Chapter - 11

java.lang Package

java.lang Package

The **java.lang** package is the default package that is automatically imported into all java programs. Therefore we need not to import this package explicitly. This package contains classes and interfaces which are commonly used in the java programs. The various classes and interfaces present in **java.lang** package are listed in the below tables.

The following table contains **classes of java.lang package**

Boolean	Byte	Character	Class
Compiler	Double	Enum	Float
Long	Math	Number	Object
Process	ProcessBuilder	Runtime	RuntimePermission
Short	StackTraceElement	StrictMath	String
StringBuilder	System	Thread	ThreadGroup
Void	ClassLoader	Integer	Package
SecurityManager	StringBuffer	Throwable	

The following table contains **interfaces of java.lang package**

Appendable	CharSequence	Cloneable	Iterable
Readable	Runnable	Comparable	

The System class

The **System class** of **java.lang** package is a public final class which cannot be inherited into subclasses. This class contains useful fields(variables) and methods. All variables and methods in System class are public static methods therefore they can be accessed using class name without using the object name. Objects cannot be instantiated to the System class.

Among the facilities provided by the System class are standard input(**in**), standard output(**out**), and error output(**err**) streams; access to externally defined "**properties**"; a means of loading files and libraries; and a utility method for quickly copying a portion of an array.

The following table shows the stream variables of **System** class.

Variables	Meaning
err	The "standard" error output stream variable defined to **PrintStream** class.
in	The "standard" input stream variable defined to **IntputStream** class.
out	The "standard" output stream variable defined to **PrintStream** class.

The following table shows methods of **System** class.

Methods	Meaning
Static void **arraycopy**(Object src, int srcPos, Object dest, int destPos, int length)	Copies an array from the specified source array, beginning at the specified position, to the specified position of the destination array.
Static long **currentTimeMillis**()	Returns the current time in milliseconds.
Static void **exit**(int status)	Terminates the currently running Java Virtual Machine.
Static void **gc**()	Runs the garbage collector.
Static Properties **getProperties**()	Determines the current system properties.
Static String **getProperty**(String key)	Gets the system property indicated by the specified key.
Static void **load**(String filename)	Loads a code file with the specified filename from the local file system as a dynamic library.
Static void **loadLibrary**(String libname)	Loads the system library specified by the libname argument.

Static void **runFinalization**()	Runs the finalization methods of any objects pending finalization.
Static void **setErr**(PrintStream err)	Reassigns the "standard" error output stream.
Static void **setIn**(InputStream in)	Reassigns the "standard" input stream.
Static void **setOut**(PrintStream out)	Reassigns the "standard" output stream.
Static void **setProperties**(Properties pro ps)	Sets the system properties to the Properties argument.
Static String **setProperty**(String key, String value)	Sets the system property indicated by the specified key.

out

The **out** variable defined as **public static final PrintStream out;** in the **System** class. The **out** variable represents as "**standard**" output stream. This stream is already open and ready to accept output data. Typically this stream corresponds to display output or another output destination specified by the host environment or user.

For simple stand-alone Java applications, a typical way to write a line of output data is:

System.out.println(data);

in

The **in** variable defined as **public static final InputStream in;** in the **System** class. The **in** variable represents as "**standard**" input stream. This stream is already open and ready to supply input data. Typically this stream corresponds to keyboard input or another input source specified by the host environment or user.

err

The **err** variable defined as **public static final PrintStream err;** in the **System** class. The **err** variable represents as "**standard**" error output stream. This stream is already open and ready to accept output data. Typically this stream corresponds to display output or another output destination specified by the host environment or user. By convention, this output stream is used to display error messages or other information that

should come to the immediate attention of a user even if the principal output stream, the value of the variable out, has been redirected to a file or other destination that is typically not continuously monitored.

The currentTimeMillis()

This method of **System** class returns the current time in milliseconds. Note that while the unit of time of the return value is a millisecond, the value depends on the underlying operating system and may be larger. For example, many operating systems measure time in milliseconds.

