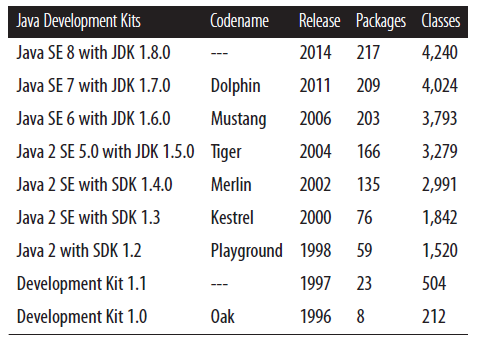
Java technology is both a programming language and a platform.

1. JavaSE(Standard Edition) : Desktop Application
2. JavaEE(Enterprise Edition) : Web Application
3. JavaME(Micro Edition) : Mobile Application

**The Java Programming Language**

The Java programming language is a high-level language that can be characterized by all of the following buzzwords:

|  |  |
| --- | --- |
| * Simple * Object oriented * Distributed * Multithreaded * Dynamic | * Architecture neutral * Portable * High performance * Robust * Secure |



In the Java programming language, all source code is first written in plain text files ending with the .java extension. Those source files are then compiled into .class files by the javac compiler. A .class file does not contain code that is native to your processor; it instead contains *bytecodes* — the machine language of the Java Virtual Machine (Java VM). The java launcher tool then runs your application with an instance of the Java Virtual Machine.



An overview of the software development process.

Because the Java VM is available on many different operating systems, the same .class files are capable of running on Microsoft Windows, the Solaris™ Operating System (Solaris OS), Linux, or Mac OS. Some virtual machines, such as the Java SE HotSpot, perform additional steps at runtime to give your application a performance boost. This includes various tasks such as finding performance bottlenecks and recompiling (to native code) frequently used sections of code.

Through the Java VM, the same application is capable of running on multiple platforms.



**The Java Platform**

A *platform* is the hardware or software environment in which a program runs. We've already mentioned some of the most popular platforms like Microsoft Windows, Linux, Solaris OS, and Mac OS. Most platforms can be described as a combination of the operating system and underlying hardware. The Java platform differs from most other platforms in that it's a software-only platform that runs on top of other hardware-based platforms.

The Java platform has two components:

* The *Java Virtual Machine*
* The *Java Application Programming Interface* (API)

The API is a large collection of ready-made software components that provide many useful capabilities. It is grouped into libraries of related classes and interfaces; these libraries are known as packages.

## Creating Your First Application

1. **The Java SE Development Kit (JDK)**

(Make sure you download the **JDK**, *not* the JRE.)

1. **A text editor**

In this example, we'll use Notepad, a simple editor included with the Windows platforms. You can easily adapt these instructions if you use a different text editor.

These two items are all you'll need to write your first application.

Your first application, HelloWorldApp, will simply display the greeting "Hello world!” To create this program, you will:

* **Create a source file**

A source file contains code, written in the Java programming language, that you and other programmers can understand. You can use any text editor to create and edit source files.

* **Compile the source file into a .class file**

The Java programming language *compiler* (javac) takes your source file and translates its text into instructions that the Java virtual machine can understand. The instructions contained within this file are known as *bytecodes*.

* **Run the program**

The Java application *launcher tool* (java) uses the Java virtual machine to run your application.

### Create a Source File

To create a source file, first, start your editor. You can launch the Notepad editor from the **Start** menu by selecting **Programs > Accessories > Notepad**. In a new document, type in the following code:

/\*\*

 \* The HelloWorldApp class implements an application that

 \* simply prints "Hello World!" to standard output.

 \*/

class HelloWorldApp {

    public static void main(String[] args) {

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

    }

}

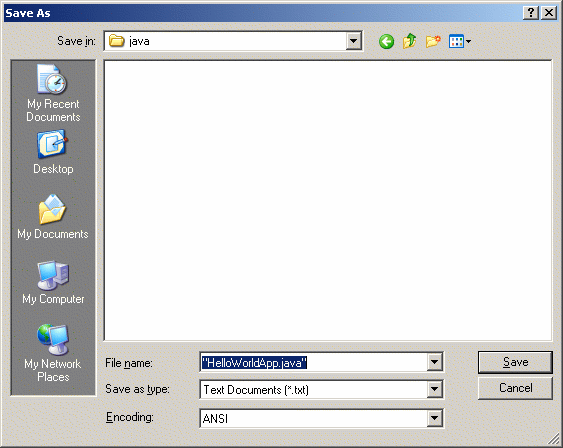
**Be Careful When You Type**

**Note:** Type all code, commands, and file names exactly as shown. Both the compiler (javac) and launcher (java) are case-sensitive, so you must capitalize consistently.  
  
HelloWorldApp is not the same as helloworldapp.

Save the code in a file with the name HelloWorldApp.java. To do this in Notepad, first choose the **File > Save As** menu item. Then, in the **Save As** dialog box:

1. Using the **Save in** combo box, specify the folder (directory) where you'll save your file. In this example, the directory is java on the C drive.
2. In the **File name** text field, type "HelloWorldApp.java", including the quotation marks.
3. From the **Save as type** combo box, choose **Text Documents (\*.txt)**.
4. In the **Encoding** combo box, leave the encoding as ANSI.

When you're finished, the dialog box should look like this.

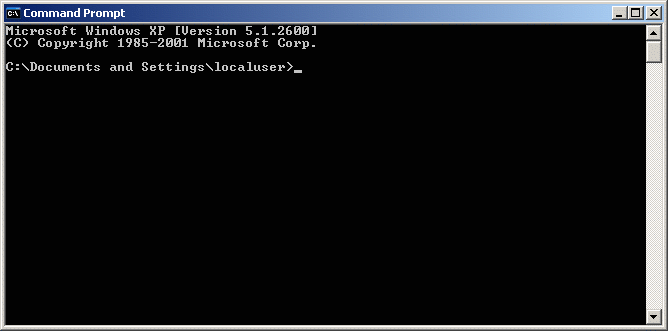


The Save As dialog just before you click **Save**.

Now click **Save**, and exit Notepad.

### Compile the Source File into a .class File

Bring up a shell, or "command," window. You can do this from the **Start** menu by choosing **Command Prompt** (Windows XP), or by choosing **Run...** and then entering cmd. The shell window should look similar to the following figure.



A shell window.

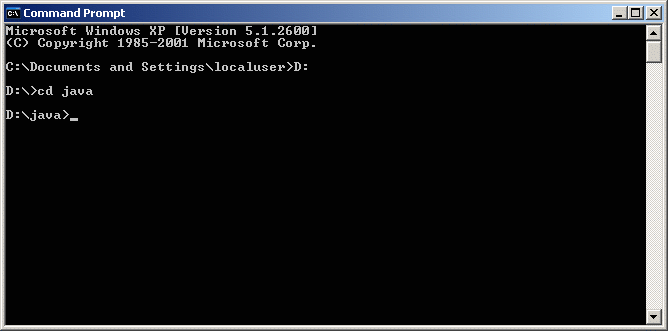
The prompt shows your *current directory*. When you bring up the prompt, your current directory is usually your home directory for Windows XP (as shown in the preceding figure.

To compile your source file, change your current directory to the directory where your file is located. For example, if your source directory is java on the C drive, type the following command at the prompt and press **Enter**:

cd C:\java

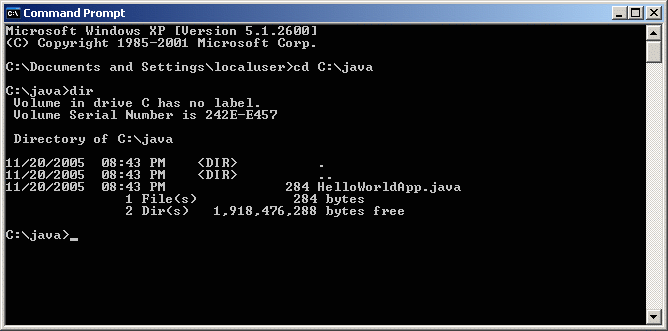
Now the prompt should change to C:\java>.

**Note:** To change to a directory on a different drive, you must type an extra command: the name of the drive. For example, to change to the java directory on the D drive, you must enter D:, as shown in the following figure.



Changing directory on an alternate drive.

If you enter dir at the prompt, you should see your source file, as the following figure shows.

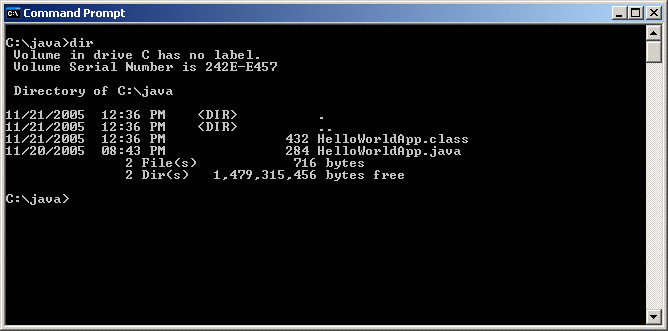


Directory listing showing the .java source file.

Now you are ready to compile. At the prompt, type the following command and press **Enter**.

javac HelloWorldApp.java

The compiler has generated a bytecode file, HelloWorldApp.class. At the prompt, type dir to see the new file that was generated, as shown in the following figure.



Directory listing, showing the generated .class file

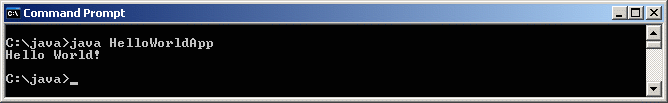
Now that you have a .class file, you can run your program.

### Run the Program

In the same directory, enter the following command at the prompt:

java HelloWorldApp

The next figure shows what you should now see:



The program prints "Hello World!" to the screen.

Congratulations! Your program works!

**Common Error Messages on Microsoft Windows Systems**

* 1. **'javac' is not recognized as an internal or external command, operable program or batch file**

If you receive this error, Windows cannot find the compiler (javac).

Here's one way to tell Windows where to find javac. Suppose you installed the JDK in C:\jdk1.7.0. At the prompt you would type the following command and press Enter:

C:\jdk1.7.0\bin\javac HelloWorldApp.java

If you choose this option, you'll have to precede your javac and java commands with C:\jdk1.7.0\bin\ each time you compile or run a program. To avoid this extra typing, consult the section **Updating the PATH** variable in the JDK 7 installation instructions.

* 1. **Class names, 'HelloWorldApp', are only accepted if annotation processing is explicitly requested**

If you receive this error, you forgot to include the .java suffix when compiling the program. Remember, the command is javac HelloWorldApp.java not javac HelloWorldApp.

## Runtime Problems

**Error Messages on Microsoft Windows Systems**

1. **Exception in thread "main" java.lang.NoClassDefFoundError: HelloWorldApp**

If you receive this error, java cannot find your bytecode file, HelloWorldApp.class.

One of the places java tries to find your .class file is your current directory. So if your .class file is in C:\java, you should change your current directory to that. To change your directory, type the following command at the prompt and press Enter:

cd c:\java

The prompt should change to C:\java>. If you enter dir at the prompt, you should see your .java and .class files. Now enter java HelloWorldApp again.

If you still have problems, you might have to change your CLASSPATH variable. To see if this is necessary, try clobbering the classpath with the following command.

set CLASSPATH=

Now enter java HelloWorldApp again. If the program works now, you'll have to change your CLASSPATH variable. The CLASSPATH variable is set in the same manner.

1. **Could not find or load main class HelloWorldApp.class**

A common mistake made by beginner programmers is to try and run the java launcher on the .class file that was created by the compiler. For example, you'll get this error if you try to run your program with java HelloWorldApp.class instead of java HelloWorldApp. Remember, the argument is the name of the class that you want to use, not the filename.

1. **Exception in thread "main" java.lang.NoSuchMethodError: main**

The Java VM requires that the class you execute with it have a main method at which to begin execution of your application.

**Creating Java Classes:**

1. A class in Java is just a blueprint telling what the objects created from it will look and act like.
2. Class is users define data types.
3. Class is a representation of similar kind of objects.
4. Class has these components:
5. Instance variables.
6. Static variables
7. Instance methods.
8. Static methods.
9. Constructors. ( For initialization and consistency)
10. Inner classes
11. Nested class
12. Instance initialize Block
13. Static initialize block
14. Class body is delineated by braces { }

**Syntax:**

<modifiers> class <name> <extends> <implements>

{ //constructors

// member variables

// member methods

// nested class

// block

}//end of class

**Static:**

1. These members are prefixed with static keyword.
2. These are associated with the class and class need not to be instanced to access these members.
3. These member are associated using either class name or object reference.
4. A static field is created when its class is loaded in memory.
5. A class is loaded when it is used first time during the execution of program.

**Instance:**

1. These members are associated with objects of the class and are accessible using object reference.
2. An instance field is created when an object of class is created.
3. Object is an imaginary boundary field of instance field.
4. Objects are created at runtime.
5. Class is created at compile time.

**Different ways of creating object in java:**

* **Using new keyword :-**

1. The new operator is used to create an object of class.
2. This is the most common way to create an object in java.

new <ClassName> (<parameters>)

1. The operator returns the reference of the object that we can store in the reference type variable.

Car c1= new Car();

Car c2= new Car();

\*\*Note: c1 & c2 are reference variables of Car type (class is users define data type) not are objects. Object is created in heap memory.

