage**](http://docs.oracle.com/javase/6/docs/api/org/omg/PortableServer/POAManagerPackage/package-summary.html) | Encapsulates the processing state of the POAs it is associated with. |
| [**org.omg.PortableServer.POAPackage**](http://docs.oracle.com/javase/6/docs/api/org/omg/PortableServer/POAPackage/package-summary.html) | Allows programmers to construct object implementations that are portable between different ORB products. |
| [**org.omg.PortableServer.portable**](http://docs.oracle.com/javase/6/docs/api/org/omg/PortableServer/portable/package-summary.html) | Provides classes and interfaces for making the server side of your applications portable across multivendor ORBs. |
| [**org.omg.PortableServer.ServantLocatorPackage**](http://docs.oracle.com/javase/6/docs/api/org/omg/PortableServer/ServantLocatorPackage/package-summary.html) | Provides classes and interfaces for locating the servant. |
| [**org.omg.SendingContext**](http://docs.oracle.com/javase/6/docs/api/org/omg/SendingContext/package-summary.html) | Provides support for the marshalling of value types. |
| [**org.omg.stub.java.rmi**](http://docs.oracle.com/javase/6/docs/api/org/omg/stub/java/rmi/package-summary.html) | Contains RMI-IIOP Stubs for the Remote types that occur in the java.rmi package. |
| [**org.w3c.dom**](http://docs.oracle.com/javase/6/docs/api/org/w3c/dom/package-summary.html) | Provides the interfaces for the Document Object Model (DOM) which is a component API of the[Java API for XML Processing](http://java.sun.com/xml). |
| [**org.w3c.dom.bootstrap**](http://docs.oracle.com/javase/6/docs/api/org/w3c/dom/bootstrap/package-summary.html) |  |
| [**org.w3c.dom.events**](http://docs.oracle.com/javase/6/docs/api/org/w3c/dom/events/package-summary.html) |  |
| [**org.w3c.dom.ls**](http://docs.oracle.com/javase/6/docs/api/org/w3c/dom/ls/package-summary.html) |  |
| [**org.xml.sax**](http://docs.oracle.com/javase/6/docs/api/org/xml/sax/package-summary.html) | This package provides the core SAX APIs. |
| [**org.xml.sax.ext**](http://docs.oracle.com/javase/6/docs/api/org/xml/sax/ext/package-summary.html) | This package contains interfaces to SAX2 facilities that conformant SAX drivers won't necessarily support. |
| [**org.xml.sax.helpers**](http://docs.oracle.com/javase/6/docs/api/org/xml/sax/helpers/package-summary.html) | This package contains "helper" classes, including support for bootstrapping SAX-based applications. |

Chapter 6:

**Declarations and Access Controls**

**Topic1:** **Class Access Modifiers Class Non Access Modifiers Member Access Modifiers Member Non Access Modifiers Static members and methods**    :

# Access Modifiers in java

1. [private access modifier](http://www.javatpoint.com/access-modifiers#accessprivate)
2. [Role of private constructor](http://www.javatpoint.com/access-modifiers#accessprivatecons)
3. [default access modifier](http://www.javatpoint.com/access-modifiers#accessdefault)
4. [protected access modifier](http://www.javatpoint.com/access-modifiers#accessprotected)
5. [public access modifier](http://www.javatpoint.com/access-modifiers#accesspublic)
6. [Applying access modifier with method overriding](http://www.javatpoint.com/access-modifiers#accessoverriding)

There are two types of modifiers in java: **access modifiers** and**non-access modifiers**.

The access modifiers in java specifies accessibility (scope) of a data member, method, constructor or class.

There are 4 types of java access modifiers:

1. private
2. default
3. protected
4. public

There are many non-access modifiers such as static, abstract, synchronized, native, volatile, transient etc. Here, we will learn access modifiers.

### 1) private access modifier

|  |
| --- |
| The private access modifier is accessible only within class. |

### Simple example of private access modifier

|  |
| --- |
| In this example, we have created two classes A and Simple. A class contains private data member and private method. We are accessing these private members from outside the class, so there is compile time error. |

1. **class** A{
2. **private** **int** data=40;
3. **private** **void** msg(){System.out.println("Hello java");}
4. }
6. **public** **class** Simple{
7. **public** **static** **void** main(String args[]){
8. A obj=**new** A();
9. System.out.println(obj.data);//Compile Time Error
10. obj.msg();//Compile Time Error
11. }
12. }

### Role of Private Constructor

|  |
| --- |
| If you make any class constructor private, you cannot create the instance of that class from outside the class. For example: |

1. **class** A{
2. **private** A(){}//private constructor
3. **void** msg(){System.out.println("Hello java");}
4. }
5. **public** **class** Simple{
6. **public** **static** **void** main(String args[]){
7. A obj=**new** A();//Compile Time Error
8. }
9. }

#### Note: A class cannot be private or protected except nested class.

### 2) default access modifier

|  |
| --- |
| If you don't use any modifier, it is treated as **default** bydefault. The default modifier is accessible only within package. |

### Example of default access modifier

|  |
| --- |
| In this example, we have created two packages pack and mypack. We are accessing the A class from outside its package, since A class is not public, so it cannot be accessed from outside the package. |

1. //save by A.java
2. **package** pack;
3. **class** A{
4. **void** msg(){System.out.println("Hello");}
5. }
6. //save by B.java
7. **package** mypack;
8. **import** pack.\*;
9. **class** B{
10. **public** **static** **void** main(String args[]){
11. A obj = **new** A();//Compile Time Error
12. obj.msg();//Compile Time Error
13. }
14. }

In the above example, the scope of class A and its method msg() is default so it cannot be accessed from outside the package.

### 3) protected access modifier

The **protected access modifier** is accessible within package and outside the package but through inheritance only.

The protected access modifier can be applied on the data member, method and constructor. It can't be applied on the class.

### Example of protected access modifier

In this example, we have created the two packages pack and mypack. The A class of pack package is public, so can be accessed from outside the package. But msg method of this package is declared as protected, so it can be accessed from outside the class only through inheritance.

1. //save by A.java
2. **package** pack;
3. **public** **class** A{
4. **protected** **void** msg(){System.out.println("Hello");}
5. }
6. //save by B.java
7. **package** mypack;
8. **import** pack.\*;
10. **class** B **extends** A{
11. **public** **static** **void** main(String args[]){
12. B obj = **new** B();
13. obj.msg();
14. }
15. }

Output:Hello

### 4) public access modifier

|  |
| --- |
| The **public access modifier** is accessible everywhere. It has the widest scope among all other modifiers. |

### Example of public access modifier

1. //save by A.java
3. **package** pack;
4. **public** **class** A{
5. **public** **void** msg(){System.out.println("Hello");}
6. }
7. //save by B.java
9. **package** mypack;
10. **import** pack.\*;
12. **class** B{
13. **public** **static** **void** main(String args[]){
14. A obj = **new** A();
15. obj.msg();
16. }
17. }

Output:Hello

### Understanding all java access modifiers

Let's understand the access modifiers by a simple table.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Access Modifier** | **within class** | **within package** | **outside package by subclass only** | **outside package** |
| **Private** | Y | N | N | N |
| **Default** | Y | Y | N | N |
| **Protected** | Y | Y | Y | N |
| **Public** | Y | Y | Y | Y |

### Java access modifiers with method overriding

If you are overriding any method, overridden method (i.e. declared in subclass) must not be more restrictive.

1. **class** A{
2. **protected** **void** msg(){System.out.println("Hello java");}
3. }
5. **public** **class** Simple **extends** A{
6. **void** msg(){System.out.println("Hello java");}//C.T.Error
7. **public** **static** **void** main(String args[]){
8. Simple obj=**new** Simple();
9. obj.msg();
10. }
11. }

|  |
| --- |
| The default modifier is more restrictive than protected. That is why there is compile time error. |

# Java static keyword

1. [Static variable](http://www.javatpoint.com/static-keyword-in-java#staticv)
2. [Program of counter without static variable](http://www.javatpoint.com/static-keyword-in-java#staticvcounter1)
3. [Program of counter with static variable](http://www.javatpoint.com/static-keyword-in-java#staticvcounter2)
4. [Static method](http://www.javatpoint.com/static-keyword-in-java#staticm)
5. [Restrictions for static method](http://www.javatpoint.com/static-keyword-in-java#staticmr)
6. [Why main method is static ?](http://www.javatpoint.com/static-keyword-in-java#staticwhymain)
7. [Static block](http://www.javatpoint.com/static-keyword-in-java#staticblock)
8. [Can we execute a program without main method ?](http://www.javatpoint.com/static-keyword-in-java#staticwithoutmain)

The **static keyword** in java is used for memory management mainly. We can apply java static keyword with variables, methods, blocks and nested class. The static keyword belongs to the class than instance of the class.

The static can be:

1. variable (also known as class variable)
2. method (also known as class method)
3. block
4. nested class

## 1) Java static variable

If you declare any variable as static, it is known static variable.

* The static variable can be used to refer the common property of all objects (that is not unique for each object) e.g. company name of employees,college name of students etc.
* The static variable gets memory only once in class area at the time of class loading.

### Advantage of static variable

It makes your program **memory efficient** (i.e it saves memory).

#### Understanding problem without static variable

1. **class** Student{
2. **int** rollno;
3. String name;
4. String college="ITS";
5. }

Suppose there are 500 students in my college, now all instance data members will get memory each time when object is created.All student have its unique rollno and name so instance data member is good.Here, college refers to the common property of all objects.If we make it static,this field will get memory only once.

#### Java static property is shared to all objects.

### Example of static variable

1. //Program of static variable
3. **class** Student8{
4. **int** rollno;
5. String name;
6. **static** String college ="ITS";
8. Student8(**int** r,String n){
9. rollno = r;
10. name = n;
11. }
12. **void** display (){System.out.println(rollno+" "+name+" "+college);}
14. **public** **static** **void** main(String args[]){
15. Student8 s1 = **new** Student8(111,"Karan");
16. Student8 s2 = **new** Student8(222,"Aryan");
18. s1.display();
19. s2.display();
20. }
21. }

[**Test it Now**](http://www.javatpoint.com/opr/test.jsp?filename=Student8)

Output:111 Karan ITS

222 Aryan ITS

### Program of counter without static variable

In this example, we have created an instance variable named count which is incremented in the constructor. Since instance variable gets the memory at the time of object creation, each object will have the copy of the instance variable, if it is incremented, it won't reflect to other objects. So each objects will have the value 1 in the count variable.

1. **class** Counter{
2. **int** count=0;//will get memory when instance is created
4. Counter(){
5. count++;
6. System.out.println(count);
7. }
9. **public** **static** **void** main(String args[]){
11. Counter c1=**new** Counter();
12. Counter c2=**new** Counter();
13. Counter c3=**new** Counter();
15. }
16. }

[**Test it Now**](http://www.javatpoint.com/opr/test.jsp?filename=Counter)

Output:1

1

1

### Program of counter by static variable

|  |
| --- |
| As we have mentioned above, static variable will get the memory only once, if any object changes the value of the static variable, it will retain its value. |

1. **class** Counter2{
2. **static** **int** count=0;//will get memory only once and retain its value
4. Counter2(){
5. count++;
6. System.out.println(count);
7. }
9. **public** **static** **void** main(String args[]){
11. Counter2 c1=**new** Counter2();
12. Counter2 c2=**new** Counter2();
13. Counter2 c3=**new** Counter2();
15. }
16. }

[**Test it Now**](http://www.javatpoint.com/opr/test.jsp?filename=Counter2)

Output:1

2

3

## 2) Java static method

If you apply static keyword with any method, it is known as static method.