Bin>edit timemillis.java

```
class timemillis
{
   public static void main(String argv[])
   {
         long start,end;
         start=System.currentTimeMillis();
         System.out.print("\nProgram executing…");
         for(int i=1;i<=50000;i++)
               System.out.print("");
         end=System.currentTimeMillis();
         System.out.print("\nProgram terminates…");
         System.out.print("\nTime taken by program to
            complete its execution is" +(end-start)+
            " Milli seconds.");
   }
}
```

Bin>javac timemillis.java
Bin>java timemillis
Program executing...
Program terminates...
Time taken by program to complete its execution is 16 Milli seconds.

Explanation:

When the above executes, in the statement **start=System.currentTimeMillis();** the currentTimeMillis() return the current time in milliseconds which assigns to **start** variable. After the for loop the statement **end=System.currentTimeMillis();** assigns time in milliseconds to **end** variable. The last **print()** prints the difference between **end** and **start** variables, which is the time of the program that took to complete its execution.

The arraycopy()

The **arraycopy()** method of **System** class copies values of an array into another array of any type. It is the easiest method of copying values from one array to another instead of copying using loops. The signature of **arraycopy()** is **public static void arraycopy(Object src, int srcPos, Object dest, int destPos, int length)**

In the above signature of **arraycopy()**, the **src** is source array name, **srcPos** is source array position from where the values should be copied. The **dest** is the destination array name into which the values should be coppied, **desPos** is the position of the destination array from where the copying should start. Finally, **length** is specifies the number of elements values should be copied.

Bin>edit copyarray.java

```
class copyarray
{
      public static void main(String argv[])
      {
            int a[]={10,20,30,40,50,60,70,80,90,100};
            int b[]=new int[10];
            int i;
            System.arraycopy(a,0,b,3,5);
            System.out.print("\n values of a array…\n");
            for(i=0; i<10; i++)
                  System.out.print("  "+a[i]);
            System.out.print("\n values of b array…\n");
            for(i=0;i<10; i++)
                  System.out.print("  "+b[i]);
      }
}
```

Bin>javac copyarray.java
Bin>java copyarray

values of a array...
10 20 30 40 50 60 70 80 90 100
values of a array...
0 0 0 10 20 30 40 50 0 0

Explanation :

In the above program, the arraycopy() copies **a** array values from **0**th element to **b** array and stores from **3**rd element , it copies **5** elements of array **a** to **b** and we get the output as shown above.

The getProperty() Method

The **getProperty()** method of **System** class returns the string value of the various environment variables. If the variable is not found then it returns null. The following are the properties present in java are:

Keys	Meaning
java.version	Java Runtime Environment version
java.vendor	Java Runtime Environment vendor
java.vendor.url	Java vendor URL
java.home	Java installation directory
java.vm.specification.version	Java Virtual Machine specification version
java.vm.specification.vendor	Java Virtual Machine specification vendor
java.vm.specification.name	Java Virtual Machine specification name
java.vm.version	Java Virtual Machine implementation version
java.vm.vendor	Java Virtual Machine implementation vendor
java.vm.name	Java Virtual Machine implementation name
java.specification.version	Java Runtime Environment specification version
java.specification.vendor	Java Runtime Environment specification vendor
java.specification.name	Java Runtime Environment specification name
java.class.version	Java class format version number
java.class.path	Java class path
java.library.path	List of paths to search when loading libraries
java.io.tmpdir	Default temp file path
java.compiler	Name of JIT compiler to use
java.ext.dirs	Path of extension directory or directories
os.name	Operating system name
os.arch	Operating system architecture
os.version	Operating system version
file.separator	File separator ("/" on UNIX)
path.separator	Path separator (":" on UNIX)
line.separator	Line separator ("\n" on UNIX)
user.name	User's account name
user.home	User's home directory
user.dir	User's current working directory

The signature of **getProperty()** is

public static String getProperty(String key)

here, key is the name of the system property(environment variable).