* **Using Class.forName():-**

1. If we know the name of the class & if it has a public default constructor we can create an object in this way. It is also known as reflection.
2. Car car = (Car) Class.forName (“com.abc. Car”).newInstance();

* **Using clone() :-**

1. The clone () can be used to create a copy of an existing object.

Car car1 = new Car ();

Car car = car1.clone ();

* **Using object deserialization :-**

1. Object deserialization is nothing but creating an object from its serialized form.

ObjectInputStream in = new OjectInputStream(anInputStream );

Car car = (Car) in.readObject();

* **Using reflection**:-(in another way)

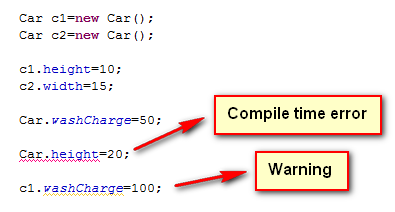
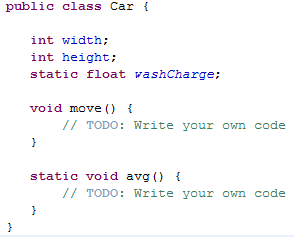
this.getClass ().getClassLoader ().loadClass (“com.abc.car”).newInstance ();

**Accessing Member:**

There are three ways to accessing members of a class:

1. <ObjectReference>.<member> (both instance & static member)
2. <ClassName>.<member> (only static member)
3. <member> (from within class )

Suppose we have a Car class with 2 instance member variable, 1 instance member method, 1 static member variable and 1 static member method.



**TOKENS:**

Lexical tokens are the basic building blocks of a source code.

Keywords (are defined in the compiler)

Identifiers (these are the name of variable, method, class etc. they must not begin with a digits and contain digits, letters underscore, currency symbols.)

Literals (values)

Integer (0b10, 10, 010, 0x10, 10L)

\*default data type is int

Floating points (3.0,-0.03, 10e-5, 20E+5)

\*default data type is double

Boolean (true, false)

Character (‘a’,’1’,’@’)

Escape Sequence

String (“abc”,”123”)

White Space (Tab,enter,space bar)

Operators

Comments

//Single line comment

/\* Multi line comment\*/

/\*\* Java documentation comment\*/

**ESCAPE SEQUENCE:**

|  |
| --- |
| \n 🡪 new line |
| \r 🡪 carriage return |
| \t🡪 tab |
| \b 🡪back slash |
| \f 🡪 form feed |
| \’ 🡪 |
| \” 🡪 |
| \\ 🡪 |

Data Types

Reference Types

Primitive Types

Class/Interface

Numeric Types

Arrays

Boolean Types

Enum

boolean(N/A)

**boolean**

double (8)

float (4)

byte(1)

int (4)

short (2)

Character Types

Integral Type

Floating point Type

long (8)

**Member variable vs. Local variable:**

1. Member variable are declared in class body. Local variables are declared in any block.
2. Local variables are not initialized by default.
3. A member variable is initialized by their default value.
4. It is an error to used uninitialized variable.

**Default values:**

|  |  |
| --- | --- |
| TYPES | VALUES |
| Boolean | false |
| Charactor | ‘\u0000’ |
| Integer | 0 |
| Floating point | 0.0D/0.0F |
| Reference | null |

**Java Coding CONVENSION (not a rule):**

1. **Name of class/interface :** My, MyFirst, MyFirstClass
2. **Name of variable/method :** my, myFirst, myFirstVar
3. **Name of constant/enum :** MY, MY\_FIRST,MYFIRST\_CONST

**The main() method:**

**public** **static** **void** main(String[] args) {

// All code goes here...

}

Called by JVM which is external entity, so it is public.

1. There is no need to make it object, hence it is static.
2. It is accessible by class name.

**Printing in Java:**

System.*out*.print(*data*);

data

System.*out*.println(*data*);

data + /n

* System is a class
* out is a static reference field in System class
* print() or println() are methods

1. When we pass a literals / primitive values in System.*out*.print(*data*) , it is converted to String and print it.
2. When we pass a reference values then toString() method of the Object is called to convert the reference value into String.

**Java source code structure:**

1. <Package declaration>
2. <import declaration>
3. <class and interfaces declaration>
4. All these are optional.
5. The name of the source code must be name of the public class/interface declared in it. Otherwise it can be anything.
6. To compile the source code we can use javac command. the syntax is :

javac <args> <Source fileName >

1. The compiler check the source code for error and generates a **.class** file for every class / interfaces declared in it.
2. These **.class** file contain the byte code of the corresponding class / interfaces.
3. To execute a java application we can use java command. the syntax is:

java <args> <Class Name having main> <args>

1. The command creates the JVM and passes everything that is given to it.
2. The JVM loads the specified class in memory and called its main method.

**Type Conversions:**

1. Java is a typed language.
2. Its checks for type compatibility at compile time as well as run time.
3. The syntax of type cast operator is:

(<type>) <expression>

1. Casting can be applied to primitive as well as reference values but not in between them.
2. The Boolean values can’t be cast to other types and vice versa.
3. The reference literals null can be cast to only reference types.

**Method:**

1. A method is a named sequence of code that can be invoked by other Java code.
2. A method takes some parameters, performs some computations and then optionally returns a value (or object).
3. Methods can be used as part of an expression statement.

**Method Declaration:**

<member modifier> <return-value-type><method-name> (<formal-parameter-list>)  
{  
 //declarations and statements  
}

1. The return type is mandatory.
2. The formal parameter are the list of variables local to the method body , with syntax:

<modifier> <type> <name>

1. The method body can contain statements, declarations.
2. The method signature is the method name and its formal parameters.
3. Java5 supports variables argument to a method , the syntax is:

----<name> (<type1><var1>, <type2><var2>, <type3> … <var3>) -----

1. Within the method body the values are accessed similar to arrays.

**Method Call:**

1. Objects communicate with each other by passing in form of method calls.
2. The syntax of method call is:
   * + <ClassName>.<methodName> (<actualParameters>)
     + <ObjRef>.<methodName> (<actualParameters>)
     + <methodName>(<actualParameters>)
3. The actual parameters are list of values separated with comma and must be type compatible with formal parameter.
4. In Java, method call is “CALL BY VALUE”.
5. Before passing the variables are created.

**Method Overloading:**

1. In a class when we have two or more methods with same name but different parameter list.This is called Method overloading.
2. The compiler can determine the method call based on parameter list, so it is called early, static, compile time binding.

**Example:**

class Test {

void main(){=}

void main(int l){=}

static void main(Car c){=}

static void main(Car c){=}

public static void main(String [] args){=}

}

\*\*) Exception: main method doesn’t found.

**Constructor:**

1. These are special member of a class used to construct the initial state of an object.
2. A constructor is a method with following properties:

* Same name as of the class.
* No return Type.
* Only access modifiers permitted.
* Declared in the same class.

1. A default constructor is without formal parameters.
2. We can declare parameterized constructer, so constructor can be overloaded.
3. When there is no constructor declared in a class the compiler creates a default constructor with the same accessibility of the class.
4. Constructors are called when an object is created using new operator.

**Arrays:**

1. In java arrays are objects.
2. Every array object has an instance field length that contains the size or array.
3. Elements of the array are its members.
4. Elements are accessed with using index starting with zero.
5. If we use an invalid index, we will get exception.
6. We can have zero length arrays.
7. An array is a list of similar things.
8. An array has a fixed:
   1. name
   2. type
   3. length
9. These must be declared when the array is created.
10. Arrays sizes cannot be changed during the execution of the code

**Declaring Array Reference variable:**

**Syntax:**

1. <type> <var> []
2. <type> [] <var>’

**Example:**

1. int [] a,b;
2. float x[],y;

**Constructing array object:**

**Syntax:**

(a) new < type> [<Size>]

(b) <var>={<values>}

**Example:**

int[] a= new int[5];

int [] b= {10, 20, 30};

**Declaring 2D-Array Reference Variable:**

**Syntax:**

* 1. <type> [][] <var>
  2. <type> <var> [][]
  3. <type>[]<var>[]

**Example:**

* 1. int [][] a , b;
  2. float [] x [] , y;
  3. Char ch [][] , j;

**Constructing 2D arrays objects:**

**Syntax:**

1. new <type> [<row>][<col>]
2. <var> ={ {<values1>},{<values2>},{<values3>} }

**Examples:**

arr=new int[4][];

arr[0]=new int [4];

arr[1]=new int [3];

arr[2]=new int [2];

**OR**

int [][] b = { {3,4,5} , {6,7} , { } , null };

**Command line arguments:**

1. When we pass arguments to the java command after the class name, it constructs a String array of those arguments and passes to the main method.
2. When no arguments are passes, it creates a zero length array.

**Anonymous Array:**

1. It contains the syntax of array creation and initialize expression:

new <type>[]{<values>}

new <type>[][]{<values>}

**Loops:**

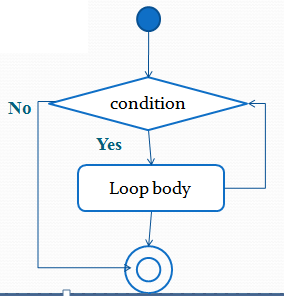
1. A loop allows the programmer to specify that a program should repeat an action while some condition remains true.
2. Java loops come in three flavors:

* while
* do-while
* for

1. The for loop has two variations:

* for
* for-each

1. **While Loop:**

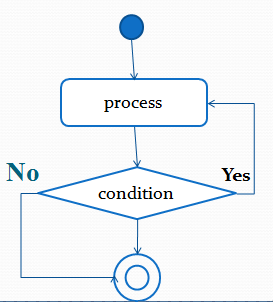
* The while loop is good for scenarios where you don't know how many times a block or statement should repeat, but you want to continue looping as long as some condition is true.
* A while statement looks like this:

**while** (expression) {

// do stuff

}

1. **do-while Loop:**

* The do...while repetition statement is similar to the while statement , except that the expression is not evaluated until the do loop's code is executed.
* Therefore the code in a do loop is guaranteed to execute at least once.
* The following shows a do loop :

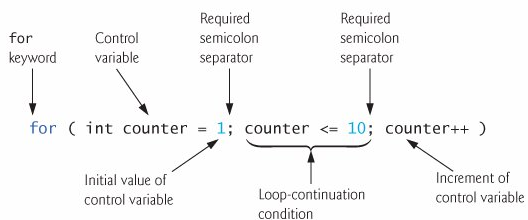
**do** {

// do stuff

} **while** (expression);

1. **For Loop:**

* The for loop is especially useful for flow control when you already know how many times you need to execute the statements in the loop's block.
* The for loop declaration has three main parts, besides the body of the loop:
  1. Declaration and initialization of variables
  2. The boolean expression (conditional test)
  3. The iteration expression.

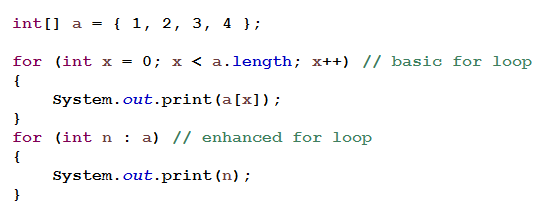


**For-each Loop**

* It is a specialized for loop that simplifies looping through an array or a collection.
* Also known as: “for-in “and "enhanced for" loop.
* For-each is declared as:

**for (<type> <var> : <expression>)**

* Difference between basic for loop & enhanced for loop:



**Class/interface Access Modifiers:**

1. **public**

A type with public modifier is accessible in any package.

1. **default**

A type with no access modifier is accessible in its own package.

(\*) Other class modifiers

1. **abstract**

A class can be declared abstract to restrict its instantiation.

1. **final**

If a class is complete, we can use final modifier with it. Such a class can’t be inherited.

**Packages:**

1. It is an encapsulation mechanism to group related classes, interfaces and sub-packages.
2. The packages members are accessed using”.” operator.
3. The packages hierarchy represents the byte code organization not a source code organization.
4. The class/interface can indicate that its byte code is placed in a package by having package declaration in its source code:

package < fullyQualifiedName >;

1. The package name is saved in the byte code of the class and interfaces contained it.
2. If the source code omits a package declaration the classes and interfaces declared in it belong to **unnamed** (default) package.
3. To access a type declared in another package, we can use either:
4. Fully qualified type name.
5. Import declaration:
   * import <fullyQualifiedPackageName>.\*;
   * import <fullyQualifiedTypeName>;
6. When using first import we can access any public type declared in the package using its name and when using second only the specified type.
7. The import declaration does not recursively imports sub packages and also does not results in inclusion of the source code of the types. It only makes the types name available.
8. All compilation units implicitly import **java.lang** package.
9. If there is any name conflict due to multiple imports. It can be resolved using either **fully qualified type name** or **fully qualified type import**.
10. When we compile a source code and use **–d** option the compiler implicitly creates the directory structure corresponding to the package declaration.
11. To execute main method of a packaged class we have to use its fully qualified name.
12. To access static member of a class we can use static import:

* import static <fullyQualifiedTypeName>.<member>
* import static <fullyQualifiedTypeName>.\*;

**Member Access Modifiers:**

1. **public**

A member with public modifier is accessible in any package.

1. **Protected**

A member with protected modifier is accessible in its own package.

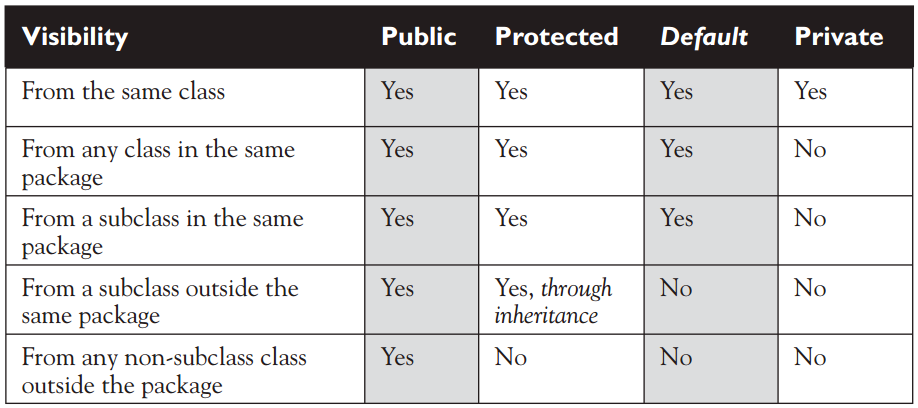
When its class is inherited, these become the part of the subclass.