* A static method belongs to the class rather than object of a class.
* A static method can be invoked without the need for creating an instance of a class.
* static method can access static data member and can change the value of it.

### Example of static method

1. //Program of changing the common property of all objects(static field).
3. **class** Student9{
4. **int** rollno;
5. String name;
6. **static** String college = "ITS";
8. **static** **void** change(){
9. college = "BBDIT";
10. }
12. Student9(**int** r, String n){
13. rollno = r;
14. name = n;
15. }
17. **void** display (){System.out.println(rollno+" "+name+" "+college);}
19. **public** **static** **void** main(String args[]){
20. Student9.change();
22. Student9 s1 = **new** Student9 (111,"Karan");
23. Student9 s2 = **new** Student9 (222,"Aryan");
24. Student9 s3 = **new** Student9 (333,"Sonoo");
26. s1.display();
27. s2.display();
28. s3.display();
29. }
30. }

[**Test it Now**](http://www.javatpoint.com/opr/test.jsp?filename=Student9)

Output:111 Karan BBDIT

222 Aryan BBDIT

333 Sonoo BBDIT

### Another example of static method that performs normal calculation

1. //Program to get cube of a given number by static method
3. **class** Calculate{
4. **static** **int** cube(**int** x){
5. **return** x\*x\*x;
6. }
8. **public** **static** **void** main(String args[]){
9. **int** result=Calculate.cube(5);
10. System.out.println(result);
11. }
12. }

[**Test it Now**](http://www.javatpoint.com/opr/test.jsp?filename=Calculate)

Output:125

### Restrictions for static method

|  |
| --- |
| There are two main restrictions for the static method. They are: |

|  |
| --- |
| 1. The static method can not use non static data member or call non-static method directly. 2. this and super cannot be used in static context. |

1. **class** A{
2. **int** a=40;//non static
4. **public** **static** **void** main(String args[]){
5. System.out.println(a);
6. }
7. }

[**Test it Now**](http://www.javatpoint.com/opr/test.jsp?filename=A)

Output:Compile Time Error

### Q) why java main method is static?

|  |
| --- |
| Ans) because object is not required to call static method if it were non-static method, jvm create object first then call main() method that will lead the problem of extra memory allocation. |

## 3) Java static block

* Is used to initialize the static data member.
* It is executed before main method at the time of classloading.

### Example of static block

1. **class** A2{
2. **static**{System.out.println("static block is invoked");}
3. **public** **static** **void** main(String args[]){
4. System.out.println("Hello main");
5. }
6. }

[**Test it Now**](http://www.javatpoint.com/opr/test.jsp?filename=A2)

Output:static block is invoked

Hello main

### Q) Can we execute a program without main() method?

Ans) Yes, one of the way is static block but in previous version of JDK not in JDK 1.7.

1. **class** A3{
2. **static**{
3. System.out.println("static block is invoked");
4. System.exit(0);
5. }
6. }

[**Test it Now**](http://www.javatpoint.com/opr/test.jsp?filename=A3)

Output:static block is invoked (if not JDK7)

In JDK7 and above, output will be:

Output:Error: Main method not found in class A3, please define the main method as:

public static void main(String[] args)

Java provides a number of non-access modifiers to achieve many other functionality.

* The *static* modifier for creating class methods and variables
* The *final* modifier for finalizing the implementations of classes, methods, and variables.
* The *abstract* modifier for creating abstract classes and methods.
* The *synchronized* and *volatile* modifiers, which are used for threads.

## The static Modifier:

## Static Variables:

The *static* key word is used to create variables that will exist independently of any instances created for the class. Only one copy of the static variable exists regardless of the number of instances of the class.

Static variables are also known as class variables. Local variables cannot be declared static.

## Static Methods:

The static key word is used to create methods that will exist independently of any instances created for the class.

Static methods do not use any instance variables of any object of the class they are defined in. Static methods take all the data from parameters and compute something from those parameters, with no reference to variables.

Class variables and methods can be accessed using the class name followed by a dot and the name of the variable or method.

## Example:

The static modifier is used to create class methods and variables, as in the following example:

public class InstanceCounter {

private static int numInstances = 0;

protected static int getCount() {

return numInstances;

}

private static void addInstance() {

numInstances++;

}

InstanceCounter() {

InstanceCounter.addInstance();

}

public static void main(String[] arguments) {

System.out.println("Starting with " +

InstanceCounter.getCount() + " instances");

for (int i = 0; i < 500; ++i){

new InstanceCounter();

}

System.out.println("Created " +

InstanceCounter.getCount() + " instances");

}

}

This would produce the following result:

Started with 0 instances

Created 500 instances

## The final Modifier:

## final Variables:

A final variable can be explicitly initialized only once. A reference variable declared final can never be reassigned to refer to an different object.

However the data within the object can be changed. So the state of the object can be changed but not the reference.

With variables, the *final* modifier often is used with *static* to make the constant a class variable.

## Example:

public class Test{

final int value = 10;

// The following are examples of declaring constants:

public static final int BOXWIDTH = 6;

static final String TITLE = "Manager";

public void changeValue(){

value = 12; //will give an error

}

}

## final Methods:

A final method cannot be overridden by any subclasses. As mentioned previously the final modifier prevents a method from being modified in a subclass.

The main intention of making a method final would be that the content of the method should not be changed by any outsider.

## Example:

You declare methods using the *final* modifier in the class declaration, as in the following example:

public class Test{

public final void changeName(){

// body of method

}

}

## final Classes:

The main purpose of using a class being declared as *final* is to prevent the class from being subclassed. If a class is marked as final then no class can inherit any feature from the final class.

## Example:

public final class Test {

// body of class

}

## The abstract Modifier:

## abstract Class:

An abstract class can never be instantiated. If a class is declared as abstract then the sole purpose is for the class to be extended.

A class cannot be both abstract and final. (since a final class cannot be extended). If a class contains abstract methods then the class should be declared abstract. Otherwise a compile error will be thrown.

An abstract class may contain both abstract methods as well normal methods.

## Example:

abstract class Caravan{

private double price;

private String model;

private String year;

public abstract void goFast(); //an abstract method

public abstract void changeColor();

}

## abstract Methods:

An abstract method is a method declared with out any implementation. The methods body(implementation) is provided by the subclass. Abstract methods can never be final or strict.

Any class that extends an abstract class must implement all the abstract methods of the super class unless the subclass is also an abstract class.

If a class contains one or more abstract methods then the class must be declared abstract. An abstract class does not need to contain abstract methods.

The abstract method ends with a semicolon. Example: public abstract sample();

## Example:

public abstract class SuperClass{

abstract void m(); //abstract method

}

class SubClass extends SuperClass{

// implements the abstract method

void m(){

.........

}

}

## The synchronized Modifier:

The synchronized key word used to indicate that a method can be accessed by only one thread at a time. The synchronized modifier can be applied with any of the four access level modifiers.

## Example:

public synchronized void showDetails(){

.......

}

## The transient Modifier:

An instance variable is marked transient to indicate the JVM to skip the particular variable when serializing the object containing it.

This modifier is included in the statement that creates the variable, preceding the class or data type of the variable.

## Example:

public transient int limit = 55; // will not persist

public int b; // will persist

## The volatile Modifier:

The volatile is used to let the JVM know that a thread accessing the variable must always merge its own private copy of the variable with the master copy in the memory.

Accessing a volatile variable synchronizes all the cached copied of the variables in the main memory. Volatile can only be applied to instance variables, which are of type object or private. A volatile object reference can be null.

## Example:

public class MyRunnable implements Runnable

{

private volatile boolean active;

public void run()

{

active = true;

while (active) // line 1

{

// some code here

}

}

public void stop()

{

active = false; // line 2

}

}

Usually, run() is called in one thread (the one you start using the Runnable), and stop() is called from another thread. If in line 1 the cached value of active is used, the loop may not stop when you set active to false in line 2. That's when you want to use *volatile*.