Bin>edit getproperty1.java

```
class  getproperty1
{
   public static void main(String argv[])
   {
         System.out.print("\n java version :" +
         System.getProperty("java.version") );
         System.out.print("\n class path :" +
         System.getProperty("java.class.path") );
         System.out.print("\n os name :" +
         System.getProperty("os.name") );
   }
}
```

Bin>javac getproperty1.java
Bin>java getproperty

The above program gives different output according to the OS, java soft and so on.

The exit( ) method

The **exit()** method of **java.lang** package terminates the program unconditionally. The signature of the **exit()** method is

public static void exit(int status)

The argument serves as a status code; by convention, a nonzero status code indicates abnormal termination.

Bin>edit exit1.java

```
class exit1
{
   public static void main(String argv[])
   {
```

```
        int n=1;
        System.out.print("\n sample program on exit() method");
        if(n==1)
          System.exit(1);
        System.out.print("\n this statement does not execute.");
    }
}
```

Bin>javac exit1.java

Bin>java exit1

sample program on exit() method

The gc( ) method

The **gc()** method of **java.lang** package, when executed invokes the garbage collector program which comes into heap memory where it deallocates the unwanted memory of java program, so that the memory can be used for other purpose. The signature of this method is

public static void gc()

It is not recommended to execute **gc()** method explicitly in the program manually, because the jvm executes this method implicitly when ever the memory is required.

The program on **gc()** method was demonstrated in the topic of **finalize()** method.

Class Process

The **Process** class is an abstract class present in **java.lang** package. The **Runtime.exec()** method create a native process and return an instance of a subclass of Process that can be used to control the process and obtain information about it. The class Process provides methods for performing input from the process, performing output to the process, waiting for the process to complete, checking the exit status of the process, and destroying (killing) the process.

The following table shows methods of **Process** class.

Methods	Meaning
abstract void **destroy**()	Kills the subprocess.
abstract int **exitValue**()	Returns the exit value for the subprocess.
abstract InputStream **getErrorStream**()	Gets the error stream of the subprocess.
abstract InputStream **getInputStream**()	Gets the input stream of the subprocess.
abstract OutputStream **getOutputStream**()	Gets the output stream of the subprocess.
abstract int **waitFor**()	causes the current thread to wait, if necessary, until the process represented by this Process object has terminated.

The waitFor( ) method

This method of **Process** class causes the current thread to wait, if necessary, until the process represented by this **Process** object has terminated. This method returns immediately if the subprocess has already terminated. If the subprocess has not yet terminated, the calling thread will be blocked until the subprocess exits.

public abstract int waitFor( ) throws InterruptedException

the exit value of the process. By convention, **0** indicates normal termination.

The exitValue( ) method

This method of the Process class returns the exit value for the subprocess. The exit value of the subprocess represented by this Process object. by convention, the value 0 indicates normal termination.

public abstract int exitValue()

The destroy() method

This method of Process class kills the subprocess. The subprocess represented by this Process object is forcibly terminated.

public abstract void destroy()

Class Runtime

The **Runtime** class is present in **java.lang** package. Every Java application has a single instance of class **Runtime** that allows the application to interface with the environment in which the application is running. The current runtime can be obtained from the **getRuntime()** method.

An application cannot create its own instance of this class.

The methods in **Runtime** class are shown in the below table.

Methods	Meaning
int **availableProcessors**()	Returns the number of processors available to the Java virtual machine.
Process **exec**(String command)	Executes the specified string command in a separate process.
Process **exec**(String[] cmdarray)	Executes the specified command and arguments in a separate process.
Process **exec**(String[] cmdarray, String[] envp)	Executes the specified command and arguments in a separate process with the specified environment.
Process **exec**(String[] cmdarray, String[] envp, File dir)	Executes the specified command and arguments in a separate process with the specified environment and working directory.
Process **exec**(String command, String[] envp)	Executes the specified string command in a separate process with the specified environment.
Process **exec**(String command, String[] envp, File dir)	Executes the specified string command in a separate process with the specified environment and working directory.
void **exit**(int status)	Terminates the currently running Java virtual machine by initiating its shutdown sequence.
long **freeMemory**()	Returns the amount of free memory in the Java Virtual Machine.
void **gc**()	Runs the garbage collector.
static Runtime **getRuntime**()	Returns the runtime object associated with the current Java application.