1. **Default**

A type with no access modifier is accessible in its own package.

1. **Private**

A member with private modifier is accessible only within its class.



**Other Member Modifiers:**

1. **static**

Associated with class See above

1. **final**

* This modifier can also be applied to local variables.
* A final variable in java is blank final i.e. it is need not to be initialized when created, but can be initialized only once.
* A final method cannot be overridden.
* If a class is complete, we can use final modifier with it. Such a class can’t be inherited.

1. **Abstract**

* A method having no method body is declared abstract.
* A class having one or more abstract methods must be declared abstract.
* The subclass of such a class must override (define) the abstract methods from its super-class otherwise it also becomes abstract.
* We cannot use final and private access modifiers with abstract.

1. **Synchronized**

When it is required that a method is access by one thread at a time it is declared synchronized.

1. **Native**

* The JNI (java native interface) API can be used to access a method implemented in C language.
* Such a method is declared native in our class.

1. **Transient**

* Objects can be serialized.
* The transient members are not saved during serialization.

1. **Volatile**

* The compiler allows of fields for efficiency reasons. This may result in inconsistency in a multithreaded environment.
* The volatile modifier can be used to inform the compiler that the catching is not allowed

**Exception Handling**

There are three categories of errors:

**Syntax errors:**

These are arising because the rules of the language have not been followed. They are detected by the compiler.

**Runtime errors:**

These are occurring while the program is running if the environment detects an operation that is impossible to carry out.

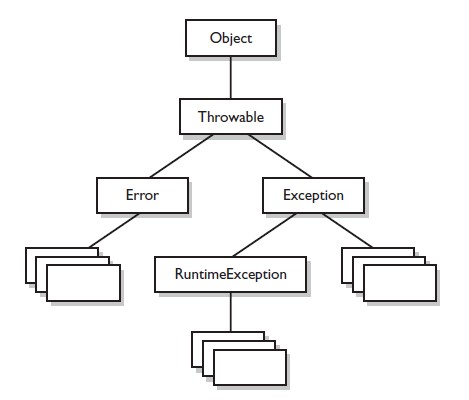
**Logic errors:**

These are occurring when a program doesn't perform the way it was intended to.

1. Except ions are unexpected conditions at run time.
2. Exception mechanism is based on throw and catch.
3. To throw an exception is indicate something unexpected.
4. To catch an exception is to take appropriate action to deal with it.
5. Exceptions in java are objects, derived from Throwable.
6. An unhandled exception propagates in the method activation stack and is finally handled by the default exception handler (part of JVM).
7. The default exception handler prints the exception message, the thread name, and the stack trace.
8. The JVM terminates the threads with unhandled exception.
9. The Throwable class provides a String field that can be set by its subclass for detailed message.
10. The class provides methods:

* String getMessage();
* void printStackTrace()
* Void printStackTrace(PrintWriter)

**Exception Hierarchy:**



**try-catch-finally:**

**try**

{

// statements that can result is exception

}

**catch**(<ExceptionType> <var> )

{

// similar to a method and declares a parameter of the exception //type that it wants to handle.

//handles exceptions that occurs in its try blocks.

}

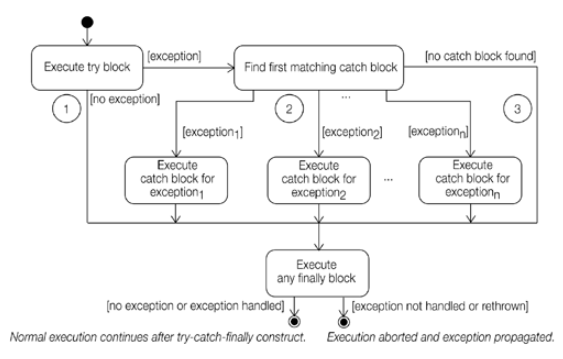
**finally**

{

// some cleanup code

}

1. The order must be same.
2. A try can have zero or more catch and zero or more finally.
3. A try without catch/finally and a catch/finally without try is an error.
4. The block notation is mandatory.
5. The try block encloses statements that can result in exception.
6. A catch block is similar to the method declare a parameter of the exception type that it wants to handle.
7. A catch blocks handles exception that occurs in its try block.
8. A catch block for super-type exception must not shadow a catch block for subtype exception.
9. The finally block is always executed so it can contain any clean-up code.
10. An exception in finally block over rules any previously unhandled exception.



**throws clause:**

Exception can categories in two category:

**Unchecked Exceptions:**

1. Object of Runtime Exception, Error and their subclasses are known as *unchecked* *exceptions*.
2. The method that can have these exceptions is not obliged to handle them.
3. These are either programming errors or irrecoverable.
4. A **NullPointerException** is thrown if you access an object through a reference variable before an object is assigned to it.
5. An **IndexOutOfBoundsException** is thrown if you access an element in an array outside the bounds of the array.
6. These are the logic errors that should be corrected in the program. Unchecked exceptions can occur anywhere in the program. To avoid cumbersome overuse of try-catch blocks, Java does not mandate you to write code to catch unchecked exceptions.

**Checked Exceptions:**

1. All other exceptions are known as *checked exceptions*, meaning that the compiler forces the programmer to check and deal with the exceptions.
2. If these can occurs in a method. The method must have one of the following actions to deal with them:
3. try construct
4. throws clause
5. By declaring an exception in its throws clause a method indicate that specifies or its sub class can occur in the method and it is handled by its caller.

**Syntax:**

---------<methodName> (---) **throws**<ExceptionType1> <ExceptionType1> {

//body

}

**Creating Custom Exception Classes:**

1. We can create our own exception type by inheriting Exception
2. Use the exception classes in the API whenever possible.
3. Create custom exception classes if the predefined classes are not sufficient.

**Example:**

**public** **class** MyException **extends** Exception {

MyException (String reason) {

**super** (reason);

}

}

**Throwing Exceptions:**

1. When the program detects an error, the program can create an instance of an appropriate exception type and throw it. This is known as *throwing an exception*.

throw <expression>;

* expression must be throwable type

1. Here is an example:

**throw** **new** MyException ();

Or

MyException ex = **new** MyException ();

**throw** ex;

**Cautions When Using Exceptions:**

Exception handling separates error-handling code from normal programming tasks, thus making programs easier to read and to modify. Be aware, however, that exception handling usually requires more time and resources because it requires instantiating a new exception object, rolling back the call stack, and propagating the errors to the calling methods.

**When to Throw Exceptions:**

An exception occurs in a method. If we want the exception to be processed by its caller, we should create an exception object and throw it. If we can handle the exception in the method where it occurs, there is no need to throw it.

**Object Oriented Programming**

**Inheritance:**

1. It is a way of code reusability.
2. Central to Java and other object-oriented (OO) languages is the concept of inheritance, which allows code defined in one class to be reused in other classes.
3. In Java, we can define a general (more abstract) super-class, and then extend it with more specific subclasses. The super-class knows nothing of the classes that inherit from it, but all of the subclasses that inherit from the super-class must explicitly declare the inheritance relationship.
4. A subclass that inherits from a super-class is automatically given accessible instance variables and methods defined by the super-class, but are also free to override super-class methods to define more specific behavior.
5. A subtype inherits members from its super-type.
6. It can declare new members and it required can change inherited behavior.
7. Java supports **single inheritance** using classes.
8. The subclass uses extends keyword to specifies its super class
9. If a class declaration does not contains extends keyword. It implicitly inherits **java.lang.Object**.

**Method Overriding:**

1. A subclass can redefine a method inherited from its super-class. The rules:
2. The name and parameter list must be same.
3. The return type must be same or it can be a subtype.
4. The new method definition cannot narrow the accessibility of the method, but it can widen it.
5. The overriding method declares a subset of exception classes (including sub classes) that are declared in throws clause of overridden method.
6. Static method cannot override an instance method and the reverse.
7. A super-type variable can contain the reference of subtype object.
8. The call of the static method is resolved by the type of the object reference value.
9. The call of instance method is resolved by the type of object. Here the actual method lookup procedure is:
10. If the object type contains a method i.e. resolved and the resolved method is accessible from the type then it is invoked
11. Otherwise the same procedure is performed in its super-classes.
12. At last if method is not found we will get AbstractMethodError.

**Field Hiding:**

1. A subtype can re-declare a field inherited from its super-class.
2. It is the type of reference value/variable that determines the field access.
3. Class A

{

**int** i ;

**void** f1(){

//body

}

}

1. **class** B **extends** A

{

**float** i;

**void** f1(){

//body

}

**void** f2(){

//body

}

}

1. **class** C **extends** B

{ String i;

**void** f1(){ //body }

**void** f3(**char** i)

{

System.*out*.println(i);

System.*out*.println(**this**.i);

System.*out*.println(**super**.i);

f1();

**super**.f1(); }

}

**Object Reference ‘this’:**

1. When an instance method is called the reference of the object is implicitly passed in the method.
2. With the method body the reference is accessible using ‘this’.
3. In the method body when a variable is accessed it is first searched locally, if not found, the reference ‘this’ is apply.
4. When a method is call with class name/reference, the reference ‘this’ is applied.
5. Static methods does not have ‘this’. Here the class name is applied.

**Object Reference ‘super’:**

1. In an instance method of the class the super contains the reference of its immediate super class that is associated with this reference of the sub-class.
2. The super cannot be use as ordinary reference. It can only be use to access members.

**Constructor chaining using “this ()”:**

1. The call can be use in a constructor to invoke another constructor of the same class.
2. The call must be within the constructor and the first statement.

**public** **class** A

{

**int** i , j;

A ()

{

//i=10;

// j=20;

**this** (10);

}

A(**int** i)

{

//this.i;

//j=20;

**this** (i,20);

}

A (**int** c,**int** d)

{

i=c;

j=d;

}

}

**Use:**

A v1 = **new** A();

A v2 = **new** A(3);

A v3 = **new** A(2, 3);

**Constructor chaining using “super ()”:**

1. The call can be use in the constructor to invoke the constructor of its immediate super-class.
2. This facilitates the initialization of the super class object, when sub-class is instantiated
3. The call must be within the constructor and the first statement.
4. If there is no this ()/super () call in a constructor, the compiler implicitly insert a super () in the constructor.

class B extends A

{ int p;

B()

{

p=50;

} B(int q)

{ super(q\*2);

}

}

public class A

{ int r;

void f1()

{

//body

}

A (int q) {

r=q;

}

}

**Interface:**

1. Interface can be used to declare a new define type.
2. They provide multiple inheritances.
3. The syntax is:

<access modifier> interface <name> <extends>

{

//body

}

1. The interface body contains:
2. **Methods:**

Are implicitly public, abstract

1. **Fields:**

Are implicitly public, static, final

1. **Nested Type:**

The interface is implicitly abstract

1. An interface can inherit one or more interfaces using extends keyword.
2. If an interface declaration is not using extends it implicitly provides the abstract declaration of all the methods of **Object** class.
3. A class can implement more than one interface using implements keyword.
4. Interface does not contain initialize blocks.
5. Here the class has to define (override) the methods from the interfaces otherwise the class becomes abstract.
6. It is guaranteed to make subclass of interface so, interface are always super-types.

**Reference Assignment:**

**<destType> = <srcType>**

1. If a srcType is class type , then the type of destType can be:
2. A super-class
3. An implemented interface
4. If a srcType is interface type , then the type of destType can be:
5. Object
6. A super interface
7. If a srcType is primitive array then the type of destType can be:
8. Object
9. Serializable/Clonable interface
10. If a srcType is reference type array then the type of destType can be:
11. Object
12. Serializable/Clonable interface
13. An array type where the element type of the srcType is assignable to the element type of the destType.

**Reference Casting:**

**<srcType> t1;**

**<destType> t2;**

**t2= (<destType>) t1;**

**<objRef> instanceof(<destType>)**

**Compile time:**

1. It is checked if a reference variable of srcType and a variable of destType can both refer to an object of a common class.

**Runtime check:**

1. The values are known, so it is checked if the value is of destType or not.
2. If the runtime checked is passed, the instanceof operator returns true and the casting is valid.
3. Otherwise the instanceof operator returns false and casting results in ClassCastException.

**Types of Classes:**

**Outer Class**

**Local Class**

**Anonymous Class**

**Nested Static Class**

**Inner Class**

1. The Java programming language allows you to define a class within another class. Such a class is called a *nested class*.

class OuterClass

{

...

class NestedClass

{ ... }

}

1. Nested classes are divided into two categories:

static and non-static.

a. Nested classes that are declared static are simply called ***static nested classes***.

b. Non-static nested classes are called ***inner classes***.

1. An interface declared in a class is implicitly static.
2. A class declared in an interface is implicitly public, static.

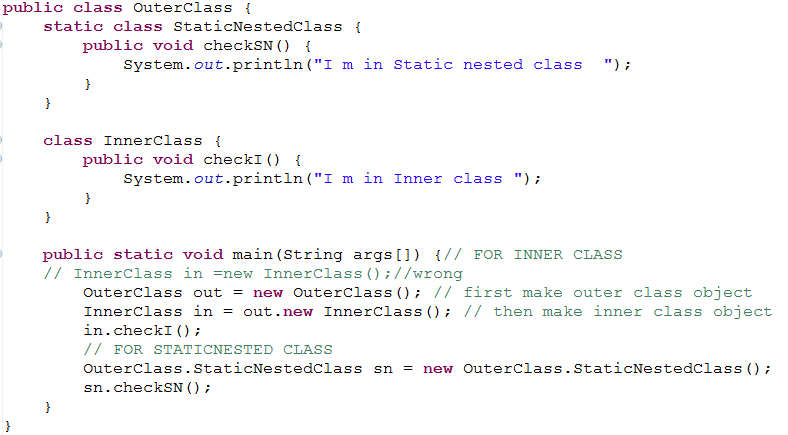
**Inner Class:**

1. An inner class is associated with an instance of its enclosing class and has direct access to that object's methods and fields.
2. Also, because an inner class is associated with an instance, it cannot define any static members itself.
3. Objects that are instances of an inner class exist within an instance of the outer class.
4. Except the inner class, there are two types of supplementary  inner classes :
5. The inner class which is declared inside the body of a method is known as the local inner classes.
6. The class declared inside the body of a method without naming it is known as anonymous inner classes.