Chapter 7: **Constructors and Data Initialization**

|  |  |
| --- | --- |
| |  | | --- | | **0. Default Constructor**    :    [**Link**](http://www.javatpoint.com/constructor) | |
| |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  | | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | | Constructor in Java  1. [Types of constructors](http://www.javatpoint.com/constructor#constypes)    1. [Default Constructor](http://www.javatpoint.com/constructor#consdef)    2. [Parameterized Constructor](http://www.javatpoint.com/constructor#conspara) 2. [Constructor Overloading](http://www.javatpoint.com/constructor#consoverloading) 3. [Does constructor return any value](http://www.javatpoint.com/constructor#consdoesreturn) 4. [Copying the values of one object into another](http://www.javatpoint.com/constructor#conscopy) 5. [Does constructor perform other task instead initialization](http://www.javatpoint.com/constructor#consothertask)   **Constructor in java** is a special type of method that is used to initialize the object.  Java constructor is invoked at the time of object creation. It constructs the values i.e. provides data  for the object that is why it is known as constructor. Rules for creating java constructor There are basically two rules defined for the constructor.   1. Constructor name must be same as its class name 2. Constructor must have no explicit return type  Types of java constructors There are two types of constructors:   1. Default constructor (no-arg constructor) 2. Parameterized constructor  Java Default Constructor  |  | | --- | | A constructor that have no parameter is known as default constructor. |  Syntax of default constructor:  1. <class\_name>(){}  Example of default constructor  |  | | --- | | In this example, we are creating the no-arg constructor in the Bike class. It will be invoked at the  time of object creation. |  1. **class** Bike1{ 2. Bike1(){System.out.println("Bike is created");} 3. **public** **static** **void** main(String args[]){ 4. Bike1 b=**new** Bike1(); 5. } 6. }   [**Test it Now**](http://www.javatpoint.com/opr/test.jsp?filename=Bike1)  Output:  Bike is created Rule: If there is no constructor in a class, compiler automatically creates a default constructor. default constructor Q) What is the purpose of default constructor? Default constructor provides the default values to the object like 0, null etc. depending on the type. Example of default constructor that displays the default values  1. **class** Student3{ 2. **int** id; 3. String name; 5. **void** display(){System.out.println(id+" "+name);} 7. **public** **static** **void** main(String args[]){ 8. Student3 s1=**new** Student3(); 9. Student3 s2=**new** Student3(); 10. s1.display(); 11. s2.display(); 12. } 13. }   [**Test it Now**](http://www.javatpoint.com/opr/test.jsp?filename=Student3)  Output:  0 null  0 null  **Explanation:**In the above class,you are not creating any constructor so compiler provides you a default constructor.Here 0 and null values are provided by default constructor. Java parameterized constructor  |  | | --- | | A constructor that have parameters is known as parameterized constructor. |  Why use parameterized constructor?  |  | | --- | | Parameterized constructor is used to provide different values to the distinct objects. |  Example of parameterized constructor  |  | | --- | | In this example, we have created the constructor of Student class that have two parameters. We  can have any number of parameters in the constructor. |  1. **class** Student4{ 2. **int** id; 3. String name; 5. Student4(**int** i,String n){ 6. id = i; 7. name = n; 8. } 9. **void** display(){System.out.println(id+" "+name);} 11. **public** **static** **void** main(String args[]){ 12. Student4 s1 = **new** Student4(111,"Karan"); 13. Student4 s2 = **new** Student4(222,"Aryan"); 14. s1.display(); 15. s2.display(); 16. } 17. }   [**Test it Now**](http://www.javatpoint.com/opr/test.jsp?filename=Student4)  Output:  111 Karan  222 Aryan Constructor Overloading in Java  |  | | --- | | Constructor overloading is a technique in Java in which a class can have any number of  constructors that differ in parameter lists. The compiler differentiates these constructors by  considering the number of parameters in the list and their type. |  Example of Constructor Overloading  1. **class** Student5{ 2. **int** id; 3. String name; 4. **int** age; 5. Student5(**int** i,String n){ 6. id = i; 7. name = n; 8. } 9. Student5(**int** i,String n,**int** a){ 10. id = i; 11. name = n; 12. age=a; 13. } 14. **void** display(){System.out.println(id+" "+name+" "+age);} 16. **public** **static** **void** main(String args[]){ 17. Student5 s1 = **new** Student5(111,"Karan"); 18. Student5 s2 = **new** Student5(222,"Aryan",25); 19. s1.display(); 20. s2.display(); 21. } 22. }   [**Test it Now**](http://www.javatpoint.com/opr/test.jsp?filename=Student5)  Output:  111 Karan 0  222 Aryan 25 Difference between constructor and method in java There are many differences between constructors and methods. They are given below.   |  |  | | --- | --- | | **Java Constructor** | **Java Method** | | Constructor is used to initialize the state of an object. | Method is used to  expose behaviour of an  object. | | Constructor must not have return type. | Method must have return  type. | | Constructor is invoked implicitly. | Method is invoked  explicitly. | | The java compiler provides a default constructor if you don't have any constructor. | Method is not provided by  compiler in any case. | | Constructor name must be same as the class name. | Method name may or may  not be same as class name. |  Java Copy Constructor There is no copy constructor in java. But, we can copy the values of one object to another like  copy constructor in C++.  There are many ways to copy the values of one object into another in java. They are:   * By constructor * By assigning the values of one object into another * By clone() method of Object class   In this example, we are going to copy the values of one object into another using java constructor. |  1. **class** Student6{ 2. **int** id; 3. String name; 4. Student6(**int** i,String n){ 5. id = i; 6. name = n; 7. } 9. Student6(Student6 s){ 10. id = s.id; 11. name =s.name; 12. } 13. **void** display(){System.out.println(id+" "+name);} 15. **public** **static** **void** main(String args[]){ 16. Student6 s1 = **new** Student6(111,"Karan"); 17. Student6 s2 = **new** Student6(s1); 18. s1.display(); 19. s2.display(); 20. } 21. }   [**Test it Now**](http://www.javatpoint.com/opr/test.jsp?filename=Student6)  Output:  111 Karan  111 Karan Copying values without constructor We can copy the values of one object into another by assigning the objects values to another object.  In this case, there is no need to create the constructor.   1. **class** Student7{ 2. **int** id; 3. String name; 4. Student7(**int** i,String n){ 5. id = i; 6. name = n; 7. } 8. Student7(){} 9. **void** display(){System.out.println(id+" "+name);} 11. **public** **static** **void** main(String args[]){ 12. Student7 s1 = **new** Student7(111,"Karan"); 13. Student7 s2 = **new** Student7(); 14. s2.id=s1.id; 15. s2.name=s1.name; 16. s1.display(); 17. s2.display(); 18. } 19. }   [**Test it Now**](http://www.javatpoint.com/opr/test.jsp?filename=Student7)  Output:  111 Karan  111 Karan Q) Does constructor return any value? **Ans:** yes, that is current class instance (You cannot use return type yet it returns a value). Can constructor perform other tasks instead of initialization? Yes, like object creation, starting a thread, calling method etc. You can perform any operation  in the constructor as you perform in the method. |
| |  | | --- | | **0. super: Constructor Call**    :    [**Link**](http://www.javatpoint.com/super-keyword) |  super keyword in java The **super** keyword in java is a reference variable that is used to refer immediate parent class object.  Whenever you create the instance of subclass, an instance of parent class is created implicitly i.e.  referred by super reference variable. Usage of java super Keyword  1. super is used to refer immediate parent class instance variable. 2. super() is used to invoke immediate parent class constructor. 3. super is used to invoke immediate parent class method.  1) super is used to refer immediate parent class instance variable. ***Problem without super keyword***   1. **class** Vehicle{ 2. **int** speed=50; 3. } 4. **class** Bike3 **extends** Vehicle{ 5. **int** speed=100; 6. **void** display(){ 7. System.out.println(speed);//will print speed of Bike 8. } 9. **public** **static** **void** main(String args[]){ 10. Bike3 b=**new** Bike3(); 11. b.display(); 12. } 13. }   [**Test it Now**](http://www.javatpoint.com/opr/test.jsp?filename=Bike3)  Output:100   |  | | --- | | In the above example Vehicle and Bike both class have a common property speed. Instance variable of current class is refered by instance bydefault, but I have to refer parent class instance variable that is why we use super keyword to distinguish between parent class instance variable and current class instance variable. |   ***Solution by super keyword***   1. //example of super keyword 3. **class** Vehicle{ 4. **int** speed=50; 5. } 7. **class** Bike4 **extends** Vehicle{ 8. **int** speed=100; 10. **void** display(){ 11. System.out.println(**super**.speed);//will print speed of Vehicle now 12. } 13. **public** **static** **void** main(String args[]){ 14. Bike4 b=**new** Bike4(); 15. b.display(); 17. } 18. }   [**Test it Now**](http://www.javatpoint.com/opr/test.jsp?filename=Bike4)  Output:50 2) super is used to invoke parent class constructor.  |  | | --- | | The super keyword can also be used to invoke the parent class constructor as given below: |  1. **class** Vehicle{ 2. Vehicle(){System.out.println("Vehicle is created");} 3. } 5. **class** Bike5 **extends** Vehicle{ 6. Bike5(){ 7. **super**();//will invoke parent class constructor 8. System.out.println("Bike is created"); 9. } 10. **public** **static** **void** main(String args[]){ 11. Bike5 b=**new** Bike5(); 13. } 14. }   [**Test it Now**](http://www.javatpoint.com/opr/test.jsp?filename=Bike5)  Output:Vehicle is created  Bike is created Note: super() is added in each class constructor automatically by compiler. java super  As we know well that default constructor is provided by compiler automatically but it also adds super ()  for the first statement. If you are creating your own constructor and you don't have  either this () or super() as the first statement, compiler will provide super()  as the first statement of the constructor. Another example of super keyword where super() is provided by thecompiler implicitly.  1. **class** Vehicle{ 2. Vehicle(){System.out.println("Vehicle is created");} 3. } 5. **class** Bike6 **extends** Vehicle{ 6. **int** speed; 7. Bike6(**int** speed){ 8. **this**.speed=speed; 9. System.out.println(speed); 10. } 11. **public** **static** **void** main(String args[]){ 12. Bike6 b=**new** Bike6(10); 13. } 14. }   [**Test it Now**](http://www.javatpoint.com/opr/test.jsp?filename=Bike6)  Output:Vehicle is created  10 3) super can be used to invoke parent class method  |  | | --- | | The super keyword can also be used to invoke parent class method. It should be used in case subclass contains the same method as parent class as in the example given below: |  1. **class** Person{ 2. **void** message(){System.out.println("welcome");} 3. } 5. **class** Student16 **extends** Person{ 6. **void** message(){System.out.println("welcome to java");} 8. **void** display(){ 9. message();//will invoke current class message() method 10. **super**.message();//will invoke parent class message() method 11. } 13. **public** **static** **void** main(String args[]){ 14. Student16 s=**new** Student16(); 15. s.display(); 16. } 17. }   [**Test it Now**](http://www.javatpoint.com/opr/test.jsp?filename=Student16)  Output:welcome to java  welcome   |  | | --- | | In the above example Student and Person both classes have message() method if we call  message() method from Student class, it will call the message() method of Student  class not of Person class because priority is given to local. |  |  | | --- | | In case there is no method in subclass as parent, there is no need to use super.  In the example given below message() method is invoked from Student class  but Student class does not have message() method, so you can directly call message() method. |  Program in case super is not required  1. **class** Person{ 2. **void** message(){System.out.println("welcome");} 3. } 5. **class** Student17 **extends** Person{ 7. **void** display(){ 8. message();//will invoke parent class message() method 9. } 11. **public** **static** **void** main(String args[]){ 12. Student17 s=**new** Student17(); 13. s.display(); 14. } 15. }   [**Test it Now**](http://www.javatpoint.com/opr/test.jsp?filename=Student17)  Output:welcome |
| |  | | --- | | **0. Invoke constructor using ‘this’**    :    [**Link**](http://docs.oracle.com/javase/tutorial/java/javaOO/thiskey.html) |  Using the this Keyword Within an instance method or a constructor, this is a reference to the *current object* — the object whose method or constructor is being called. You can refer to any member of the current object from within an instance method or a constructor by using this. Using this with a Field The most common reason for using the this keyword is because a field is shadowed by a method or constructor parameter.  For example, the Point class was written like this  public class Point {  public int x = 0;  public int y = 0;    **//constructor**  **public Point(int a, int b) {**  **x = a;**  **y = b;**  **}**  }  but it could have been written like this:  public class Point {  public int x = 0;  public int y = 0;    **//constructor**  **public Point(int x, int y) {**  **this.x = x;**  **this.y = y;**  **}**  }  Each argument to the constructor shadows one of the object's fields — inside the constructor **x** is a local copy of the constructor's first argument. To refer to the Point field **x**, the constructor must use **this.x**. Using this with a Constructor From within a constructor, you can also use the this keyword to call another constructor in the same class. Doing so is called an *explicit constructor invocation*. Here's anotherRectangle class, with a different implementation from the one in the [Objects](http://docs.oracle.com/javase/tutorial/java/javaOO/objects.html) section.  public class Rectangle {  private int x, y;  private int width, height;    public Rectangle() {  **this(0, 0, 1, 1);**  }  public Rectangle(int width, int height) {  **this(0, 0, width, height);**  }  public Rectangle(int x, int y, int width, int height) {  this.x = x;  this.y = y;  this.width = width;  this.height = height;  }  ...  }  This class contains a set of constructors. Each constructor initializes some or all of the rectangle's member variables. The constructors provide a default value for any member variable whose initial value is not provided by an argument. For example, the no-argument constructor creates a 1x1 Rectangle at coordinates 0,0. The two-argument constructor calls the four-argument constructor, passing in the width and height but always using the 0,0 coordinates. As before, the compiler determines which constructor to call, based on the number and the type of arguments.  If present, the invocation of another constructor must be the first line in the constructor. |
| |  | | --- | | **0. Constructor Chaining**    :    [**Link**](http://javarevisited.blogspot.in/2012/12/constructor-chaining-in-java-calling-another-constructor.html) |   **Constructor Chaining in Java**  In Java you can call one constructor from another and it’s known as constructor chaining in Java.  Don’t confuse between [constructor overloading](http://javarevisited.blogspot.sg/2012/01/what-is-constructor-overloading-in-java.html) and constructor chaining, former is just a way to declare more than one [constructor in Java](http://javarevisited.blogspot.sg/2012/12/what-is-constructor-in-java-example-chainning-overloading.html). this and super keyword is used to call one constructor from other in Java. this() can be used to call other  constructor of same class while super() can be used to call constructor from super class in Java. Just keep in mind that this() in realty calls **no argument constructor of same class**, while this(2) calls another constructor of same class which accepts one integer parameter. Similarly super() can be used to call no argument constructor of super class and super with parameter can be used to call other [overloaded constructor](http://javarevisited.blogspot.sg/2012/01/what-is-constructor-overloading-in-java.html) of parent class. Calling one constructor from other is called*constructor chaining in Java*, which we seen while discussing constructor overloading in Java. Constructor chaining is also used to implement [telescoping pattern](http://javarevisited.blogspot.ro/2012/06/builder-design-pattern-in-java-example.html) where an object can be created with combination of multiple property. In our last tutorial we have seen some important properties of Java constructor as well as answered question [What is Constructor in Java](http://javarevisited.blogspot.sg/2012/12/what-is-constructor-in-java-example-chainning-overloading.html) and in this Java tutorial we will see example of*how to call one constructor from other* for same class and super class. How to call overloaded constructor in Java Constructor overloading allows you to declare multiple constructor while constructor chaining allows you to  call those constructor. This is main difference between constructor overloading and constructor chaining. Since constructor  can only be called from another constructor in Java, constructor chaining is an important concept to understand to know  How and [when class is initialized in Java.](http://javarevisited.blogspot.sg/2012/07/when-class-loading-initialization-java-example.html)  Here is complete code example of constructor chaining which shows How to call  overloaded constructor of same class and parent class in Java.  /\*\*  \* Simple **Java program to demonstrate how to call one constructor from other**.  \* Calling one constructor from other is called constructor chaining.  \* this() is used to call constructor of same class while super() is used to  \* call constructor of Super class in Java.   \*  \* @author Javin  Paul  \*/ **public** **class** ConstructorChainingExample {           **public** **static** **void** main(**String** args[]) {              *//this will first call one argument constructor of Child Class which*         *//in turn call corresponding constructor of super class using super(String)*         **System**.out.println("Constructor chaining Example in Java");         Child child = **new** Child("Jeremy");                *//this constructor will call no argument constructor of Child,*  *//which then call one argument constructor of*         *//same class, which finally call corresponding one argument constructor*  *// of super class Parent.*         **System**.out.println("---------------------------------");         Child emptyChild = **new** Child();     }     }  **class** Parent{     **private** **String** name;     */\*      \* Calling constructor of same class with one String argument      \*/*     **protected** Parent(){         **this**("");         **System**.out.println("No argument constructor of Parent called ");     }        **protected** Parent(**String** name){         **this**.name = name;         **System**.out.println("One String argument constructor of Parent called ");     } }  **class** Child **extends** Parent{     **private** **String** name;        */\*      \* Calling constructor same class with one argument      \*/*     **protected** Child(){         **this**("");         **System**.out.println("No argument constructor of Child called ");     }        */\*      \* Calling constructor of super class with one argument      \* call to super() must be first line in constructor      \*/*     **protected** Child(**String** name){         **super**(name);         **System**.out.println("One argument constructor of Super class called from sub class ");     } }  **Constructor** chaining Example in Java One **String** argument constructor of Parent called One argument constructor of Super **class** called from sub **class** --------------------------------- One **String** argument constructor of Parent called One argument constructor of Super **class** called from sub **class** No argument constructor of Child called  That’s all on **What is constructor chaining in Java**. We have seen How to call overloaded constructor from same class using this() and constructor from super class using super(). Key thing to remember is that call to another constructor must be first line in calling constructor.  Read more: <http://javarevisited.blogspot.com/2012/12/constructor-chaining-in-java-calling-another-constructor.html#ixzz3gDsPs8eW> |