void **halt**(int status)	Forcibly terminates the currently running Java virtual machine.
long **maxMemory**()	Returns the maximum amount of memory that the Java virtual machine will attempt to use.
long **totalMemory**()	Returns the total amount of memory in the Java virtual machine.

The getRuntime( ) method

This method of **Runtime** class returns the runtime object associated with the current Java application. Most of the methods of class **Runtime** are instance methods therefore they must be called using the **Runtime** class object.

public static Runtime getRuntime()

The exec( ) method

This method of **Runtime** class executes the specified string command in a separate process and returns a new Process object for managing the subprocess.

public Process exec(String command)throws IOException

The command include any executable program(Ex: **notepad.exe**, **calc.exe**, etc,.)

The freeMemory( ) method

This method of **Runtime** class returns the amount of free memory in the Java Virtual Machine. Calling the **gc()** method may result in increasing the value returned by **freeMemory()**.

public long freeMemory()

The **freeMemory()** method returns an approximation to the total amount of memory currently available for future allocated objects, measured in bytes.

The totalMemory( ) method

This method of **Runtime** class returns the total amount of memory in the Java virtual machine. The value returned by this method may vary over time, depending on the host environment.

public long totalMemory()

The total amount of memory currently available for current and future objects, measured in bytes.

The following program executes notepad program using Process and Runtime classes

Bin>edit note1.java

```
class note1
{
   public static void main(String argv[])
   {
         Process p;
         Runtime r=Runtime.getRuntime();
         try
         {
               p=r.exec("notepad");
         }
         catch(Exception e)
         {
               System.out.print("\nError :"+e);
         }
   }
}
```

Bin>javac note1.java

Bin>java note1

output: we get a notepad on the screen.

The Object class

Class **Object** is the root of the class hierarchy. Any class (pre-defined or user-defined) is a sub-class of Object class either directly or indirectly. **Object** is the supermost class in java and it is the super class of any class in java. The methods of **Object** class are available to all classes in java. All objects, including arrays, implement the methods of this class. Methods present in the **Object** class are shown in below table.

Methods	Meaning
Object clone() throws CloneNotSupportedException	Creates a new object that is the same as the invoking object.
boolean equals(Object object)	Returns true if the invoking object is equivalent to object.
void finalize() throws Throwable	Default finalize() method. This is usually overridden by subclasses.
final Class getClass()	Obtains a Class object that describes the invoking object.
in hashCode()	Returns the hash code of a thread waiting on the invoking object.
final void notify()	Resumes execution of all thread waiting on the invoking object.
final void notifyAll()	Resumes execution of all threads waiting on the invoking object.
String toString()	Returns a string that describes the object.
final void wait() throws InterruptedException	Wait on another thread of execution.
final void wait(long milliseconds) throws InterruptedException	Wait up to the specified number of milliseconds on another thread of execution.
final void wait(long milliseconds, int nanoseconds) throws InterruptedException	Wait up to the specified number of milliseconds plus nanoseconds on another thread of execution.

The hashCode( ) method

The **hashCode()**method is defined in **Object** class. Each object that is created has its own hash code and no two objects have same hashcode. This method returns a hash code value of the object.