**Static Nested Class:**

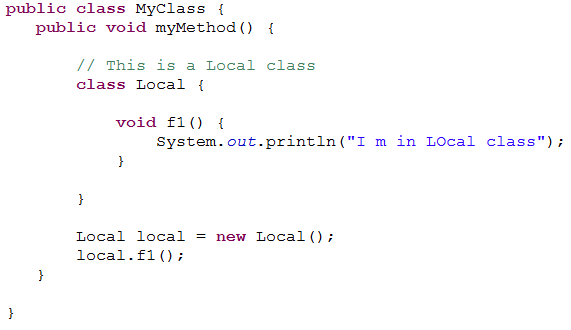
1. A nested class that is declared static is called a static nested class.
2. Memory to the objects of any static nested classes is allocated independently of any particular outer class object.
3. A static nested class uses the instance variables or methods defined in its enclosing class only through an object reference.
4. A static nested class interacts with the instance members of its outer class or any other class just like a top-level class.

**Example:**

****

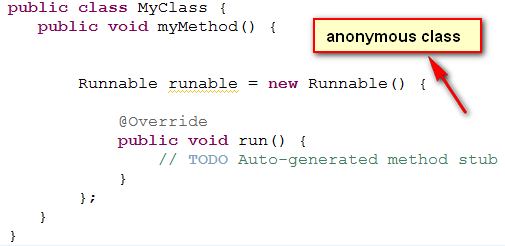
**Local class:**

1. A local class is declared locally within a block of Java code, rather than as a member of a class.
2. Typically, a local class is defined within a method, but it can also be defined within a static initiailzer or instance initializer of a class.

****

**Anonymous class:**

1. An *anonymous class* is a local class without a name. An anonymous class is defined and instantiated in a single succinct expression using the new operator.
2. Anonymous class definition is an expression, which means that it can be included as part of a larger expression, such as a method call.
3. When a local class is used only once, consider using anonymous class syntax, which places the definition and use of the class in exactly the same place.
4. The *anonymous inner classes* are very useful in some situation. For example consider a situation where we need to create the instance of an object without creating subclass of a class and also performing additional tasks such as method overloading.



**Garbage Collector:**

1. The garbage collector is the part of the JVM.
2. The JVM invokes it when there is lack of memory (heap) for new objects.
3. The garbage collector identifies the objects that are no longer accessible and reclaims the area occupied by them.
4. Garbage collection relieves java programmer from **memory management** which is essential part of C++ programming and gives more time to focus on business logic.
5. **Garbage Collection in Java** is carried by a daemon thread called ***Garbage Collector***.
6. Before removing an object from memory **Garbage collection thread invokes finalize () method** of that object and gives an opportunity to perform any sort of cleanup required.
7. We as Java programmer **cannot force Garbage collection in Java**; it will only **trigger** if JVM thinks it needs a garbage collection **based on Java heap size.**
8. There are methods like System.gc **()** and Runtime.gc () which **is used to send request of Garbage collection to JVM** but it’s *not guaranteed that garbage collection will happen*.
9. If there is no memory space for creating new object in Heap **Java Virtual Machine** throws **OutOfMemoryError** or **java.lang.OutOfMemoryError heap space.**

**When an Object becomes Eligible for Garbage Collection:**

1. An Object becomes eligible for Garbage collection or GC if it’s not reachable from any live threads or any static references.
2. An object becomes eligible for garbage collections if it’s all references are *null*.
3. Cyclic dependencies are not counted as reference so if Object A has reference of object B and object B has reference of Object A and they don't have any other live reference then both Objects A and B will be *eligible for Garbage collectio*n.
4. Generally an object becomes *eligible for garbage collection in Java* on following cases:
5. All references of that object explicitly set to null e.g. object = null
6. Object is created inside a block and reference goes out scope once control exit that block.
7. Parent object set to null, if an object holds reference of another object and when we set container object's reference null, child or contained object automatically becomes eligible for garbage collection.
8. If there are two references of that object pointing different objects of same type and we assign one’s reference value to another , then second object is becomes eligible for garbage collection.

**Object Finalization:**

1. Objects can have resources associated with them.
2. It is their responsibility to free the resources.
3. The finalize (), is declared in Object class and is call by garbage collector once, just before destroying the object.
4. An object can take any last action using this method just before its area is reclaimed by the garbage collector.
5. The finalize () method can also be used to make an object accessible again.
6. The finalizer in the sub-class should call the finalizer of its super-class as its last action, because unlike constructor finalizer are not chained.
7. The Object finalize method performs no actions but it may be overridden by any class
8. Normally it should be overridden to clean-up non-Java resources i.e. .closing a file
9. If overriding finalize () it is good programming practice to use a try-catch-finally statement and to always call super.finalize ().
10. This is a safety measure to ensure you do not inadvertently miss closing a resource used by the objects calling class

**protected** **void** finalize() **throws** Throwable

{

**super**.finalize();

// Do your own work

}

1. Any exception thrown by finalize () during garbage collection halts the finalization but is otherwise ignored.
2. finalize () is never run more than once on any object.

**Initializers:**

1. These are used to initialize fields of a class /object.
2. These can be initializer expression or initialize blocks in class body.
3. Advantages of a constructor:

* Different values are passes
* Constructors are similar to methods, so we can use try catch, if else etc.

1. If an exception can occurs in an instance initialize block, it must be handled using either:

* try-construct
* Declaring it in throws clause of every constructor of the class.

**Constructing initial object state:**

1. When an object is created using new operator:
2. All instance fields are created and initialized with their default values.
3. The constructor is invoked.
4. This may results in local chaining of constructors finally with a super () call.
5. When the control returns from the super class:
   * 1. All instance initialize expression and blocks are executed in specified order
     2. The constructor’s execution resumes.
6. In general final fields are declared static to create constants.
7. If there is a final instance field it must be initialized only once during the control flow as specify above.

**Class/interface Initialization:**

1. These initialized when loaded in memory. It involves execution of static initiailzer expression and blocks in specified order.
2. Interface does not contained initiailzer blocks.
3. The super type is loaded and initialized before the sub type.

**Fundamental Classes**

The java.lang package provides few fundamental classes:

1. **java.lang.Object:**
2. The Object class sits at the top of the class hierarchy tree in the Java development environment.
3. Every class in the Java system is a descendent (direct or indirect) of the Object class.
4. The Object class defines the basic state and behavior that all objects must have, such as the ability to compare one to another object, to convert to a string, to wait on a condition variable, to notify other objects that a condition variable has changed, and to return the object's class.
5. **boolean equals(Object):-**
6. Use the equals to compare two objects for equality. This method returns true if the objects are equal, false otherwise. Note that equality does not mean that the objects are the same object.

Integer one = **new** Integer(1);

Integer anotherOne = **new** Integer(1);

**if** (one.equals(anotherOne)) {

System.*out*.println("objects are equal");

}

1. **final Class getClass():-**
2. The getClass method is a final method (cannot be overridden) that returns a runtime representation of the class of this object. This method returns a Class object.

**void** printClassName(Object obj) {

System.*out*.println("The Object's class is " + obj.getClass().getName());

}

1. **String toString():-**
2. Object’s toString () method returns a String representation of the object. We can use toString () to display an object.
3. For example, we could display a String representation of the current Thread like this:

System.*out*.println(Thread.*currentThread*().toString());

1. The String representation for an object is entirely dependent on the object. The String representation of an Integer object is the integer value displayed as text.
2. The String representation of a Thread object contains various attributes about the thread, such as its name and priority. **Output:** Thread[main,5,main]
3. **int hashCode():-**
4. This method returns unique id for an object. It is used in collections.
5. **protected Object clone() throws CloneNotSupportedException:-**
6. clone () is a method in the Java programming language for object duplication.
7. In Java, objects are manipulated through reference variables, and there is no operator for *copying* an object—the assignment operator duplicates the reference, not the object.
8. The clone () method provides this missing functionality.
9. The object to be clone must implement **Clonable** interface otherwise will get exception.
10. **Wrapper classes:**
11. Corresponding to every primitive type, there is a class that represents the primitive values as an object.
12. These are:

Byte , Short , Character , Integer , Float , Long , Double , Boolean , Void.

1. The Void is non instance able and represents the return type void.
2. These classes (except character) provide two constructors to create object.
3. Since java-5, the primitive values are implicitly converted to the wrapper types and the reverse .This is called auto-boxing and un-boxing.
4. The numeric wrapper classes provide constant:

* MIN\_VALUE
* MAX\_VALUE

to represent the corresponding primitive type.

1. The numeric wrapper classes provide static method parseX() to convert the string values in corresponding primitive type.
2. These methods can throw-NumberFormatException.

**Example:** Integer i1 = **new** Integer(10);

Integer i2 = **new** Integer("10");

Integer i3 = 10; // autoboxing

**int** k = i3; // unboxing

**Use of parseX():**

**int** j= Integer.*parseInt*("40");

**double** d= Double.*parseDouble*("2.3");

**int** risky=Integer.*parseInt*("Abs");//NumberFormatException

**Use of constant:**

System.*out*.println(Integer.*MIN\_VALUE*);//-2147483648

System.*out*.println(Integer.*MAX\_VALUE*);//2147483647

1. **String Class:**
2. An object of String class represents sequence of characters.
3. The object is non-changeable/immutable.
4. The String literals are representing using object of String class.
5. Two literal with same character sequence are represented by same object.
6. A simple String can be created using a string literal enclosed inside double quotes as shown;

String str1 = "I Love JAVA";

1. Since a string literal is a reference, it can be manipulated like any other String reference. The reference value of a string literal can be assigned to another String reference.
2. If two or more Strings have the same set of characters in the same sequence then they share the same reference in memory. Below illustrates this phenomenon.

String str1 = "I Love JAVA";

String str2 = "I Love JAVA";

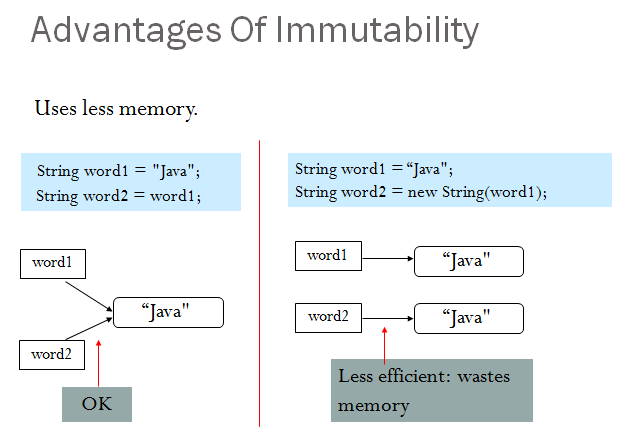
String str3 = "I Love " + "JAVA";// Compile time expression

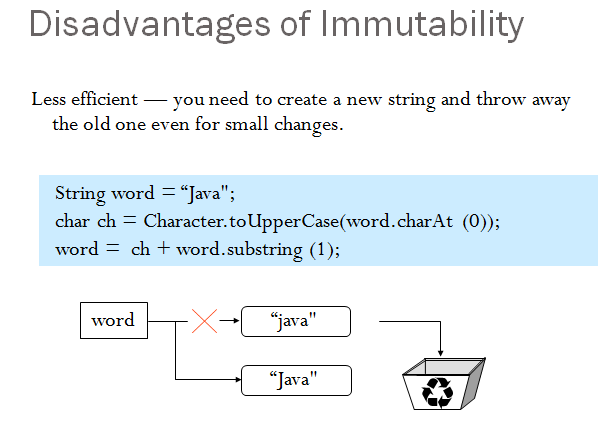
String name = "Shadab";

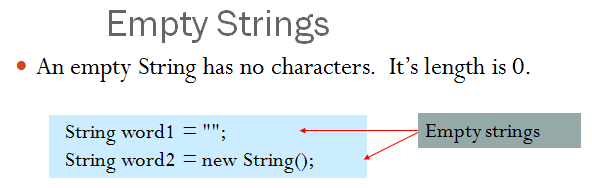
String str4 = "My name is" + name;

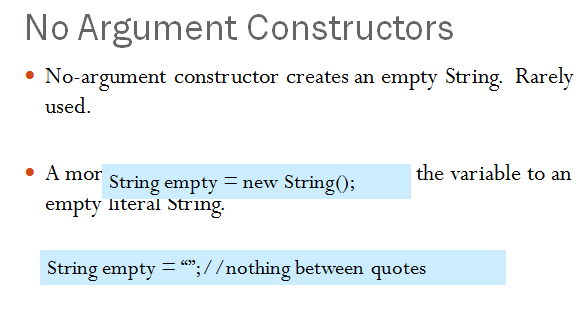
String str5 = **new** String ("My name is Shadab");

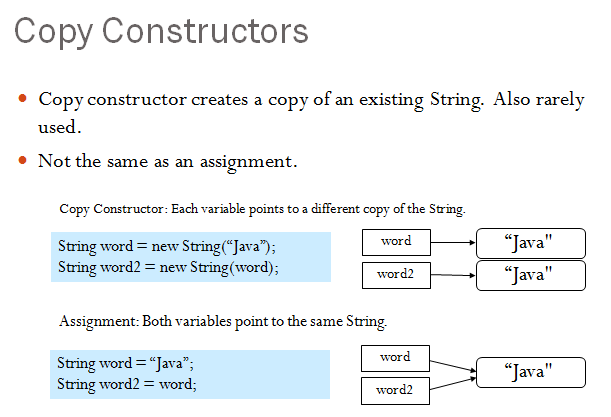
1. In the above code all the String references str1, str2 and str3 denote the same String object, initialized with the character string: “My name is Shadab”. But the Strings str4 and str5 denote new String objects.