| |  | | --- | | **0. Constructor Initialization**    :    [**Link**](http://math.hws.edu/javanotes/c5/s2.html) |  Constructors and Object Initialization OBJECT TYPES IN JAVA are very different from the primitive types. Simply declaring a variable whose  type is given as a class does not automatically create an object of that class. Objects must be explicitly  constructed. For the computer, the process of constructing an object means, first, finding some unused  memory in the heap that can be used to hold the object and, second, filling in the object's instance variables.  As a programmer, you don't care where in memory the object is stored, but you will usually want to exercise  some control over what initial values are stored in a new object's instance variables. In many cases,  you will also want to do more complicated initialization or bookkeeping every time an object is created. 5.2.1 Initializing Instance Variables An instance variable can be assigned an initial value in its declaration, just like any other variable.  For example, consider a class named *PairOfDice*. An object of this class will represent a pair of dice. It will contain two instance variables to represent the numbers showing on the dice and an instance method for rolling the dice:  public class PairOfDice {  public int die1 = 3; // Number showing on the first die.  public int die2 = 4; // Number showing on the second die.  public void roll() {  // Roll the dice by setting each of the dice to be  // a random number between 1 and 6.  die1 = (int)(Math.random()\*6) + 1;  die2 = (int)(Math.random()\*6) + 1;  }    } // end class PairOfDice  The instance variables die1 and die2 are initialized to the values 3 and 4 respectively. These initializations are executed whenever a *PairOfDice* object is constructed. It's important to understand when and how this happens. There can be many *PairOfDice* objects. Each time one is created, it gets its own instance variables, and the assignments "die1 = 3" and "die2 = 4" are executed to fill in the values of those variables. To make this clearer, consider a variation of the *PairOfDice* class:  public class PairOfDice {  public int die1 = (int)(Math.random()\*6) + 1;  public int die2 = (int)(Math.random()\*6) + 1;    public void roll() {  die1 = (int)(Math.random()\*6) + 1;  die2 = (int)(Math.random()\*6) + 1;  }    } // end class PairOfDice  Here, every time a new *PaifOfDice* is created, the dice are initialized to random values, as if a new pair of dice were being thrown onto the gaming table. Since the initialization is executed for each new object, a set of random initial values will be computed for each new pair of dice. Different pairs of dice can have different initial values. For initialization of**static** member variables, of course, the situation is quite different. There is only one copy of a static variable, and initialization of that variable is executed just once, when the class is first loaded.  If you don't provide any initial value for an instance variable, a default initial value is provided automatically. Instance variables of numerical type (int, double, etc.) are automatically initialized to zero if you provide no other values; boolean variables are initialized to false; and char variables, to the Unicode character with code number zero. An instance variable can also be a variable of object type. For such variables, the default initial value is null. (In particular, since Strings are objects, the default initial value for *String* variables is null.) 5.2.2  Constructors Objects are created with the operator, new. For example, a program that wants to use a *PairOfDice* object could say:  PairOfDice dice; // Declare a variable of type PairOfDice.  dice = new PairOfDice(); // Construct a new object and store a  // reference to it in the variable.  In this example, "new PairOfDice()" is an expression that allocates memory for the object, initializes the object's instance variables, and then returns a reference to the object. This reference is the value of the expression, and that value is stored by the assignment statement in the variable, dice, so that after the assignment statement is executed, dice refers to the newly created object. Part of this expression, "PairOfDice()", looks like a subroutine call, and that is no accident. It is, in fact, a call to a special type of subroutine called a constructor. This might puzzle you, since there is no such subroutine in the class definition. However, every class has at least one constructor. If the programmer doesn't write a constructor definition in a class, then the system will provide a default constructor for that class. This default constructor does nothing beyond the basics: allocate memory and initialize instance variables. If you want more than that to happen when an object is created, you can include one or more constructors in the class definition.  The definition of a constructor looks much like the definition of any other subroutine, with three exceptions. A constructor does not have any return type (not even void). The name of the constructor must be the same as the name of the class in which it is defined. And the only modifiers that can be used on a constructor definition are the access modifiers public,private, and protected. (In particular, a constructor can't be declared static.)  However, a constructor does have a subroutine body of the usual form, a block of statements. There are no restrictions on what statements can be used. And a constructor can have a list of formal parameters. In fact, the ability to include parameters is one of the main reasons for using constructors. The parameters can provide data to be used in the construction of the object. For example, a constructor for the *PairOfDice* class could provide the values that are initially showing on the dice. Here is what the class would look like in that case:  public class PairOfDice {  public int die1; // Number showing on the first die.  public int die2; // Number showing on the second die.    public PairOfDice(int val1, int val2) {  // Constructor. Creates a pair of dice that  // are initially showing the values val1 and val2.  die1 = val1; // Assign specified values  die2 = val2; // to the instance variables.  }  public void roll() {  // Roll the dice by setting each of the dice to be  // a random number between 1 and 6.  die1 = (int)(Math.random()\*6) + 1;  die2 = (int)(Math.random()\*6) + 1;  }    } // end class PairOfDice  The constructor is declared as "public PairOfDice(int val1, int val2) ...", with no return type and with the same name as the name of the class. This is how the Java compiler recognizes a constructor. The constructor has two parameters, and values for these parameters must be provided when the constructor is called. For example, the expression "new PairOfDice(3,4)" would create a *PairOfDice* object in which the values of the instance variables die1 and die2 are initially 3 and 4. Of course, in a program, the value returned by the constructor should be used in some way, as in  PairOfDice dice; // Declare a variable of type PairOfDice.  dice = new PairOfDice(1,1); // Let dice refer to a new PairOfDice  // object that initially shows 1, 1.  Now that we've added a constructor to the *PairOfDice* class, we can no longer create an object by saying "new PairOfDice()"! The system provides a default constructor for a class**only** if the class definition does not already include a constructor. In this version of *PariOfDice*, there is only one constructor in the class, and it requires two actual parameters. However, this is not a big problem, since we can add a second constructor to the class, one that has no parameters. In fact, you can have as many different constructors as you want, as long as their signatures are different, that is, as long as they have different numbers or types of formal parameters. In the *PairOfDice* class, we might have a constructor with no parameters which produces a pair of dice showing random numbers:  public class PairOfDice {  public int die1; // Number showing on the first die.  public int die2; // Number showing on the second die.    public PairOfDice() {  // Constructor. Rolls the dice, so that they initially  // show some random values.  roll(); // Call the roll() method to roll the dice.  }    public PairOfDice(int val1, int val2) {  // Constructor. Creates a pair of dice that  // are initially showing the values val1 and val2.  die1 = val1; // Assign specified values  die2 = val2; // to the instance variables.  }  public void roll() {  // Roll the dice by setting each of the dice to be  // a random number between 1 and 6.  die1 = (int)(Math.random()\*6) + 1;  die2 = (int)(Math.random()\*6) + 1;  }  } // end class PairOfDice  Now we have the option of constructing a *PairOfDice* object either with "new PairOfDice()" or with "new PairOfDice(x,y)", where x and y are int-valued expressions.  This class, once it is written, can be used in any program that needs to work with one or more pairs of dice. None of those programs will ever have to use the obscure incantation "(int)(Math.random()\*6)+1", because it's done inside the *PairOfDice* class. And the programmer, having once gotten the dice-rolling thing straight will never have to worry about it again. Here, for example, is a main program that uses the *PairOfDice* class to count how many times two pairs of dice are rolled before the two pairs come up showing the same value. This illustrates once again that you can create several instances of the same class:  public class RollTwoPairs {  public static void main(String[] args) {    PairOfDice firstDice; // Refers to the first pair of dice.  firstDice = new PairOfDice();    PairOfDice secondDice; // Refers to the second pair of dice.  secondDice = new PairOfDice();    int countRolls; // Counts how many times the two pairs of  // dice have been rolled.    int total1; // Total showing on first pair of dice.  int total2; // Total showing on second pair of dice.    countRolls = 0;    do { // Roll the two pairs of dice until totals are the same.    firstDice.roll(); // Roll the first pair of dice.  total1 = firstDice.die1 + firstDice.die2; // Get total.  System.out.println("First pair comes up " + total1);    secondDice.roll(); // Roll the second pair of dice.  total2 = secondDice.die1 + secondDice.die2; // Get total.  System.out.println("Second pair comes up " + total2);    countRolls++; // Count this roll.    System.out.println(); // Blank line.    } while (total1 != total2);    System.out.println("It took " + countRolls  + " rolls until the totals were the same.");    } // end main()  } // end class RollTwoPairs  Constructors are subroutines, but they are subroutines of a special type. They are certainly not instance methods, since they don't belong to objects. Since they are responsible for creating objects, they exist before any objects have been created. They are more like static member subroutines, but they are not and cannot be declared to be static. In fact, according to the Java language specification, they are technically not members of the class at all! In particular, constructors are not referred to as "methods."  Unlike other subroutines, a constructor can only be called using the new operator, in an expression that has the form  new **class-name** ( **parameter-list** )  where the **parameter-list** is possibly empty. I call this an expression because it computes and returns a value, namely a reference to the object that is constructed. Most often, you will store the returned reference in a variable, but it is also legal to use a constructor call in other ways, for example as a parameter in a subroutine call or as part of a more complex expression. Of course, if you don't save the reference in a variable, you won't have any way of referring to the object that was just created.  A constructor call is more complicated than an ordinary subroutine or function call. It is helpful to understand the exact steps that the computer goes through to execute a constructor call:   1. First, the computer gets a block of unused memory in the heap, large enough to hold an object of the specified type. 2. It initializes the instance variables of the object. If the declaration of an instance variable specifies an initial value, then that value is computed and stored in the instance variable. Otherwise, the default initial value is used. 3. The actual parameters in the constructor, if any, are evaluated, and the values are assigned to the formal parameters of the constructor. 4. The statements in the body of the constructor, if any, are executed. 5. A reference to the object is returned as the value of the constructor call.   The end result of this is that you have a reference to a newly constructed object.  For another example, let's rewrite the *Student* class that was used in [Section 1](http://math.hws.edu/javanotes/c5/s1.html). I'll add a constructor, and I'll also take the opportunity to make the instance variable, name, private.  public class Student {  private String name; // Student's name.  public double test1, test2, test3; // Grades on three tests.    Student(String theName) {  // Constructor for Student objects;  // provides a name for the Student.  // The name can't be null.  if ( theName == null )  throw new IllegalArgumentException("name can't be null");  name = theName;  }    public String getName() {  // Getter method for reading the value of the private  // instance variable, name.  return name;  }    public double getAverage() {  // Compute average test grade.  return (test1 + test2 + test3) / 3;  }  } // end of class Student  An object of type *Student* contains information about some particular student. The constructor in this class has a parameter of type *String*, which specifies the name of that student. Objects of type *Student* can be created with statements such as:  std = new Student("John Smith");  std1 = new Student("Mary Jones");  In the original version of this class, the value of name had to be assigned by a program after it created the object of type *Student*. There was no guarantee that the programmer would always remember to set the name properly. In the new version of the class, there is no way to create a *Student* object except by calling the constructor, and that constructor automatically sets the name. Furthermore, the constructor makes it impossible to have a student object whose name is null. The programmer's life is made easier, and whole hordes of frustrating bugs are squashed before they even have a chance to be born.  Another type of guarantee is provided by the private modifier. Since the instance variable, name, is private, there is no way for any part of the program outside the *Student* class to get at the name directly. The program sets the value of name, indirectly, when it calls the constructor. I've provided a getter function, getName(), that can be used from outside the class to find out the name of the student. But I haven't provided any setter method or other way to change the name. Once a student object is created, it keeps the same name as long as it exists.  Note that it would be legal, and good style, to declare the variable name to be "final" in this class. An instance variable can be final provided it is either assigned a value in its declaration or is assigned a value in every constructor in the class. It is illegal to assign a value to a final instance variable, except inside a constructor.  Let's take this example a little farther to illustrate one more aspect of classes: What happens when you mix static and non-static in the same class? In that case, it's legal for an instance method in the class to use static member variables or call static member subroutines. An object knows what class it belongs to, and it can refer to static members of that class. But there it only one copy of the static member, in the class itself. Effectively, all the objects share one copy of the static member.  As an example, consider a version of the *Student* class to which I've added an ID for each student and a static member called nextUniqueID. Although there is an ID variable in each student object, there is only one nextUniqueID variable.  public class Student {  private String name; // Student's name.  public double test1, test2, test3; // Grades on three tests.    private int ID; // Unique ID number for this student.  private static int nextUniqueID = 0;  // keep track of next available unique ID number    Student(String theName) {  // Constructor for Student objects; provides a name for the Student,  // and assigns the student a unique ID number.  name = theName;  nextUniqueID++;  ID = nextUniqueID;  }    public String getName() {  // Getter method for reading the value of the private  // instance variable, name.  return name;  }    public int getID() {  // Getter method for reading the value of ID.  return ID;  }    public double getAverage() {  // Compute average test grade.  return (test1 + test2 + test3) / 3;  }    } // end of class Student  Since nextUniqueID is a static variable, the initialization "nextUniqueID = 0" is done only once, when the class is first loaded. Whenever a *Student* object is constructed and the constructor says "nextUniqueID++;", it's always the same static member variable that is being incremented. When the very first *Student* object is created, nextUniqueID becomes 1. When the second object is created, nextUniqueID becomes 2. After the third object, it becomes 3. And so on. The constructor stores the new value of nextUniqueID in the ID variable of the object that is being created. Of course, ID is an instance variable, so every object has its own individual ID variable. The class is constructed so that each student will automatically get a different value for its ID variable. Furthermore, the ID variable is private, so there is no way for this variable to be tampered with after the object has been created. You are guaranteed, just by the way the class is designed, that every student object will have its own permanent, unique identification number. Which is kind of cool if you think about it.  (Unfortunately, if you think about it a bit more, it turns out that the guarantee isn't quite absolute. The guarantee is valid in programs that use a single thread. But, as a preview of the difficulties of parallel programming, I'll note that in multi-threaded programs, where several things can be going on at the same time, things can get a bit strange. In a multi-threaded program, it is possible that two threads are creating *Student* objects at exactly the same time, and it becomes possible for both objects to get the same ID number. We'll come back to this in [Subsection 12.1.3](http://math.hws.edu/javanotes/c12/s1.html#threads.1.3), where you will learn how to fix the problem.) 5.2.3  Garbage Collection So far, this section has been about creating objects. What about destroying them? In Java, the destruction of objects takes place automatically.  An object exists in the heap, and it can be accessed only through variables that hold references to the object. What should be done with an object if there are no variables that refer to it? Such things can happen. Consider the following two statements (though in reality, you'd never do anything like this in consecutive statements!):  Student std = new Student("John Smith");  std = null;  In the first line, a reference to a newly created *Student* object is stored in the variable std. But in the next line, the value of std is changed, and the reference to the *Student* object is gone. In fact, there are now no references whatsoever to that object, in any variable. So there is no way for the program ever to use the object again! It might as well not exist. In fact, the memory occupied by the object should be reclaimed to be used for another purpose.  Java uses a procedure called garbage collection to reclaim memory occupied by objects that are no longer accessible to a program. It is the responsibility of the system, not the programmer, to keep track of which objects are "garbage." In the above example, it was very easy to see that the *Student* object had become garbage. Usually, it's much harder. If an object has been used for a while, there might be several references to the object stored in several variables. The object doesn't become garbage until all those references have been dropped.  In many other programming languages, it's the programmer's responsibility to delete the garbage. Unfortunately, keeping track of memory usage is very error-prone, and many serious program bugs are caused by such errors. A programmer might accidently delete an object even though there are still references to that object. This is called a dangling pointer error, and it leads to problems when the program tries to access an object that is no longer there. Another type of error is a memory leak, where a programmer neglects to delete objects that are no longer in use. This can lead to filling memory with objects that are completely inaccessible, and the program might run out of memory even though, in fact, large amounts of memory are being wasted.  Because Java uses garbage collection, such errors are simply impossible. Garbage collection is an old idea and has been used in some programming languages since the 1960s. You might wonder why all languages don't use garbage collection. In the past, it was considered too slow and wasteful. However, research into garbage collection techniques combined with the incredible speed of modern computers have combined to make garbage collection feasible. Programmers should rejoice. |
| |  | | --- | | **0. Static Blocks**    :    [**Link**](http://www.jusfortechies.com/java/core-java/static-blocks.php) | |