As much as is reasonably practical, the hashCode method defined by class **Object** does return distinct integers for distinct objects.
The signature of **hashCode()** method is

public int hashCode( )

A Program to demonstrate hashCode() method

Bin>edit hashcode1.java

```
class  A
{
  int x=0;
}
class hashcode1
{
  public static void main(String argv[])
  {
        A obj1=new A();
        A obj2=new A();
        int obj1hcode,obj2hcode;
        obj1hcode=obj1.hashCode();
        obj2hcode=obj2.hashCode();
        System.out.print("\n hash code of obj1="+obj1hcode);
        System.out.print("\n hash code of obj2="+ obj2hcode);
  }
}
```

Bin>javac hashcode1.java

Bin>java hashcode1

hash code of obj1 =17523401

hash code of obj2 =857361

The equals( ) method

The **equals()** method of **Object** class checks where two object are equal or not. The **equals()** method of class **Object** implements the most discriminating possible equivalence relation on objects; that is, for any reference values **x** and **y**, this method returns true if and only if **x** and y refer to the same object (**x==y** has the value **true**). The signature of **equals()** method is..

public boolean equals(Object obj)

Bin>edit equals1.java

```
class A
{
     int x;
}
```

```
class eualsl
{
   public static void main(String argv[])
   {
         A obj1=new A();
         A obj2=new A();
         A obj3=new A();
         obj1.x=10;
         obj2.x=10;
         obj3.x=20;
         A obj4=obj1;
         System.out.print("\nobj1.equals(obj2) =" +
                                   obj1.equals(obj2));
         System.out.print("\nobj1.equals(obj3) =" +
                                   obj1.equals(obj3));
         System.out.print("\nobj1.equals(obj4) =" +
                                   obj1.equals(obj4));
   }
}
```

Bin>javac equals1.java
Bin>java equals1

obj1.equals(obj2) =false
obj1.equals(obj3) =false
obj1.equals(obj4) =true

The clone( ) Method

The **clone()** of **Object** class creates and returns a copy of the invoking object. This method can be used to make the duplicate object of the original. It copies bit by bit of the object so that the original and the duplicate are similar. The objects whose classes that implements from **Cloneable** interface such objects can be duplicated using **clone()** method otherwise the jvm throws **CloneNotSupportedException**. The **Cloneable** interface is an empty interface and need not to override any methods in the subclass. This interface is just used to indicate to the java compiler that whose subclasses object can be duplicated.

The precise meaning of "**copy**" may depend on the class of the object. The general intent is that, for any object x, the expression:

x.clone() != x

will be **true**, and that the expression:

x.clone().getClass() == x.getClass()

will be **true**, but these are not absolute requirements. While it is typically the case that:

x.clone().equals(x)

will be true, this is not an absolute requirement. Copying an object will typically entail creating a new instance of its class, but it also may require copying of internal data structures as well. No constructors are called.
The Object class does not itself implement from the interface **Cloneable**, so calling the **clone()** method on an object whose class is **Object** will result in throwing an exception at run time. The **clone()** method is implemented by the class **Object** as a convenient. This method can also overridden in the subclass. The signature of the **clone()** method is:

protected Object clone()throws CloneNotSupportedException

A program to demonstrate the clone() method to duplicate the objects

Bin>edit clone1.java

```
class A implements Cloneable
{
   public int x;
   public double y;

   public A getduplicate()
   {
         try
         {
               return (A)super.clone();
         }
         catch(CloneNotSupportedException e)
         {
               System.out.print("\nError : "+e);
               return this;
         }
   }
}
```

```
class clone1
{
    public static void main(String argv[])
    {
          A obj1=new A();
          obj1.x=10;
          obj1.y=5.6;
          A obj2=obj1.getduplicate();
          System.out.print("\n obj1 details...");
          System.out.print("\nx="+obj1.x+"\t y="+obj1.y);
          System.out.print("\n obj2 details...");
          System.out.print("\nx="+obj2.x+"\t y="+obj2.y);
    }
}
```

Bin>javac clone1.java
Bin>java clone1

obj1 details...
x=10 y=5.6
obj2 details...
x=10 y=5.6

Explanation:

In the main() method, **obj1** object is created, **x** and **y** of **obj1** are assigned with **10** and **5.6** values. In the statement **obj2=obj1.getduplicate();** the **obj1** calls the **getduplicate()** method and inside the method **super.clone()** creates a duplicate object of the object that invoked the **getduplicate()** method i.e. **obj1** and the returned object reference is type casted to class **A** type and is returned. The returned object reference is assigned to **obj2** in the main(). Therefore the values of objects **obj1** and **obj2** are same.