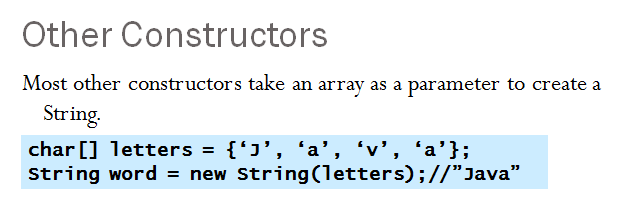


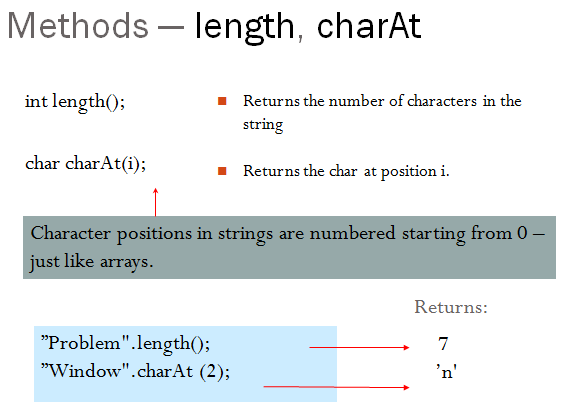


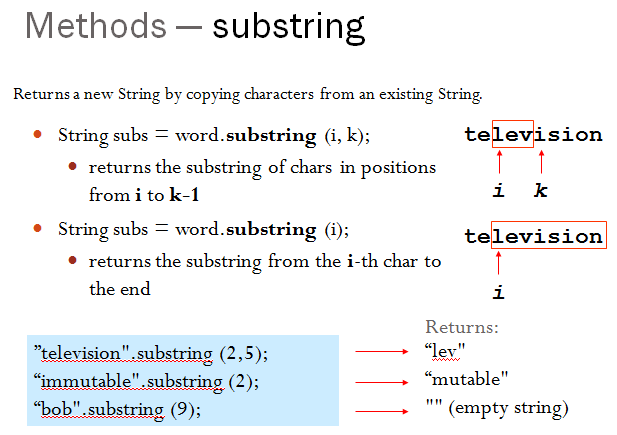


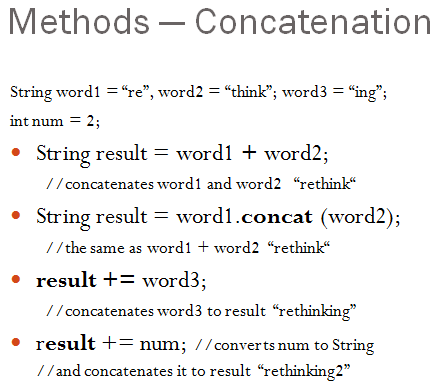


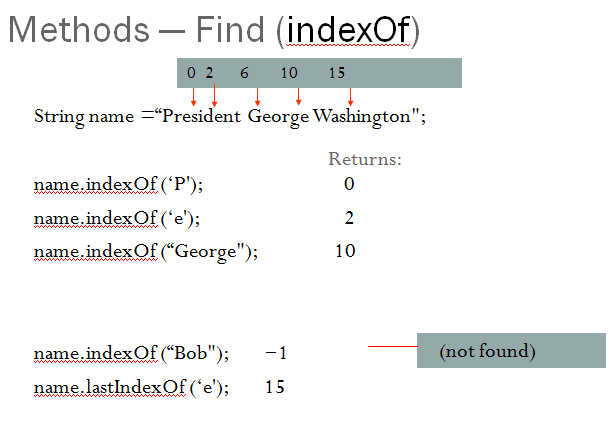


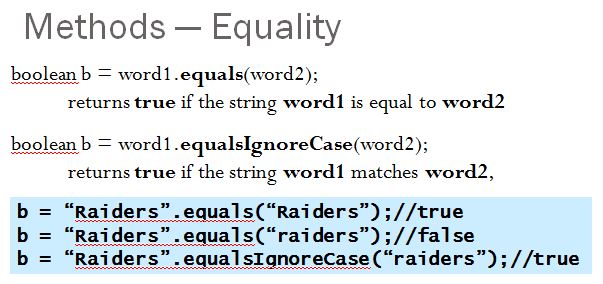


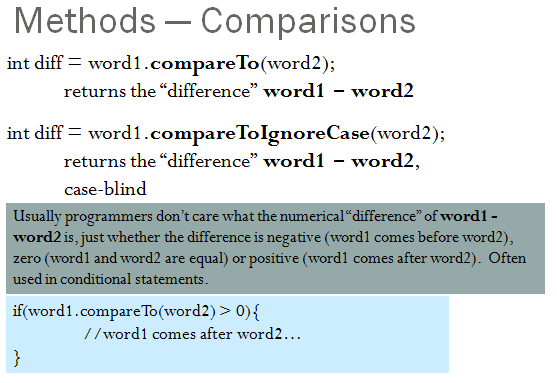


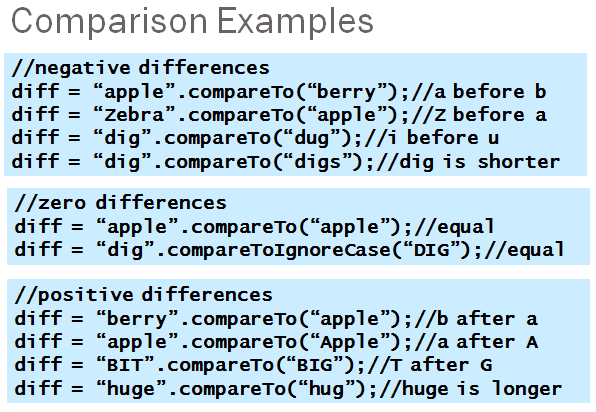


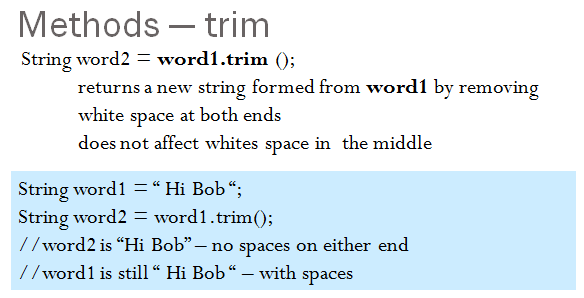


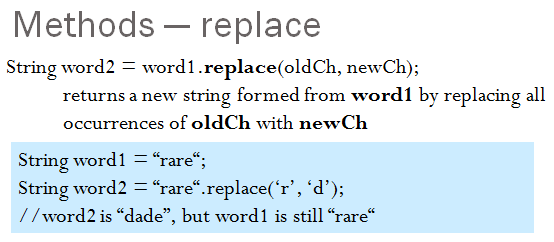


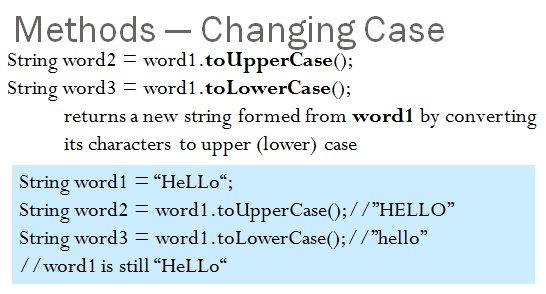












**Difference between String, StringBuffer and StringBuilder:**

**An object of StrinBuffer represents the sequence of character.**

**The object is changeable.**

**The StringBuffer is thread safe and provides methods similar to StringBuilder**

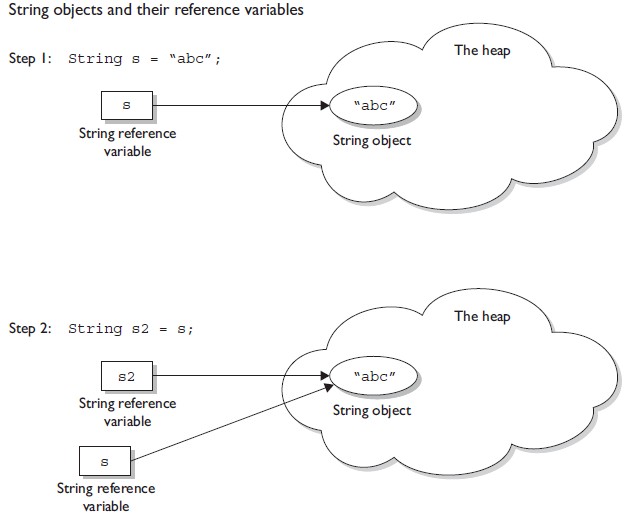
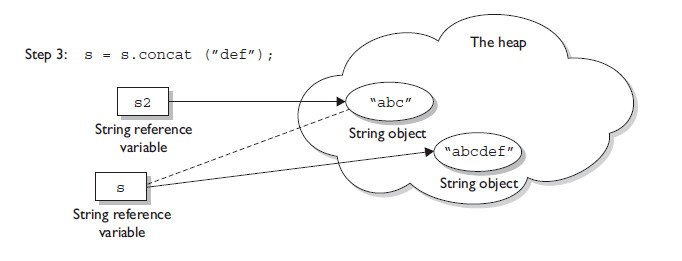
1. Well, the most important difference between String and StringBuffer/StringBuilder in java is that String object is immutable whereas StringBuffer/StringBuilder objects are mutable.
2. By immutable, we mean that the value stored in the String object cannot be changed.
3. Then the next question that comes to our mind is “If String is immutable then how is I able to change the contents of the object whenever I wish to?” Well, to be precise it’s not the same String object that reflects the changes you do. Internally a new String object is created to do the changes.
4. So suppose we declare a String object:

**Step1:** String s = "abc";

**Step2:** String s2 = s;

1. Next, we want to append “Guest” to the same String. What do you do?

**Step3:** s=s+"def";

****

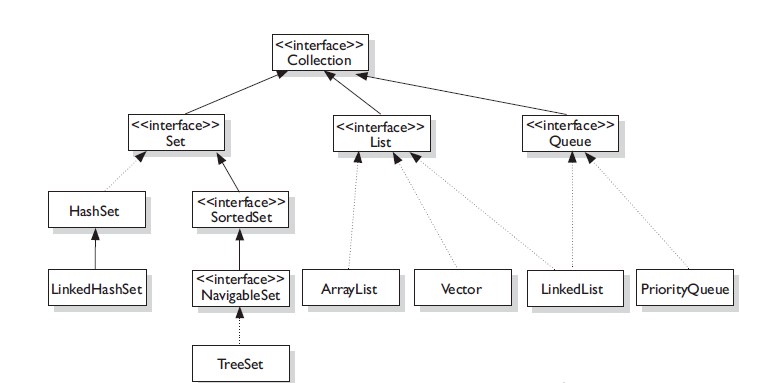
**Now isn’t that a performance issue?**

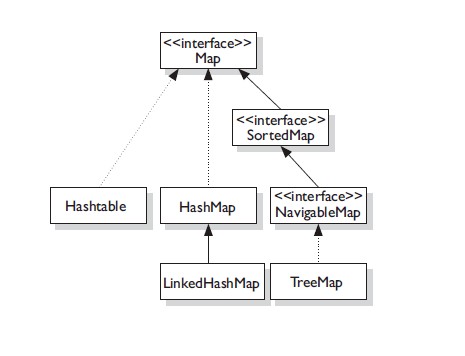
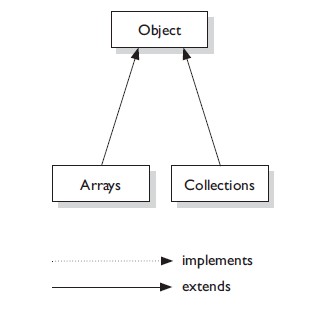
1. Yes, it definitely is.
2. Then how do we make our string operations efficient?
3. By using StringBuffer or StringBuilder.
4. How would that help? Well, since StringBuffer/StringBuilder objects are mutable, we can make changes to the value stored in the object. What this effectively means is that string operations such as append would be more efficient if performed using StringBuffer/StringBuilder objects than String objects.
5. Finally, what’s the difference between StringBuffer and StringBuilder?
6. StringBuffer and StringBuilder have the same methods with one difference and that’s of synchronization.
7. StringBuffer is synchronized (thread safe) whereas StringBuilder is not synchronized (it isn’t thread safe).
8. So, if we aren’t going to use threading then use the StringBuilder class as it’ll be more efficient than StringBuffer due to the absence of synchronization.

**Collection Framework**

* 1. The framework is provided by java.util package.
  2. The Java language API provides many of the data structures from this class for you.
  3. It defines a “collection” as “an object that represents a group of objects”.
  4. It defines a collections framework as “a unified architecture for representing and manipulating collections, allowing them to be manipulated independent of the details of their representation.”
  5. The core collection interfaces encapsulate different types of collections. They represent the abstract data types that are part of the collections framework. They are interfaces so they do not provide an implementation!
  6. Three meanings for "collection":
* **collection (lowercase c):** Represents the data structure in which objects are stored
* **Collection(java.util.Collection):**It is an interface from which Set and List extend
* **Collections(java.util.Collections):** A class that holds static collection utility methods

1. **ArrayList**: Fast iteration and fast random access.
2. Vector: It's like a slower ArrayList, but it has synchronized methods.
3. **LinkedList:** Good for adding elements to the ends, i.e., stacks and queues.
4. **HashSet:** Fast access, assures no duplicates, and provides no ordering.
5. **LinkedHashSet:** No duplicates; iterates by insertion order.
6. **TreeSet:** No duplicates; iterates in sorted order.
7. **HashMap:** Fastest updates (key/values); allows one null key, many null values.
8. **Hashtable:** Like a slower HashMap (as with Vector, due to its synchronized methods). No null values or null keys allowed.
9. **LinkedHashMap:** Faster iterations; iterates by insertion order or last accessed; allows one null key, many null values.
10. **TreeMap:** A sorted map.
11. **PriorityQueue:** A to-do list ordered by the elements' priority.