**Static Blocks in Java**

In java you see "static variables", "static methods", "static classes" and "static blocks". Static variables, static methods and static classes are known to everyone but what is this "static block". Lets see what, where and how these static blocks are used.

But before going into "static block", lets refresh what other static stuff are. Now "static variables" are class variables i.e., there will be only one copy for each class and not one copy for each object of the class and these variables will be accessed without instantiating the class. Then what are static methods. Again they are class methods i.e., they can be accessed without creating an instance of the class and like static variables, static methods will be accessed without instantiating the class. Note that static methods cannot access instance variables. They can access only static variables. Next what are static classes. You cannot declare a top-level class as a static class. Java will throw a compilation error. Only inner classes that are member classes can be declared as static. If we declare member classes as static, we can use it as a top-level class outside the context of top-level class. One catch here is "The static keyword does not do to a class declaration what it does to a variable or a method declaration." - what it means is say for example you have a static variable, then to access that static variable you will use the notation   
*<<Class Name>>.<<Variable Name>>*  
but when you want to use the static inner class, you need to instantiate like   
*<<Top-level class name>>.<<Inner static class name>> newClass = new <<Top-level class name>>.<<Inner static class name>>();*

*Static blocks*are also called *Static initialization blocks*. A static initialization block is a normal block of code enclosed in braces, { }, and preceded by the static keyword. Here is an example:

static {

// whatever code is needed for initialization goes here

}

A class can have any number of static initialization blocks, and they can appear anywhere in the class body. The runtime system guarantees that static initialization blocks are called in the order that they appear in the source code. And dont forget, this code will be executed when JVM loads the class. JVM combines all these blocks into one single static block and then executes. Here are a couple of points I like to mention:

* If you have executable statements in the static block, JVM will automatically execute these statements when the class is loaded into JVM.
* If you’re referring some static variables/methods from the static blocks, these statements will be executed after the class is loaded into JVM same as above i.e., now the static variables/methods referred and the static block both will be executed.

Lets see an example:

public class StaticExample{

static {

System.out.println("This is first static block");

}

public StaticExample(){

System.out.println("This is constructor");

}

public static String staticString = "Static Variable";

static {

System.out.println("This is second static block and "

+ staticString);

}

public static void main(String[] args){

StaticExample statEx = new StaticExample();

StaticExample.staticMethod2();

}

static {

staticMethod();

System.out.println("This is third static block");

}

public static void staticMethod() {

System.out.println("This is static method");

}

public static void staticMethod2() {

System.out.println("This is static method2");

}

}

What will happen when you execute the above code? You will see below output.

This is first static block

This is second static block and Static Variable

This is static method

This is third static block

This is constructor

This is static method2

Now lets the output. First all static blocks are positioned in the code and they are executed when the class is loaded into JVM. Since the static method staticMethod() is called inside third static block, its executed before calling the main method. But the staticMethod2() static method is executed after the class is instantiated because it is being called after the instantiation of the class.