A program to demonstrate the duplication of object by overriding the clone() method.

Bin>edit clone2.java

```
class A implements Cloneable
{
    public int x;
    public double y;
    protected Object clone()
    {
          try
          {
```

```
                return (A)super.clone();
            }
            catch(CloneNotSupportedException e)
            {
                System.out.print("\nError : "+e);
                return this;

            }
    }
}
class clone2
{
    public static void main(String argv[])
    {
            A obj1=new A( );
            obj1.x=10;
            obj1.y=5.6;
            A obj2=(A)obj1.clone();
            System.out.print("\n obj1 details...");
            System.out.print("\nx="+obj1.x+"\t y="+obj1.y);
            System.out.print("\n obj2 details...");
            System.out.print("\nx="+obj2.x+"\t y="+obj2.y);
    }
}
```

Bin>javac clone2.java

Bin>java clone2

obj1 details...
x=10 y=5.6
obj2 details...
x=10 y=5.6

Explanation:

The above program is same as that of previous program, the difference is, above program is overriding the clone() method

The getClass( ) method

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. You can query the Class object for a variety of information about the class, such as its name, its superclass, and the names of the interfaces

that it implements. The following method gets and displays the class name of an object:

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

One handy use of the **getClass()** method is to create a new instance of a class without knowing what the class is at compile time. This sample method creates a new instance of the same class as **obj** which can be any class that inherits from **Object** (which means that it could be any class):

```
obj2= obj1.getClass().newInstance();
```

public final Class getClass()

We will discuss about **getClass()** method in the **Class** class

Note: See the notes on **wait()**, **notify()** and **notifyAll()** in the multithreading chapter.

The finalize( ) method

The **finalize()** method of **Object** class is automatically called by the garbage collector on an object when garbage collection determines that there are no more references to the object. A subclass overrides the **finalize()** method to dispose of system resources or to perform other cleanup.

The **finalize()** method of object can be overridden in the subclasses to save the resources of the object or to execute any instructions before the object is deleted by garbage collector. The finalize method is never invoked more than once by a Java virtual machine for any given object. The signature of this method is..

protected void finalize()throws Throwable

Note: The **finalize()** method was demonstrated in the garbage collector concept(see in the index).

The Class class

When a class is loaded into memory the jvm creates an object of **Class** type of the class. Using this **Class** type objet we can get the run-time information about class such as class name, super class name, method names, properties etc. The central class representing this information is **java.lang.Class**, a class confusingly named Class.

We can use the **getClass()** method to get a **Class** object. Every class has the class **Object** as an ancestor. A reflection method in the **Object** class is **getClass()**, which returns the object of **Class** type of invoking object class. The **java.lang.Class** has two useful methods: **getName()** which returns the name of the class of an object, **getSuperclass()** returns the object of **Class** type of super class. If a class does not have superclass then it returns **null**.

The following table shows the methods of **Class** are:

Methods	Meaning
static Class **forName**(String className)	Loads the className and returns the Class object associated with the class or interface with the given string name.
static Class **forName**(String name, boolean initialize, ClassLoader loader)	Returns the Class object associated with the class or interface with the given string name, using the given class loader.
Class[]**getClasses**()	Returns an array containing Class objects representing all the public classes and interfaces that are members of the class represented by this Class object.
Constructor[]**getConstructors**()	Returns an array containing Constructor objects reflecting all the public constructors of the class represented by this Class object.
Constructor **getDeclaredConstructor** (Class[] parameterTypes)	Returns a Constructor object that reflects the specified constructor of the class or interface represented by this Class object.
Constructor[]**getDeclaredConstructors**()	Returns an array of Constructor objects reflecting all the constructors declared by the class represented by this Class object.
Field **getDeclaredField**(String name)	Returns a Field object that reflects the specified declared field of the class or interface represented by this Class object.
Field[]**getDeclaredFields**()	Returns an array of Field objects reflecting all the fields declared by the class or interface represented by this Class object.