**public interface Collection<E> extends Iterable<E>:**

The root of the collection hierarchy. A collection represents a group of objects known as its *elements*. The Collection interface is the least common denominator that all collections implement and is used to pass collections around and to manipulate them when maximum generality is desired. Some types of collections allow duplicate elements, and others do not. Some are ordered and others are unordered. The Java platform doesn't provide any direct implementations of this interface but provides implementations of more specific sub interfaces, such as Set and List.

**public** **interface** Collection<E> **extends** Iterable<E> {

// Basic operations

**int** size();

**boolean** isEmpty();

**boolean** contains(Object element);

**boolean** add(E element); //optional

**boolean** remove(Object element); //optional

Iterator<E> iterator();

// Bulk operations

**boolean** containsAll(Collection<?> c);

**boolean** addAll(Collection<? **extends** E> c); //union

**boolean** removeAll(Collection<?> c);//minus

**boolean** retainAll(Collection<?> c); //intersection

**void** clear();

// Array operations

Object[] toArray();

<T> T[] toArray(T[] a);

}

**A note on iterator:**

An Iterator is an object that enables you to traverse through a collection and to remove elements from the collection selectively, if desired. You get an Iterator for a collection by calling its iterator () method. The following is the Iterator interface.

**public** **interface** Iterator<E> {

**boolean** hasNext();

E next();

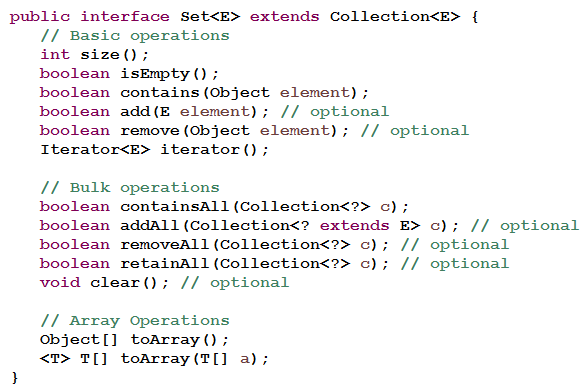
**void** remove(); // optional

}

**public interface Set<E> extends Collection<E>:**

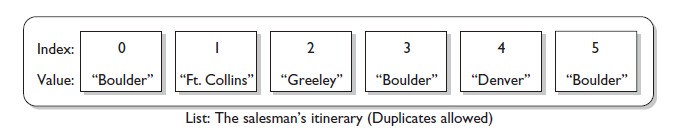
1. Set — A Set cares about uniqueness—it doesn't allow duplicates elements.
2. The three Set implementations are:

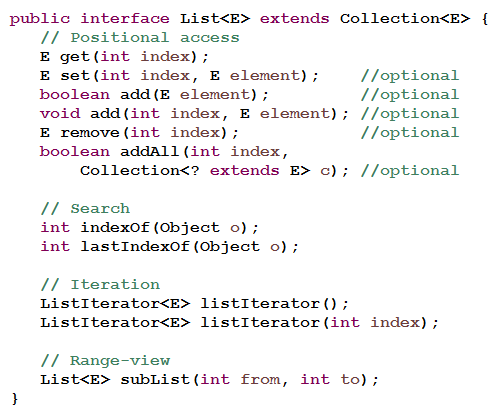
* HashSet
* LinkedHashSet
* TreeSet



**public interface List<E>extends Collection<E>:**

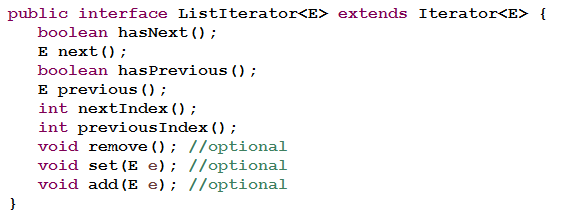
1. List — an ordered collection (sometimes called a *sequence*).
2. Lists can contain duplicate elements. The user of a List generally has precise control over where in the list each element is inserted and can access elements by their integer index (position).
3. A List cares about the index. The one thing that List has that non-lists don't have is a set of methods related to the index. Those key methods include things like get (int index), indexOf(Object o), add(int index, Object obj), and so on.
4. All three List implementations are ordered by index position:-

* ArrayList
* Vector
* LinkedList



**A note on ListIterators:**

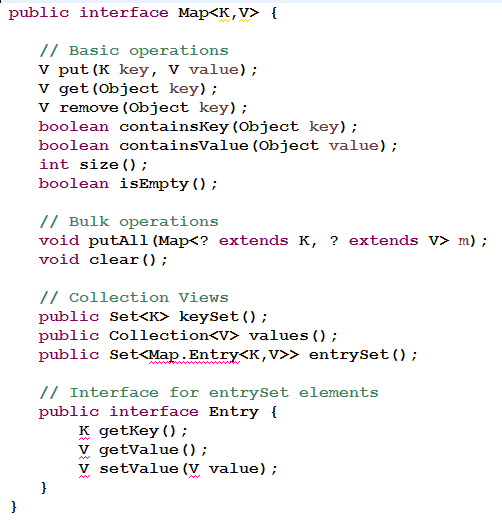
1. The three methods that ListIterator inherits from Iterator (hasNext, next, and remove) do exactly the same thing in both interfaces. The hasPrevious and the previous operations are exact analogues of hasNext and next. The former operations refer to the element before the (implicit) cursor, whereas the latter refer to the element after the cursor. The previous operation moves the cursor backward, whereas next moves it forward.
2. The nextIndex method returns the index of the element that would be returned by a subsequent call to next, and previousIndex returns the index of the element that would be returned by a subsequent call to previous
3. The set method overwrites the last element returned by next or previous with the specified element.
4. The add method inserts a new element into the list immediately before the current cursor position.

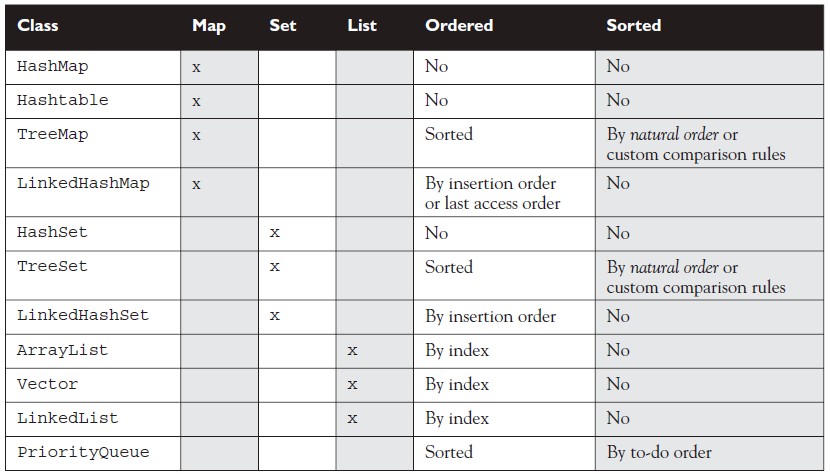


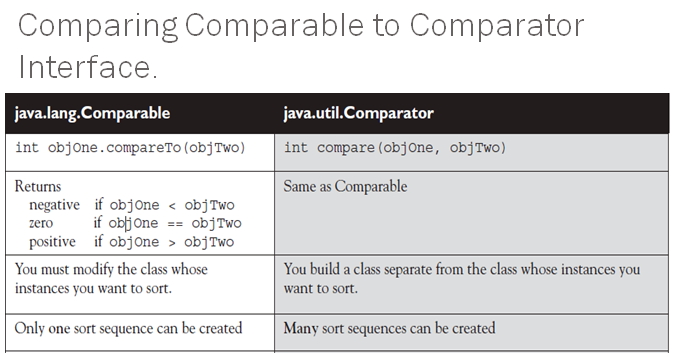
**public interface Map<K,V>:**

1. Map — an object that maps keys to values. A Map cannot contain duplicate keys; each key can map to at most one value. If you've used Hashtable, you're already familiar with the basics of Map.
2. The three Set implementations are:

* HashMap
* Hashtable
* LinkedHashMap
* TreeMap



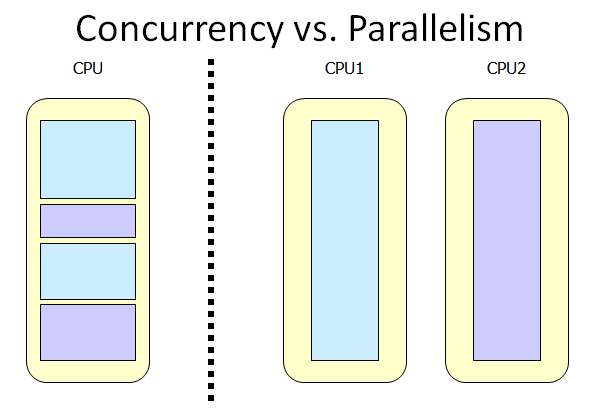


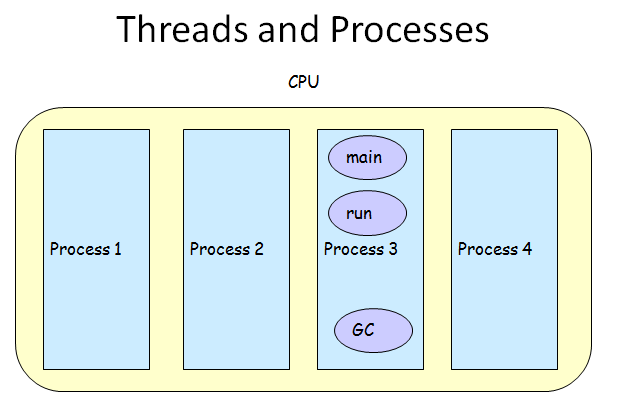


**Java Threads**

**Multitasking and Multithreading:**

1. Multitasking:
   1. refers to a computer's ability to perform multiple jobs concurrently
   2. more than one program are running concurrently, e.g., UNIX
2. Multithreading:
   1. A thread is a single sequence of execution within a program
   2. refers to multiple threads of control within a single program
   3. each program can run multiple threads of control within it, e.g., Web Browser





**What are Threads Good For?**

1. To maintain responsiveness of an application during a long running task
2. To enable cancellation of separable tasks
3. Some problems are intrinsically parallel
4. To monitor status of some resource (e.g., DB)
5. Some APIs and systems demand it (e.g., Swing)

**Application Thread:**

1. When we execute an application:
   1. The JVM creates a Thread object whose task is defined by the main() method
   2. The JVM starts the thread
   3. The thread executes the statements of the program one by one
   4. After executing all the statements, the method returns and the thread dies

**Multiple Threads in an Application:**

1. Each thread has its private run-time stack
2. If two threads execute the same method, each will have its own copy of the local variables the methods uses
3. However, all threads see the same dynamic memory, i.e., heap (are there variables on the heap?)
4. Two different threads can act on the same object and same static fields concurrently

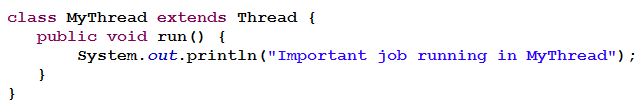
**Creating Threads:**

1. There are two ways to create our own Thread object:

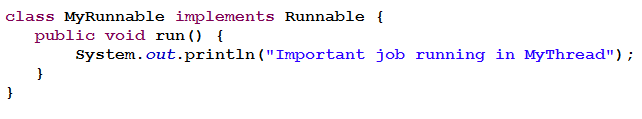
* Sub-classing the Thread class and instantiating a new object of that class
* Implementing the Runnable interface

1. In both cases the run() method should be implemented
2. Thread objects can also be created by calling the Thread constructor that takes a Runnable argument. The Runnable object is said to be the *target of the* thread.
3. We can call start () on a Thread object only once. If start () is called more than once on a Thread object, it will throw a RuntimeException.
4. It is legal to create many Thread objects using the same Runnable object as the target.
5. When a Thread object is created, it does not become a *thread of execution* until its start () method is invoked. When a Thread object exists but hasn't been started, it is in the *new state and is not considered alive.*

**Extending Thread:**



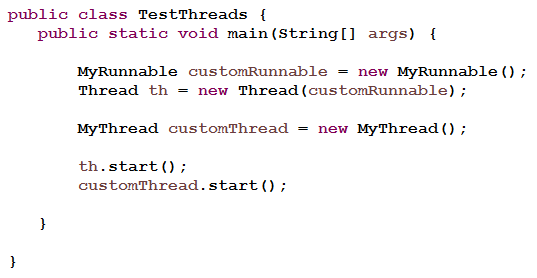
**Implementing Runnable:**



**A Runnable Object:**

1. When running the Runnable object, a Thread object is created from the Runnable object.
2. The Thread object’s run () method calls the Runnable object’s run () method.
3. Allows threads to run inside any object, regardless of inheritance.

**Starting the Threads:**



**Thread Methods:**

1. **void start()**

Creates a new thread and makes it runnable.

This method can be called only once.

1. **void run()**

The new thread begins its life inside this method

1. **void stop() (deprecated)**

The thread is being terminated.

1. **void yield()**

Causes the currently executing thread object to temporarily pause and allow other threads to execute

Allow only threads of the same priority to run

1. **void sleep(int *m*) or sleep(int *m*, int *n*)**

The thread sleeps for *m* milliseconds, plus *n* nanoseconds

1. **public final void join()**

The non-static join() method of class Thread lets one thread "join onto the end“ of another thread.