Again if you miss to precede the block with "static" keyword, the block is called "constructor block" and will be executed when the class is instantiated. The constructor block will be copied into each constructor of the class. Say for example you have four parameterized constructors, then four copies of contructor blocks will be placed inside the constructor, one for each. Lets execute the below example and see the output.

public class ConstructorBlockExample{

{

System.out.println("This is first constructor block");

}

public ConstructorBlockExample(){

System.out.println("This is no parameter constructor");

}

public ConstructorBlockExample(String param1){

System.out.println("This is single parameter constructor");

}

public ConstructorBlockExample(String param1, String param2){

System.out.println("This is two parameters constructor");

}

{

System.out.println("This is second constructor block");

}

public static void main(String[] args){

ConstructorBlockExample constrBlockEx =

new ConstructorBlockExample();

ConstructorBlockExample constrBlockEx1 =

new ConstructorBlockExample("param1");

ConstructorBlockExample constrBlockEx2 =

new ConstructorBlockExample("param1", "param2");

}

}

The output is.

This is first constructor block

This is second constructor block

This is no parameter constructor

This is first constructor block

This is second constructor block

This is single parameter constructor

This is first constructor block

This is second constructor block

This is two parameters constructor

The above example is self-explanatory.

Now lets go back to static blocks.

There is an alternative to static blocks —you can write a private static method.

class PrivateStaticMethodExample {

public static varType myVar = initializeClassVariable();

private static varType initializeClassVariable() {

//initialization code goes here

}

}

The advantage of private static methods is that they can be reused later if you need to reinitialize the class variable. So, you kind of get more flexibility with a private static method in comparison to the corresponding static initialization block. This should not mislead that a 'public' static method can't do the same. But, we are talking about a way of initializing a class variable and there is hardly any reason to make such a method 'public'.

*So what are the advantages of static blocks?*

* If you’re loading drivers and other items into the namespace. For ex, Class class has a static block where it registers the natives.
* If you need to do computation in order to initialize your static variables,you can declare a static block which gets executed exactly once,when the class is first loaded.
* Security related issues or logging related tasks

*Ofcourse there are limitations for static blocks*

* There is a limitation of JVM that a static initializer block should not exceed 64K.
* You cannot throw Checked Exceptions.
* You cannot use this keyword since there is no instance.
* You shouldn’t try to access super since there is no such a thing for static blocks.
* You should not return anything from this block.
* Static blocks make testing a nightmare.

*Finally how to handle Exceptions in static blocks?*  
In methods, an exception can be handled by either passing through the Exception or handling it. But in a static block code, you cannot handle exceptions this way.

Generally a clean way to handle it is using a try-catch block but here since we dont have this option lets look at the available three options.

First: After logging the exception throw a RuntimeException which will end the current thread (unless caught by code instantiating / calling a static method on the class for the first time).

Second is calling System.exit(1) but this is not desirable in a managed environment like a servlet. This option is only for java applications and only if the static initializer block performs some critical (without which the program cannot be run successfully) function like loading the database driver.

Third and final option is to set a flag indicating failure. Later the constructors can check the flag and throw exceptions or retry in rare cases.

Finally, if the operation is not important to the functioning of the program then maybe a simple log entry is all that is required.

**Polymorphism**

In this tutorial we will learn the concepts of Ploymorphism and how to achieve polymorphism at run time.

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| |  |  |  |  | | --- | --- | --- | --- | | |  | | --- | | **0. Overriding methods**    :    [**Link**](http://www.javatpoint.com/runtime-polymorphism-in-java) |  Polymorphism in Java **Polymorphism in java** is a concept by which we can perform a single action by different ways. Polymorphism is derived from 2 greek words: poly and morphs. The word "poly" means many and "morphs" means forms. So polymorphism means many forms.  There are two types of polymorphism in java: compile time polymorphism and runtime polymorphism. We can perform polymorphism in java by method overloading and method overriding.  If you overload static method in java, it is the example of compile time polymorphism. Here, we will focus on runtime polymorphism in java. Runtime Polymorphism in Java **Runtime polymorphism** or **Dynamic Method Dispatch** is a process in which a call to an overridden method is resolved at runtime rather than compile-time.  In this process, an overridden method is called through the reference variable of a superclass. The determination of the method to be called is based on the object being referred to by the reference variable.  Let's first understand the upcasting before Runtime Polymorphism. Upcasting When reference variable of Parent class refers to the object of Child class, it is known as upcasting. For example:  Upcasting in java   1. **class** A{} 2. **class** B **extends** A{} 3. A a=**new** B();//upcasting  Example of Java Runtime Polymorphism In this example, we are creating two classes Bike and Splendar. Splendar class extends Bike class and overrides its run() method. We are calling the run method by the reference variable of Parent class. Since it refers to the subclass object and subclass method overrides the Parent class method, subclass method is invoked at runtime.  Since method invocation is determined by the JVM not compiler, it is known as runtime polymorphism.   1. **class** Bike{ 2. **void** run(){System.out.println("running");} 3. } 4. **class** Splender **extends** Bike{ 5. **void** run(){System.out.println("running safely with 60km");} 7. **public** **static** **void** main(String args[]){ 8. Bike b = **new** Splender();//upcasting 9. b.run(); 10. } 11. }   [**Test it Now**](http://www.javatpoint.com/opr/test.jsp?filename=Splender)  Output:running safely with 60km. Real example of Java Runtime Polymorphism Consider a scenario, Bank is a class that provides method to get the rate of interest. But, rate of interest may differ according to banks. For example, SBI, ICICI and AXIS banks could provide 8%, 7% and 9% rate of interest.  Java Runtime Polymorphism example of bank  Note: It is also given in method overriding but there was no upcasting.   1. **class** Bank{ 2. **int** getRateOfInterest(){**return** 0;} 3. } 5. **class** SBI **extends** Bank{ 6. **int** getRateOfInterest(){**return** 8;} 7. } 9. **class** ICICI **extends** Bank{ 10. **int** getRateOfInterest(){**return** 7;} 11. } 12. **class** AXIS **extends** Bank{ 13. **int** getRateOfInterest(){**return** 9;} 14. } 16. **class** Test3{ 17. **public** **static** **void** main(String args[]){ 18. Bank b1=**new** SBI(); 19. Bank b2=**new** ICICI(); 20. Bank b3=**new** AXIS(); 21. System.out.println("SBI Rate of Interest: "+b1.getRateOfInterest()); 22. System.out.println("ICICI Rate of Interest: "+b2.getRateOfInterest()); 23. System.out.println("AXIS Rate of Interest: "+b3.getRateOfInterest()); 24. } 25. }   [**Test it Now**](http://www.javatpoint.com/opr/test.jsp?filename=Test3)  Output:  SBI Rate of Interest: 8  ICICI Rate of Interest: 7  AXIS Rate of Interest: 9 Java Runtime Polymorphism with data member  |  | | --- | | Method is overridden not the datamembers, so runtime polymorphism can't be achieved by data members. | | In the example given below, both the classes have a datamember speedlimit, we are accessing the datamember by the reference variable of Parent class which refers to the subclass object. Since we are accessing the datamember which is not overridden, hence it will access the datamember of Parent class always. |  Rule: Runtime polymorphism can't be achieved by data members.  1. **class** Bike{ 2. **int** speedlimit=90; 3. } 4. **class** Honda3 **extends** Bike{ 5. **int** speedlimit=150; 7. **public** **static** **void** main(String args[]){ 8. Bike obj=**new** Honda3(); 9. System.out.println(obj.speedlimit);//90 10. }   [**Test it Now**](http://www.javatpoint.com/opr/test.jsp?filename=Honda3)  Output:90 Java Runtime Polymorphism with Multilevel Inheritance Let's see the simple example of Runtime Polymorphism with multilevel inheritance.   1. **class** Animal{ 2. **void** eat(){System.out.println("eating");} 3. } 5. **class** Dog **extends** Animal{ 6. **void** eat(){System.out.println("eating fruits");} 7. } 9. **class** BabyDog **extends** Dog{ 10. **void** eat(){System.out.println("drinking milk");} 12. **public** **static** **void** main(String args[]){ 13. Animal a1,a2,a3; 14. a1=**new** Animal(); 15. a2=**new** Dog(); 16. a3=**new** BabyDog(); 18. a1.eat(); 19. a2.eat(); 20. a3.eat(); 21. } 22. }   [**Test it Now**](http://www.javatpoint.com/opr/test.jsp?filename=BabyDog)  Output: eating  eating fruits  drinking Milk Try for Output  1. **class** Animal{ 2. **void** eat(){System.out.println("animal is eating...");} 3. } 5. **class** Dog **extends** Animal{ 6. **void** eat(){System.out.println("dog is eating...");} 7. } 9. **class** BabyDog1 **extends** Dog{ 10. **public** **static** **void** main(String args[]){ 11. Animal a=**new** BabyDog1(); 12. a.eat(); 13. }}   [**Test it Now**](http://www.javatpoint.com/opr/test.jsp?filename=BabyDog1)  Output: Dog is eating  Since, BabyDog is not overriding the eat() method, so eat() method of Dog class is invoked. | | |  | | --- | | **0. Dynamic Method Dispatch**    :    [**Link**](http://www.studytonight.com/java/dynamic-method-dispatch.php)  **[Link](http://howtoprogramwithjava.com/java-practice-assignment-2/" \t "_blank)** |  Runtime Polymorphism or Dynamic method dispatch Dynamic method dispatch is a mechanism by which a call to an overridden method is resolved at runtime. This is how java implements runtime polymorphism. When an overridden method is called by a reference, java determines which version of that method to execute based on the type of object it refer to. In simple words the type of object which it referred determines which version of overridden method will be called.  upcasting in java Upcasting When **Parent** class reference variable refers to **Child** class object, it is known as **Upcasting** Example class **Game**  {  public void type()  { System.out.println("Indoor & outdoor"); }  }  Class **Cricket** extends **Game**  {  public void type()  { System.out.println("outdoor game"); }  public static void main(String[] args)  {  Game gm = new Game();  Cricket ck = new Cricket();  gm.type();  ck.type();  gm=ck; **//gm refers to Cricket object**  gm.type(); **//calls Cricket's version of type**  }  }  **Output:**  Indoor & outdoor  Outdoor game  Outdoor game  Notice the last output. This is because of **gm = ck**; Now gm.type() will call Cricket version of type method. Because here gm refers to cricket object. Q. Difference between Static binding and Dynamic binding in java ?  |  | | --- | | **0. Shadowing vs. Hiding**    :    [**Link**](http://www.xyzws.com/Javafaq/what-is-variable-hiding-and-shadowing/15)  **[Link](http://techsymphony.wordpress.com/2011/01/23/difference-between-hiding-and-shadowing-in-java/" \t "_blank)** |   Static binding in Java occurs during compile time while dynamic binding occurs during runtime. Static binding uses type(Class) information for binding while dynamic binding uses instance of class(Object) to resolve calling of method at run-time. Overloaded methods are bonded using static binding while overridden methods are bonded using dynamic binding at runtime.  « [Stack and heap memory](https://techsymphony.wordpress.com/2011/01/23/stack-and-heap-memory/)  [OO principles from Head First Design Pattern](https://techsymphony.wordpress.com/2011/02/04/oo-principles-from-head-first-design-pattern/) » Difference between hiding and shadowing in java January 23, 2011 by [novice](https://techsymphony.wordpress.com/author/techsymphony/)  **Hiding** : Happens during inheritance (between superclass and subclass).  **Shadowing** : Happens within a class (between a member variable and local variable).  **Examples** :  **Hiding** :  class Parent {  String city = “Dallas”;  }  class Child extends Parent {  String city =”Zurich”;  }  “city” in Child hides the one in Parent.  **Shadowing** :  class Main {  String city = “Dallas”;  void method() {  String city = “Zurich”;  }  }  “city”, local variable shadows the member variable. | | |  | | --- | | **0. Dynamic Method Lookup**    :    [**Link**](http://etutorials.org/Misc/programmers+guide+java+certification/Chapter+6.+Object-oriented+Programming/6.7+Polymorphism+and+Dynamic+Method+Lookup/)  **[Link](http://www.programr.com/practice/programr211_promo_228/node/576" \t "_blank)** | | | 6.7 Polymorphism and Dynamic Method Lookup Which object a reference will actually denote during runtime, cannot always be determined at compile time. Polymorphism allows a reference to denote objects of different types at different times during execution. A supertype reference exhibits polymorphic behavior, since it can denote objects of its subtypes.  When a non-private instance method is invoked on an object, the method definition actually executed is determined both by the type of the object at runtime and the method signature. Dynamic method lookup is the process of determining which method definition a method signature denotes during runtime, based on the type of the object. However, a call to a private instance method is not polymorphic. Such a call can only occur within the class, and gets bound to the private method implementation at compile time.  The inheritance hierarchy depicted in Figure 6.5 is implemented in Example 6.14. The implementation of the method draw() is overridden in all subclasses of the class Shape. The invocation of the draw() method in the two loops at (3) and (4) in Example 6.14, relies on the polymorphic behavior of references and dynamic method lookup. The array shapes holds Shape references denoting a Circle, a Rectangle and a Square, as shown at (1). At runtime, dynamic lookup determines the draw() implementation to execute, based on the type of the object denoted by each element in the array. This is also the case for the elements of the array drawables at (2), which holds IDrawable references that can be assigned any object of a class that implements the IDrawable interface. The first loop will still work without any change if objects of new subclasses of the class Shape are added to the array shapes. If they did not override the draw() method, then an inherited version of the method would be executed. This polymorphic behavior applies to the array drawables, where the subtype objects are guaranteed to have implemented the IDrawable interface. **Figure 6.5. Polymorphic Methods** Chapter 9: **Abstract Classes and Interfaces**  In this tutorial we will learn how to use abstract class and interfaces. Also will understand the difference between them and when to use what.   |  |  |  |  | | --- | --- | --- | --- | | |  |  | | --- | --- | | |  | | --- | | **0. Concept of Abstract Classes and Interfaces**    :    [**Link**](http://www.javatpoint.com/abstract-class-in-java)  **[Link](http://www.javatpoint.com/interface-in-java" \t "_blank)**  **[Link](http://www.javaworld.com/javaqa/2001-04/03-qa-0420-abstract.html" \t "_blank)**  **[Link](http://www.javabeginner.com/learn-java/java-abstract-class-and-interface" \t "_blank)** **[Link](http://etutorials.org/Misc/programmers+guide+java+certification/Chapter+6.+Object-oriented+Programming/Programming+Exercises/" \t "_blank)**  **[Link](http://www.programr.com/practice/programr211_promo_228/node/544" \t "_blank)** | | |  |  Abstract class in Java A class that is declared with abstract keyword, is known as abstract class in java. It can have abstract and non-abstract methods (method with body).  Before learning java abstract class, let's understand the abstraction in java first. Abstraction in Java **Abstraction** is a process of hiding the implementation details and showing only functionality to the user.  Another way, it shows only important things to the user and hides the internal details for example sending sms, you just type the text and send the message. You don't know the internal processing about the message delivery.  Abstraction lets you focus on what the object does instead of how it does it. Ways to achieve Abstaction There are two ways to achieve abstraction in java   1. Abstract class (0 to 100%) 2. Interface (100%)  Abstract class in Java A class that is declared as abstract is known as **abstract class**. It needs to be extended and its method implemented. It cannot be instantiated. Example abstract class  1. **abstract** **class** A{}  abstract method  |  | | --- | | A method that is declared as abstract and does not have implementation is known as abstract method. |  Example abstract method  1. **abstract** **void** printStatus();//no body and abstract  Example of abstract class that has abstract method In this example, Bike the abstract class that contains only one abstract method run. It implementation is provided by the Honda class.   1. **abstract** **class** Bike{ 2. **abstract** **void** run(); 3. } 5. **class** Honda4 **extends** Bike{ 6. **void** run(){System.out.println("running safely..");} 8. **public** **static** **void** main(String args[]){ 9. Bike obj = **new** Honda4(); 10. obj.run(); 11. } 12. }   [**Test it Now**](http://www.javatpoint.com/opr/test.jsp?filename=Honda4)  running safely.. Understanding the real scenario of abstract class In this example, Shape is the abstract class, its implementation is provided by the Rectangle and Circle classes. Mostly, we don't know about the implementation class (i.e. hidden to the end user) and object of the implementation class is provided by the **factory method**.  A **factory method** is the method that returns the instance of the class. We will learn about the factory method later.  In this example, if you create the instance of Rectangle class, draw() method of Rectangle class will be invoked.  *File: TestAbstraction1.java*   1. **abstract** **class** Shape{ 2. **abstract** **void** draw(); 3. } 4. //In real scenario, implementation is provided by others i.e. unknown by end user 5. **class** Rectangle **extends** Shape{ 6. **void** draw(){System.out.println("drawing rectangle");} 7. } 9. **class** Circle1 **extends** Shape{ 10. **void** draw(){System.out.println("drawing circle");} 11. } 13. //In real scenario, method is called by programmer or user 14. **class** TestAbstraction1{ 15. **public** **static** **void** main(String args[]){ 16. Shape s=**new** Circle1();//In real scenario, object is provided through method e.g. getShape() method 17. s.draw(); 18. } 19. }   [**Test it Now**](http://www.javatpoint.com/opr/test.jsp?filename=TestAbstraction1)  drawing circle Another example of abstract class in java *File: TestBank.java*   1. **abstract** **class** Bank{ 2. **abstract** **int** getRateOfInterest(); 3. } 5. **class** SBI **extends** Bank{ 6. **int** getRateOfInterest(){**return** 7;} 7. } 8. **class** PNB **extends** Bank{ 9. **int** getRateOfInterest(){**return** 7;} 10. } 12. **class** TestBank{ 13. **public** **static** **void** main(String args[]){ 14. Bank b=**new** SBI();//if object is PNB, method of PNB will be invoked 15. **int** interest=b.getRateOfInterest(); 16. System.out.println("Rate of Interest is: "+interest+" %"); 17. }}   [**Test it Now**](http://www.javatpoint.com/opr/test.jsp?filename=TestBank)  Rate of Interest is: 7 % Abstract class having constructor, data member, methods etc. An abstract class can have data member, abstract method, method body, constructor and even main() method.  *File: TestAbstraction2.java*   1. //example of abstract class that have method body 2. **abstract** **class** Bike{ 3. Bike(){System.out.println("bike is created");} 4. **abstract** **void** run(); 5. **void** changeGear(){System.out.println("gear changed");} 6. } 8. **class** Honda **extends** Bike{ 9. **void** run(){System.out.println("running safely..");} 10. } 11. **class** TestAbstraction2{ 12. **public** **static** **void** main(String args[]){ 13. Bike obj = **new** Honda(); 14. obj.run(); 15. obj.changeGear(); 16. } 17. }   [**Test it Now**](http://www.javatpoint.com/opr/test.jsp?filename=TestAbstraction2)  bike is created  running safely..  gear changed Rule: If there is any abstract method in a class, that class must be abstract.  