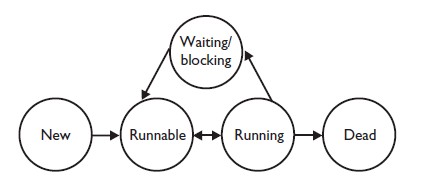
Thread t = **new** Thread();

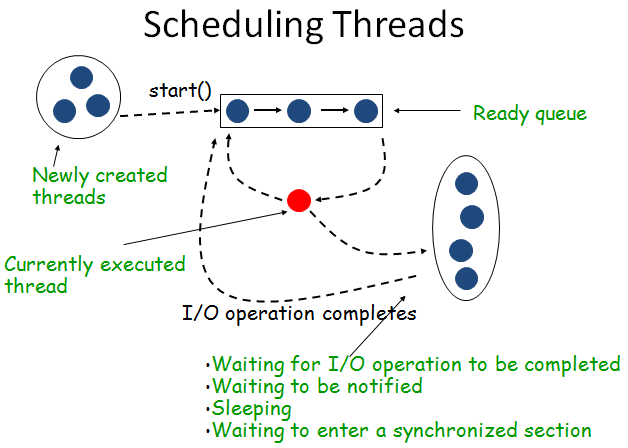
t.start();

t.join();

In other words, the code t.join() means "Join me (the current thread) to the end of t, so that t must finish before I (the current thread) can run again."

**Transitioning Between Thread States:**





1. There's no guarantee that threads will take turns in any fair way. It's up to the thread scheduler, as determined by the particular virtual machine implementation. If you want a guarantee that your threads will take turns regardless of the underlying JVM, you can use the sleep() method. This prevents one thread from hogging the running process while another thread starves. (In most cases, though, yield() works well enough to encourage your threads to play together nicely.)
2. A running thread may enter a blocked/waiting state by a wait(), sleep(),or join() call.
3. A running thread may enter a blocked/waiting state because it can't acquire the lock for a synchronized block of code.
4. When the sleep or wait is over, or an object's lock becomes available, the thread can only re-enter the runnable state. It will go directly from waiting to running (well, for all practical purposes anyway).
5. A dead thread cannot be started again.

**Scheduling:**

1. Thread scheduling is the mechanism used to determine how runnable threads are allocated CPU time.
2. A thread-scheduling mechanism is either preemptive or non-preemptive.
3. Preemptive scheduling – the thread scheduler preempts (pauses) a running thread to allow different threads to execute
4. Non-preemptive scheduling – the scheduler never interrupts a running thread.
5. The non-preemptive scheduler relies on the running thread to yield control of the CPU so that other threads may execute.

**Starvation:**

1. A non-preemptive scheduler may cause starvation (runnable threads, ready to be executed, wait to be executed in the CPU a very long time, maybe even forever)
2. Sometimes, starvation is also called a dead lock

**Java Scheduling:**

1. Execution of multiple threads in some order on a single CPU system is called scheduling.
2. Java uses fixed-priority scheduling algorithms to decide which thread to execute.
3. The thread with the highest priority runs first.
4. If another thread with a higher priority is started, Java makes the lower priority thread wait.
5. If more than one thread exists with the same priority, Java quickly switches between them in round-robin fashion BUT only if the operating system uses time-slicing.
6. It is possible to assign a thread priority.
7. Top priority is 10, lowest priority is 1, Normal priority i.e. priority by default is 5.

* Thread.MIN\_PRIORITY - minimum thread priority
* Thread.MAX\_PRIORITY - maximum thread priority
* Thread.NORM\_PRIORITY - maximum thread priority

1. The main thread is created with priority NORM\_PRIORITY.
2. When a thread is created, it takes the priority of the thread which created it.
3. We can check a threads priority using getPriority().
4. We can change a threads priority using setPriority().
5. If we change the priority on an executing thread to a lesser priority, it may stop executing as there may be another thread with a higher-priority.
6. Actual Scheduling depends on the OS.
7. The above act as a guide to scheduling however the actual implementation depends on the Operating System.
8. DO NOT rely on thread priority as a guarantee that the highest priority thread will always be running; the operating system has the final say.
9. Priorities are used as guides to efficiency.

**Daemon Threads:**

1. Daemon threads are “background” threads, that provide services to other threads, e.g., the garbage collection thread
2. The Java VM will not exit if non-Daemon threads are executing.
3. The Java VM will exit if only Daemon threads are executing.
4. Daemon threads die when the Java VM exits

**Synchronization:**

1. Synchronized methods prevent more than one thread from accessing an object's critical method code simultaneously.
2. We can use the synchronized keyword as a method modifier, or to start a synchronized block of code.
3. To synchronize a block of code (in other words, a scope smaller than the whole method), we must specify an argument that is the object whose lock we want to synchronize on.
4. While only one thread can be accessing synchronized code of a particular instance, multiple threads can still access the same object's unsynchronized code.
5. When a thread goes to sleep, its locks will be unavailable to other threads.
6. Static methods can be synchronized, using the lock from the java.lang.Class instance representing that class.

**void** **synchronized** function( ){

//do stuff;

}

**What is Enum in Java?**

Enum in Javais a keyword, a feature which is used to represent fixed number of well-known values in Java, For example, Number of days in Week, Number of planets in Solar system etc. Enumeration (Enum) in Java was introduced in JDK 1.5.  
Enumeration (Enum) was not originally available in Java though it was available in another language like C and C++, but eventually, Java realized and introduced Enum on JDK 5 (Tiger) by keyword Enum.

If you have used Enumeration before in C or C++ then you will not be uncomfortable with Java Enum but in my opinion, Enum in Java is more rich and versatile than in any other language.   
  
**How to represent enumerable value without Java enum:**

1. Since Enum in Java is only available from Java 1.5 it's worth to discuss how we used to represent enumerable values in Java prior JDK 1.5 and without it. I use public static [final constant](http://javarevisited.blogspot.sg/2011/12/final-variable-method-class-java.html) to replicate enum like behavior. Let’s see an Enum example in Java to understand the concept better. In this example, we will use US Currency Coin as enumerable which has values like PENNY (1) NICKLE (5), DIME (10), and QUARTER (25).
   1. **public** **class** CurrencyDenom {

**public** **static** **final** **int** ***PENNY*** = 1;

**public** **static** **final** **int** ***NICKLE*** = 5;

**public** **static** **final** **int** ***DIME*** = 10;

**public** **static** **final** **int** ***QUARTER*** = 25;

}

* 1. **public** **class** Currency {

**private** **int** currency; //CurrencyDenom.PENNY,CurrencyDenom.NICKLE,

// CurrencyDenom.DIME,CurrencyDenom.QUARTER

}  
 Though this can serve our purpose it has some serious limitations:  
  
**1) No Type-Safety**: First of all it’s not [type-safe](http://javarevisited.blogspot.sg/2011/09/generics-java-example-tutorial.html); we can assign any valid int value to currency e.g. 99 though there is no coin to represent that value.  
**2) No Meaningful Printing**: printing value of any of these constant will print its numeric value instead of meaningful name of coin e.g. when we print NICKLE it will print "5" instead of "NICKLE"  
**3) No namespace:** to access the currencyDenom constant we need to prefix class name e.g. CurrencyDenom.PENNY instead of just using PENNY though this can also be achieved by using static import.

**Java Enum** is the answer of all this limitation. Enum in Java is type-safe, provides meaningful String names and has their own namespace. Now let's see the same example using Enum in Java:

**public** **enum** Currency {

***PENNY***, ***NICKLE***, ***DIME***, ***QUARTER***

};

Here Currency is our enum and PENNY, NICKLE, DIME, QUARTER are enum constants. Notice curly braces around enum constants because Enum is a type like [class](http://javarevisited.blogspot.sg/2011/10/class-in-java-programming-general.html) and [interface in Java](http://javarevisited.blogspot.sg/2012/04/10-points-on-interface-in-java-with.html). Also, we have followed the similar naming convention for enum like class and interface (first letter in Caps) and since Enum constants are implicitly static final we have used all caps to specify them like Constants in Java.

**What is Enum in Java?**

Now back to primary questions **“What is Enum in java”** simple *answer Enum is a keyword in java* and on more detail term Java Enum is a type like class and interface and can be used to define a set of Enum constants.   
  
Enum constants are [implicitly static and final](http://javarevisited.blogspot.sg/2011/12/final-variable-method-class-java.html) and we cannot change their value once created. Enum in Java provides type-safety and can be used inside switch statement like int variables.   
  
Since enum is a keyword we cannot use as a variable name and since its only introduced in JDK 1.5 all our previous code which has an enum as a variable name will not work and needs to be refactored.

**Benefits of using Enums in Java:**

1) **Enum is type-safe** we cannot assign anything else other than predefined Enum constants to an Enum variable. It is a compiler error to assign something else, unlike the public static final variables used in [Enumint pattern](http://javarevisited.blogspot.com/2015/10/133-java-interview-questions-answers-from-last-5-years.html) and [Enum String pattern](http://java67.blogspot.com/2015/03/top-40-core-java-interview-questions-answers-telephonic-round.html).  
2) Enum has its own namespace.  
3) The best feature of Enum is **we can use Enum in Java inside Switch statement** like int or char primitive data type.

4) Adding new constants on [Enum in Java](http://java67.blogspot.com/2014/04/what-java-developer-should-know-about-Enumeration-type-in-Java.html) is easy and we can add new constants without breaking the existing code.

**Important points about Enum in Java**

1) **Enums in Java are type-safe** and has their own namespace. It means our enum will have a type for example "Currency" in below example and we cannot assign any value other than specified in Enum Constants.

**public** **enum** Currency {

***PENNY***, ***NICKLE***, ***DIME***, ***QUARTER***

};

Currency coin =Currency.***PENNY***;//compile

coin=1; //compilation error

2**) Enum in Java are reference types** like [class](http://javarevisited.blogspot.sg/2011/10/class-in-java-programming-general.html) or [interface](http://javarevisited.blogspot.sg/2012/04/10-points-on-interface-in-java-with.html) and we can define constructor, methods and variables inside java Enum which makes it more powerful than Enum in C and C++ as shown in next example of Java Enum type.  
  
3) We can specify values of enum constants at the creation time as shown in below example:   
**public** **enum** Currency {***PENNY(1)***, ***NICKLE(4)***, ***DIME(10)***, ***QUARTER(25)***};

But for this to work we need to define a member variable and a constructor because PENNY (1) is actually [calling a constructor](http://javarevisited.blogspot.sg/2012/01/what-is-constructor-overloading-in-java.html) which accepts int value, see below example.

  **public** **enum** Currency {

***PENNY***(1), ***NICKLE***(5), ***DIME***(10), ***QUARTER***(25);

**private** **int** value;

**private** Currency(**int** value) {

**this**.value = value;

}

};

**The constructor of enum in java** must be [private](http://javarevisited.blogspot.sg/2012/03/private-in-java-why-should-you-always.html) any other access modifier will result in compilation error. Now to get the value associated with each coin we can define a public getValue () method inside Java enum like any normal Java class. Also, the semicolon in the first line is optional.  
4) Enum constants are implicitly [static](http://javarevisited.blogspot.sg/2012/03/mixing-static-and-non-static.html) and [final](http://javarevisited.blogspot.sg/2010/10/why-string-is-immutable-in-java.html) and cannot be changed once created. For example, below code of java enum will result in compilation error:

Currency.***PENNY*** = Currency.***DIME***;

The final field EnumExamples.Currency.PENNY cannot be reassigned.  
  
  5) **Enum in java can be used as an argument on switch statement** and with "case:" like int or char primitive type. This feature of java enum makes them very useful for switch operations. Let’s see an example of how to use java enum inside switch statement:

Currency usCoin = Currency.***DIME***;

**switch** (usCoin) {

**case** ***PENNY***:

System.***out***.println("Penny coin");

**break**;

**case** ***NICKLE***:

System.***out***.println("Nickle coin");

**break**;

**case** ***DIME***:

System.***out***.println("Dime coin");

**break**;

**case** ***QUARTER***:

System.***out***.println("Quarter coin");

}

6) Since **constants defined inside Enum in Java are final we can safely compare them using "==", the equality operator** as shown in following example of JavaEnum:

Currency usCoin = Currency.***DIME***;

**if** (usCoin == Currency.***DIME***) {

System.***out***.println("enum in java can be compared using ==");

}

By the way comparing objects using == operator is not recommended, Always use [equals() method](http://javarevisited.blogspot.sg/2011/02/how-to-write-equals-method-in-java.html) or [compareTo() method](http://javarevisited.blogspot.sg/2011/11/how-to-override-compareto-method-in.html) to compare Objects.

7) Java compiler automatically generates static [values () method](http://java67.blogspot.com/2013/03/how-to-iterate-over-java-enum-using.html) for every enum in java. Values() method returns array of Enum constants in the same order they have listed in Enum and we can use values() to [iterate](http://javarevisited.blogspot.sg/2011/10/java-iterator-tutorial-example-list.html) over values of Enum  in Java as shown in below example:

**for** (Currency coin : Currency.*values*()) {

System.***out***.println("coin: " + coin);

}

**And it will print:**

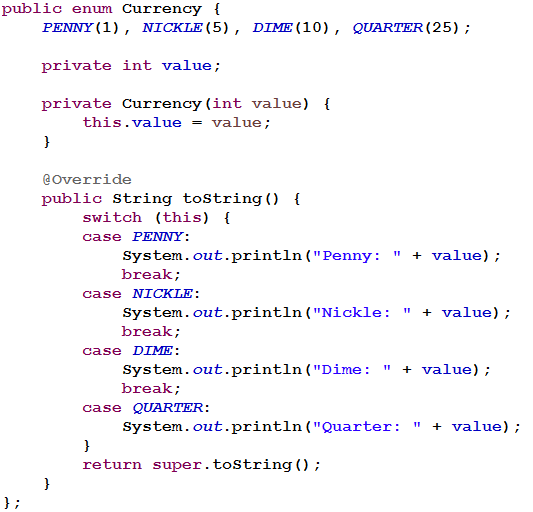
coin**:PENNY**

coin**:NICKLE**

coin**:DIME**

coin**:QUARTER**

Notice the order is exactly the same **as defined order in the Enum**.  
  