1. **class** Bike12{ 2. **abstract** **void** run(); 3. }   [**Test it Now**](http://www.javatpoint.com/opr/test.jsp?filename=Bike12)  compile time error Rule: If you are extending any abstract class that have abstract method, you must either provide the implementation of the method or make this class abstract.Another real scenario of abstract class The abstract class can also be used to provide some implementation of the interface. In such case, the end user may not be forced to override all the methods of the interface. Note: If you are beginner to java, learn interface first and skip this example.  1. **interface** A{ 2. **void** a(); 3. **void** b(); 4. **void** c(); 5. **void** d(); 6. } 8. **abstract** **class** B **implements** A{ 9. **public** **void** c(){System.out.println("I am C");} 10. } 12. **class** M **extends** B{ 13. **public** **void** a(){System.out.println("I am a");} 14. **public** **void** b(){System.out.println("I am b");} 15. **public** **void** d(){System.out.println("I am d");} 16. } 18. **class** Test5{ 19. **public** **static** **void** main(String args[]){ 20. A a=**new** M(); 21. a.a(); 22. a.b(); 23. a.c(); 24. a.d(); 25. }}   [**Test it Now**](http://www.javatpoint.com/opr/test.jsp?filename=Test5)  Output:I am a  I am b  I am c  I am d Interface in Java  1. [Interface](http://www.javatpoint.com/interface-in-java) 2. [Example of Interface](http://www.javatpoint.com/interface-in-java#interfaceex) 3. [Multiple inheritance by Interface](http://www.javatpoint.com/interface-in-java#interfacemultiple) 4. [Why multiple inheritance is supported in Interface while it is not supported in case of class.](http://www.javatpoint.com/interface-in-java#interfacewhynot) 5. [Marker Interface](http://www.javatpoint.com/interface-in-java#interfacemarker) 6. [Nested Interface](http://www.javatpoint.com/nested-interface)   An **interface in java** is a blueprint of a class. It has static constants and abstract methods only.  The interface in java is **a mechanism to achieve fully abstraction**. There can be only abstract methods in the java interface not method body. It is used to achieve fully abstraction and multiple inheritance in Java.  Java Interface also **represents IS-A relationship**.  It cannot be instantiated just like abstract class. Why use Java interface? There are mainly three reasons to use interface. They are given below.   * It is used to achieve fully abstraction. * By interface, we can support the functionality of multiple inheritance. * It can be used to achieve loose coupling.  The java compiler adds public and abstract keywords before the interface method and public, static and final keywords before data members. In other words, Interface fields are public, static and final bydefault, and methods are public and abstract.  interface Understanding relationship between classes and interfaces As shown in the figure given below, a class extends another class, an interface extends another interface but a **class implements an interface**.  relationship between class and interface Simple example of Java interface  |  | | --- | | In this example, Printable interface have only one method, its implementation is provided in the A class. |  1. **interface** printable{ 2. **void** print(); 3. } 5. **class** A6 **implements** printable{ 6. **public** **void** print(){System.out.println("Hello");} 8. **public** **static** **void** main(String args[]){ 9. A6 obj = **new** A6(); 10. obj.print(); 11. } 12. }   [**Test it Now**](http://www.javatpoint.com/opr/test.jsp?filename=A6)  Output:Hello Multiple inheritance in Java by interface If a class implements multiple interfaces, or an interface extends multiple interfaces i.e. known as multiple inheritance.  multiple inheritance in java   1. **interface** Printable{ 2. **void** print(); 3. } 5. **interface** Showable{ 6. **void** show(); 7. } 9. **class** A7 **implements** Printable,Showable{ 11. **public** **void** print(){System.out.println("Hello");} 12. **public** **void** show(){System.out.println("Welcome");} 14. **public** **static** **void** main(String args[]){ 15. A7 obj = **new** A7(); 16. obj.print(); 17. obj.show(); 18. } 19. }   [**Test it Now**](http://www.javatpoint.com/opr/test.jsp?filename=A7)  Output:Hello  Welcome Q) Multiple inheritance is not supported through class in java but it is possible by interface, why?  |  | | --- | | As we have explained in the inheritance chapter, multiple inheritance is not supported in case of class. But it is supported in case of interface because there is no ambiguity as implementation is provided by the implementation class. For example: |  1. **interface** Printable{ 2. **void** print(); 3. } 4. **interface** Showable{ 5. **void** print(); 6. } 8. **class** TestTnterface1 **implements** Printable,Showable{ 9. **public** **void** print(){System.out.println("Hello");} 10. **public** **static** **void** main(String args[]){ 11. TestTnterface1 obj = **new** TestTnterface1(); 12. obj.print(); 13. } 14. }   [**Test it Now**](http://www.javatpoint.com/opr/test.jsp?filename=testinterface1)  Hello  As you can see in the above example, Printable and Showable interface have same methods but its implementation is provided by class TestTnterface1, so there is no ambiguity. Interface inheritance A class implements interface but one interface extends another interface .   1. **interface** Printable{ 2. **void** print(); 3. } 4. **interface** Showable **extends** Printable{ 5. **void** show(); 6. } 7. **class** Testinterface2 **implements** Showable{ 9. **public** **void** print(){System.out.println("Hello");} 10. **public** **void** show(){System.out.println("Welcome");} 12. **public** **static** **void** main(String args[]){ 13. Testinterface2 obj = **new** Testinterface2(); 14. obj.print(); 15. obj.show(); 16. } 17. }   [**Test it Now**](http://www.javatpoint.com/opr/test.jsp?filename=Testinterface2)  Hello  Welcome Q) What is marker or tagged interface? An interface that have no member is known as marker or tagged interface. For example: Serializable, Cloneable, Remote etc. They are used to provide some essential information to the JVM so that JVM may perform some useful operation.   1. //How Serializable interface is written? 2. **public** **interface** Serializable{ 3. }  Nested Interface in Java Note: An interface can have another interface i.e. known as nested interface. We will learn it in detail in the nested classes chapter. For example:   1. **interface** printable{ 2. **void** print(); 3. **interface** MessagePrintable{ 4. **void** msg(); 5. } 6. }  ABSTRACT CLASS IN JAVA **Java Abstract classes** are used to declare common characteristics of subclasses. An abstract class cannot be instantiated. It can only be used as a superclass for other classes that extend the abstract class. Abstract classes are declared with the abstract keyword. Abstract classes are used to provide a template or design for concrete subclasses down the inheritance tree.  Like any other class, an abstract class can contain fields that describe the characteristics and methods that describe the actions that a class can perform. An abstract class can include methods that contain no implementation. These are called abstract methods. The abstract method declaration must then end with a semicolon rather than a block. If a class has any abstract methods, whether declared or inherited, the entire class must be declared abstract. Abstract methods are used to provide a template for the classes that inherit the abstract methods.  Abstract classes cannot be instantiated; they must be subclassed, and actual implementations must be provided for the abstract methods. Any implementation specified can, of course, be overridden by additional subclasses. An object must have an implementation for all of its methods. You need to create a subclass that provides an implementation for the abstract method.  A class abstract Vehicle might be specified as abstract to represent the general abstraction of a vehicle, as creating instances of the class would not be meaningful.  abstract class Vehicle {  int numofGears;  String color;  abstract boolean hasDiskBrake();  abstract int getNoofGears();  }  Example of a shape class as an abstract class  abstract class Shape {  public String color;  public Shape() {  }  public void setColor(String c) {  color = c;  }  public String getColor() {  return color;  }  abstract public double area();  }  We can also implement the generic shapes class as an abstract class so that we can draw lines, circles, triangles etc. All shapes have some common fields and methods, but each can, of course, add more fields and methods. The abstract class guarantees that each shape will have the same set of basic properties. We declare this class abstract because there is no such thing as a generic shape. There can only be concrete shapes such as squares, circles, triangles etc.  public class Point extends Shape {  static int x, y;  public Point() {  x = 0;  y = 0;  }  public double area() {  return 0;  }  public double perimeter() {  return 0;  }  public static void print() {  System.out.println("point: " + x + "," + y);  }  public static void main(String args[]) {  Point p = new Point();  p.print();  }  }  **Output**  point: 0, 0  Notice that, in order to create a Point object, its class cannot be abstract. This means that all of the abstract methods of the Shape class must be implemented by the Point class.  The subclass must define an implementation for every abstract method of the abstract superclass, or the subclass itself will also be abstract. Similarly other shape objects can be created using the generic Shape Abstract class.  A big Disadvantage of using abstract classes is not able to use multiple inheritance. In the sense, when a class extends an abstract class, it can’t extend any other class. JAVA INTERFACE In Java, this multiple inheritance problem is solved with a powerful construct called **interfaces**. Interface can be used to define a generic template and then one or more abstract classes to define partial implementations of the interface. Interfaces just specify the method declaration (implicitly public and abstract) and can only contain fields (which are implicitly public static final). Interface definition begins with a keyword interface. An interface like that of an abstract class cannot be instantiated.  Multiple Inheritance is allowed when extending interfaces i.e. one interface can extend none, one or more interfaces. Java does not support multiple inheritance, but it allows you to extend one class and implement many interfaces.  If a class that implements an interface does not define all the methods of the interface, then it must be declared abstract and the method definitions must be provided by the subclass that extends the abstract class.  **Example 1**: Below is an example of a Shape interface  interface Shape {  public double area();  public double volume();  }  Below is a Point class that implements the Shape interface.  public class Point implements Shape {  static int x, y;  public Point() {  x = 0;  y = 0;  }  public double area() {  return 0;  }  public double volume() {  return 0;  }  public static void print() {  System.out.println("point: " + x + "," + y);  }  public static void main(String args[]) {  Point p = new Point();  p.print();  }  }  Similarly, other shape objects can be created by interface programming by implementing generic Shape Interface.  **Example 2:** Below is a java interfaces program showing the power of interface programming in java  Listing below shows 2 interfaces and 4 classes one being an abstract class. Note: The method *toString* in class *A1* is an overridden version of the method defined in the class named**Object**. The classes B1 and *C1* satisfy the interface contract. But since the class**D1** does not define all the methods of the implemented interface *I2*, the class D1 is declared abstract. Also, i1.methodI2() produces a compilation error as the method is not declared in *I1* or any of its super interfaces if present. Hence a downcast of interface reference I1 solves the problem as shown in the program. The same problem applies to i1.methodA1(), which is again resolved by a downcast.  When we invoke the toString() method which is a method of an Object, there does not seem to be any problem as every interface or class extends Object and any class can override the default toString() to suit your application needs. ((C1)o1).methodI1() compiles successfully, but produces a ClassCastException at runtime. This is because B1 does not have any relationship with C1 except they are “siblings”. You can’t cast siblings into one another.  When a given interface method is invoked on a given reference, the behavior that results will be appropriate to the class from which that particular object was instantiated. This is runtime polymorphism based on interfaces and overridden methods.  interface I1 {  void methodI1(); // public static by default  }  interface I2 extends I1 {  void methodI2(); // public static by default  }  class A1 {  public String methodA1() {  String strA1 = "I am in methodC1 of class A1";  return strA1;  }  public String toString() {  return "toString() method of class A1";  }  }  class B1 extends A1 implements I2 {  public void methodI1() {  System.out.println("I am in methodI1 of class B1");  }  public void methodI2() {  System.out.println("I am in methodI2 of class B1");  }  }  class C1 implements I2 {  public void methodI1() {  System.out.println("I am in methodI1 of class C1");  }  public void methodI2() {  System.out.println("I am in methodI2 of class C1");  }  }  // Note that the class is declared as abstract as it does not  // satisfy the interface contract  abstract class D1 implements I2 {  public void methodI1() {  }  // This class does not implement methodI2() hence declared abstract.  }  public class InterFaceEx {  public static void main(String[] args) {  I1 i1 = new B1();  i1.methodI1(); // OK as methodI1 is present in B1  // i1.methodI2(); Compilation error as methodI2 not present in I1  // Casting to convert the type of the reference from type I1 to type I2  ((I2) i1).methodI2();  I2 i2 = new B1();  i2.methodI1(); // OK  i2.methodI2(); // OK  // Does not Compile as methodA1() not present in interface reference I1  // String var = i1.methodA1();  // Hence I1 requires a cast to invoke methodA1  String var2 = ((A1) i1).methodA1();  System.out.println("var2 : " + var2);  String var3 = ((B1) i1).methodA1();  System.out.println("var3 : " + var3);  String var4 = i1.toString();  System.out.println("var4 : " + var4);  String var5 = i2.toString();  System.out.println("var5 : " + var5);  I1 i3 = new C1();  String var6 = i3.toString();  System.out.println("var6 : " + var6); // It prints the Object toString() method  Object o1 = new B1();  // o1.methodI1(); does not compile as Object class does not define  // methodI1()  // To solve the probelm we need to downcast o1 reference. We can do it  // in the following 4 ways  ((I1) o1).methodI1(); // 1  ((I2) o1).methodI1(); // 2  ((B1) o1).methodI1(); // 3  /\*  \*  \* B1 does not have any relationship with C1 except they are "siblings".  \*  \* Well, you can't cast siblings into one another.  \*  \*/  // ((C1)o1).methodI1(); Produces a ClassCastException  }  }  **Output**  I am in methodI1 of class B1 I am in methodI2 of class B1 I am in methodI1 of class B1 I am in methodI2 of class B1 var2 : I am in methodC1 of class A1 var3 : I am in methodC1 of class A1 var4 : toString() method of class A1 var5 : toString() method of class A1 var6 : C1@190d11 I am in methodI1 of class B1 I am in methodI1 of class B1 I am in methodI1 of class B1  Polymorphism and dynamic method lookup form a powerful programming paradigm that simplifies client definitions, encourages object decoupling, and supports dynamically changing relationships between objects at runtime. **Example 6.14 Polymorphism and Dynamic Method Lookup** interface IDrawable {  void draw();  }  class Shape implements IDrawable {  public void draw() { System.out.println("Drawing a Shape."); }  }  class Circle extends Shape {  public void draw() { System.out.println("Drawing a Circle."); }  }  class Rectangle extends Shape {  public void draw() { System.out.println("Drawing a Rectangle."); }  }  class Square extends Rectangle {  public void draw() { System.out.println("Drawing a Square."); }  }  class Map implements IDrawable {  public void draw() { System.out.println("Drawing a Map."); }  }  public class PolymorphRefs {  public static void main(String[] args) {  Shape[] shapes = {new Circle(), new Rectangle(), new Square()}; // (1)  IDrawable[] drawables = {new Shape(), new Rectangle(), new Map()};// (2)  System.out.println("Draw shapes:");  for (int i = 0; i < shapes.length; i++) // (3)  shapes[i].draw();  System.out.println("Draw drawables:");  for (int i = 0; i < drawables.length; i++) // (4)  drawables[i].draw();  }  }  Output from the program:  Draw shapes:  Drawing a Circle.  Drawing a Rectangle.  Drawing a Square.  Draw drawables:  Drawing a Shape.  Drawing a Rectangle.  Drawing a Map.  - See more at: http://etutorials.org/Misc/programmers+guide+java+certification/Chapter+6.+Object-oriented+Programming/6.7+Polymorphism+and+Dynamic+Method+Lookup/#sthash.rOA34WSM.dpuf | |