8) In Java, [Enum can override methods](http://java67.blogspot.com/2012/08/how-to-convert-enum-to-string-in-java.html) also. Let’s see an example of overriding toString() method **inside Enum in Java** to provide a **meaningful description** for enums constants.



And here is how it looks like when displayed:

Currency usCoin = Currency.***DIME***;

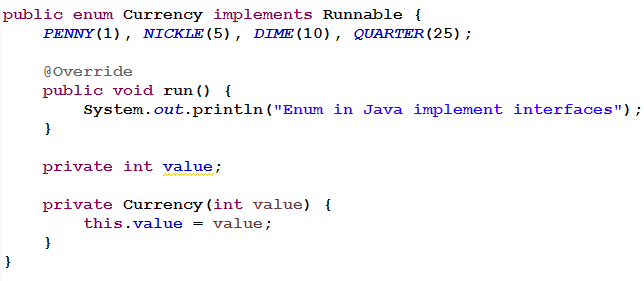
System.***out***.println(usCoin);

**Output:**

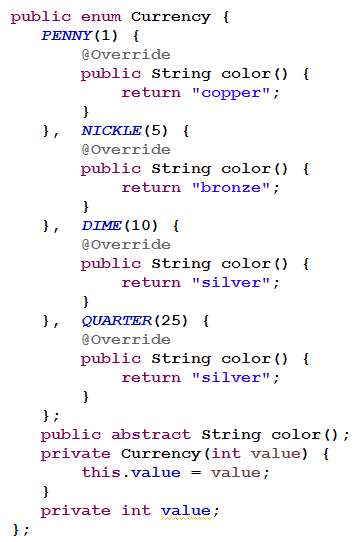
Dime: 10

DIME

9) Two new collection classes [EnumMap](http://javarevisited.blogspot.com/2012/09/what-is-enummap-in-java-example-tutorial.html) and [EnumSet](http://javarevisited.blogspot.com/2014/03/how-to-use-enumset-in-java-with-example.html)are added into collection package to support Java Enum. These classes are a high-performance implementation of [Map and Set interface in Java](http://javarevisited.blogspot.sg/2012/07/create-read-only-list-map-set-example-java.html) and we should use this whenever there is any opportunity.  
  
EnumSet doesn't have any public constructor instead it provides factory methods to create instance e.g. EnumSet.of() methods. This design allows EnumSet to internally choose between two different implementations depending upon the size of Enum constants.  
  
If Enum has less than 64 constants than EnumSet uses RegularEnumSet class which internally uses a long variable to store those 64 Enum constants and if Enum has more keys than 64 then it uses JumboEnumSet  
10) **We cannot create an instance of enums by using new operator** in Java because the [constructor of Enum in Java can only be private](http://java67.blogspot.com/2012/11/java-enum-example-with-constructor.html) and Enums constants can only be created inside Enums itself.  
  
11) An instance of Enum in Java is created when any Enum constants are first called or referenced in code.  
  
12) **Enum in Java can implement the interface** and override any method like normal class It’s also worth noting that Enum in java implicitly implements both [Serializable](http://javarevisited.blogspot.sg/2012/01/serializable-externalizable-in-java.html)and [Comparable](http://javarevisited.blogspot.sg/2011/06/comparator-and-comparable-in-java.html) interface. Let's see an example of **how to implement interface using Java Enum**:



13) **We can define abstract methods inside Enum in Java** and can also provide a different implementation for different instances of enum in java.  Let’s see an example of using [abstract method](http://javarevisited.blogspot.sg/2010/10/abstraction-in-java.html) inside enum in java



In this example since every coin will have the different color we made the color() method abstract and let each instance of Enum to define  their own color. We can get color of any coin by just calling the color() method as shown in below example of Java Enum:

System.***out***.println("Color: " + Currency.***DIME***.color());

**Real world Examples of Enum in Java:**

So far we have learned what Enum can do for us in Java. We learned that enum can be used to represent well known fixed set of constants,  [enum can implement interfac](http://java67.blogspot.com/2013/07/java-enum-code-example-softdrinks-how.html)e, it can be used in switch case like int, short and String and Enum has so many useful built-in metods like values(), vlaueOf(), name(), and ordinal(), but we didn't learn where to use the Enum in Java?   
  
I think some real world examples of enum will do a lot of good to many pepole and that's why I am going to summarize some of the popular usage of Enum in Java world below.   
  
**Enum as Thread Safe Singleton:**  
One of the most popular uses of Java Enum is to impelment the Singleton design pattern in Java. In fact, Enum is the easieset way to create a [thread-safe Singleton in Java](http://javarevisited.blogspot.com/2012/12/how-to-create-thread-safe-singleton-in-java-example.html). It offer so many advantage over traditional implementation using class e.g. built-in Serialization; guarantee that Singleton will always be Singleton and many more.

**Strategy Pattern using Enum:**  
we can also implement the Strategy design pattern using Enumeration type in Java. Since Enum can implement interface, it's a good candidate to implement the [Strategy interface](http://java67.blogspot.com/2014/12/strategy-pattern-in-java-with-sample.html) and define individual strategy. By keeping all related Strategy in one place, Enum offer better maintenance support. It also doesn't break the open closed design principle as per se because any error will be detected at compile time.

**Enum as replacement of Enum String or int pattern:**  
There is now no need to use String or integer constant to represent fixed set of things e.g. status of object like ON and OFF for a button or START, IN PROGRESS and DONE for a Task. Enum is much better suited for those needs as it provide compile time type safety and better debugging assistant than String or Integer.  
  
**Enum as State Machine:**  
We can also use Enum to implement State machine in Java. A State machine transition to predefine set of states based upon current state and given input. Since Enum can implement interface and override method, we can use it as State machine in Java.

**Enum Java valueOf example**

One of my readers pointed out that I have not mentioned about the valueOf method of enum in Java, which is used to [convert String to enum in Java](http://java67.blogspot.com/2012/08/how-to-create-enum-from-string-in-java.html).  
  
“We could also include **valueOf() method of enum** in java which is added by compiler in any enum along with values() method. **Enum.valueOf()** is a static method which takes a string argument and can be used to convert a String into an enum. One think though we would like to keep in mind is that valueOf(String) method of enum will throw "**Exception in thread "main" java.lang.IllegalArgumentException: No enum const class**" if we supply any string other than enum values.

Ordinal method of Java Enum returns the position of a Enum constant as they declared in enum while name()of Enum returns the exact string which is used to create that particular Enum constant.” name() method can also be used for [converting Enum to String in Java](http://javarevisited.blogspot.sg/2011/12/convert-enum-string-java-example.html).

**File Navigation, I/O and Object Serialization**

**File:** The API says that the class File is "An abstract representation of file and directory pathnames." The File class isn't used to actually read or write data; it's used to work at a higher level, making new empty files, searching for files, deleting files, making directories, and working with paths.

**FileReader:** This class is used to read character files. Its read()methods are fairly low-level, allowing you to read single characters, the whole stream of characters, or a fixed number of characters. FileReaders are usually wrapped by higher-level objects such as BufferedReaders, which improve performance and provide more convenient ways to work with the data.

**BufferedReader:** This class is used to make lower-level Reader classes like FileReader more efficient and easier to use. Compared to FileReaders, BufferedReaders read relatively large chunks of data from a file at once, and keep this data in a buffer.

BufferedReader provides more convenient methods such as readLine(), that allow you to get the next line of characters from a file.

**FileWriter:** This class is used to write to character files. Its write() methods allow you to write character(s) or Strings to a file. FileWriters are usually wrappedby higher-level Writer objects such as BufferedWriters or PrintWriters, which provide better performance and higher-level, more flexible methods to write data.

**BufferedWriter:** This class is used to make lower-level classes like FileWriters more fficient and easier to use.

The BufferedWriter class also provides a newLine() method to create platform-specific line separators automatically.

**PrintWriter:** This class has been enhanced significantly in Java 5. Because of newly created methods and constructors (like building a PrintWriter with a File or a String), you might find that you can use PrintWriter in places where you previously needed a Writer to be wrapped with a FileWriter and/or a BufferedWriter. New methods like format (), printf(), and append() make PrintWriters very flexible and powerful.

\*\* Stream classes are used to read and write bytes, and Readers and Writers are used to read and write characters.

**Working with Files and Directories**

1. Always creates a File object, and then does one of two things:

File file = new File ("foo");

1. If "foo" does NOT exist, no actual file is created.
2. If "foo" does exist, the new File object refers to the existing file.
3. Notice that File file = new File ("foo"); NEVER creates an actual file.

3. There are two ways to create a file:

1. Invoke the createNewFile()method on a File object.

**For example:**

File file = new File("foo"); // no file yet

file.createNewFile(); // make a file, "foo" which is assigned to //'file'

1. Create a Writer or a Stream. Specifically, create a FileWriter, a PrintWriter, or a FileOutputStream. Whenever you create an instance of one of these classes, you automatically create a file, unless one already exists, for instance.

File file = new File ("foo"); // no file yet

PrintWriter pw = new PrintWriter(file); // make a PrintWriter object AND make a file, "foo" to //which 'file' is assigned, AND assign 'pw' to the PrintWriter.

1. Creating a directory:

File myDir = new File("mydir"); // create an object

myDir.mkdir(); // create an actual directory

1. Once you've got a directory, you put files into it, and work with those files:

File myFile = new File(myDir, "myFile.txt");

myFile.createNewFile();

1. Checking whether it is file or directory:

File existingDir = new File("existingDir"); // assign a dir

System.out.println(existingDir.isDirectory());

1. Getting the content from a folder:

File existingDir = **new** File("existingDir"); // assign a dir

**if** (existingDir.isDirectory()) {

String[] contents = existingDir.list();

System.*out*.println(Arrays.*toString*(contents));

}

1. Delete a file or directory:

File myFile = new File(myDir, "myFile.txt");

myFile.delete();

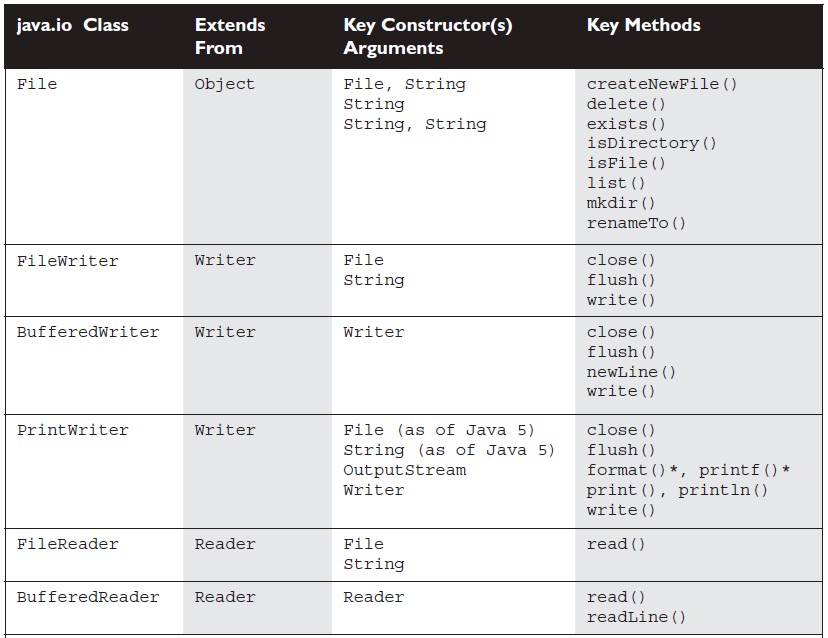
File existingDir = new File("existingDir"); // assign a dir

existingDir.delete();

\*\*Note: We cannot delete **non-empty** directory

Streams and I/O:

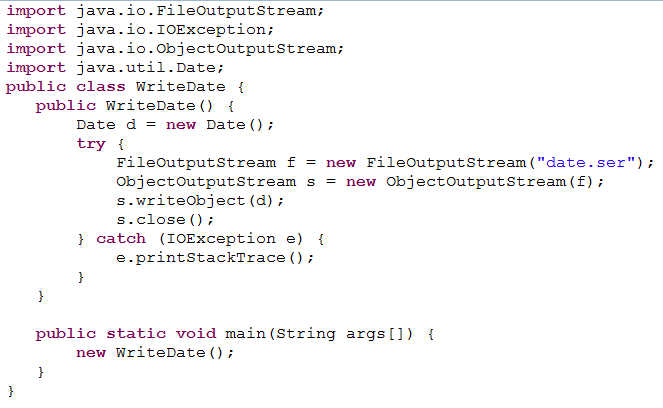
1. Stream is a general mechanism of I/O in java.
2. Stream provides sequential access of data.
3. An input stream can be used by an application to read data. An output stream to write data.
4. Streams are categorized as:
5. Byte Stream :Binary
6. Character Stream: Text
7. A file, network connection, arrays can act as stream.
8. Whenever we have to make program of I/O then we use java.io package



**Object serialization:**

1. Primary purpose of java serialization is to write an object into a stream, so that it can be transported through a network and that object can be rebuilt again.
2. Write objects to a file, instead of writing primitive types.
3. During serialization the following information is send to the stream:
4. The values of all non-transient and non- static fields
5. The class information needed to reconstruct the object (class name, package etc.).
6. During serialization aggregate state is implicitly serialized.
7. The object to be serialized must implements Serializable interface
8. Use the ObjectInputStream, ObjectOutputStream classes, and the same way that filters are used.

**Write an object to a file:**



**Read an object from a file:**