Method **getDeclaredMethod** (String name, Class[] parameterTypes)	Returns a Method object that reflects the specified declared method of the class or interface represented by this Class object.
Method[]**getDeclaredMethods**()	Returns an array of Method objects reflecting all the methods declared by the class or interface represented by this Class object.
Class **getDeclaringClass**()	If the class or interface represented by this Class object is a member of another class, returns the Class object representing the class in which it was declared.
Field **getField**(String name)	Returns a Field object that reflects the specified public member field of the class or interface represented by this Class object.
Field[] **getFields**()	Returns an array containing Field objects
Class[] **getInterfaces**()	Determines the interfaces implemented by the class or interface represented by this object.
Method **getMethod**(String name, Class[] parameterTypes)	Returns a Method object that reflects the specified public member method of the class or interface represented by this Class object.
Method[] **getMethods**()	Returns an array containing Method objects reflecting all the public *member* methods of the class or interface represented by this Class object, including those declared by the class or interface and and those inherited from superclasses and superinterfaces.
int **getModifiers**()	Returns the Java language modifiers for this class or interface, encoded in an integer.
String **getName**()	Returns the name of the entity (class, interface, array class, primitive type, or void) represented by this Class object, as a String.
Package **getPackage**()	Gets the package for this class.

Class **getSuperclass**()	Returns the Class representing the superclass of the entity (class, interface, primitive type or void) represented by this Class.
Boolean **isArray**()	Determines if this Class object represents an array class.
String **toString**()	Converts the object to a string.

The forName( ) method

The **forName()** method of **Class** can be used to load the classes at runtime. The static block is executed when the class is loaded. The signature of **forName()** method is

```
public static Class forName(String className)throws ClassNotFoundException
```

A program demonstrate forName() to load the class into memory at runtime.

Bin>edit class1.java

```
class sample
{
   static
   {
         System.out.print("\n Sample class is loaded.");
         System.out.print("\n Static block executed.");
   }
}
class class1
{
   public static void main(String argv[]) throws
                                    ClassNotFoundException
   {
         try
         {
               Class.forName("sample");
         }
         catch(ClassNotFoundException  e)
         {
               System.out.print("\n Error : "+e);
         }
   }
}
```

```
Bin>javac class1.java
Bin>java class1
```

Sample class is loaded.
Static block executed.

Explanation:

In the above program, the statement **Class.forName("sample");** loads the sample class into memory and executes static block therefore we get the output as shown above.

We can also give the path of class file in the **forName()** method as

Class.forName("java.lang.Thread");

The getName() method

The **getName()** method of **Class** returns the name of the entity (class, interface, array class, primitive type, or void) as string. The signature of **getName()** method is.

public String getName()

The getSuperclass() method

The **getSuperclass()** method of Class returns the **Class** object representing the superclass of the invoking object. If the invoking object represents either the Object class, an interface, a primitive type, or void, then null is returned. If this object represents an array class then the **Class** object representing the **Object** class is returned.

public Class getSuperclass()

A program to demonstrate getName() and getSuperclass() of Class

```
Bin>edit class2.java
```

```
class A
{
      int x=0;
}
class B extends A
{
      double y=5.6;
}
class class2
{
      public static void main(String argv[])
      {
```

```
        B objb=new B();
        Class ob=objb.getClass();
        System.out.print("\n class name="+ob.getName());
        ob=ob.getSuperclass();
        System.out.print("\n super class name="+
                                        ob.getName());
    }
}
```

Bin>javac class2.java
Bin>java class2

class name=B
super class name=A

Explanation:

In the above program, class **B** extends **A** therefore class **A** is the super class and class **B** is the subclass. In the main(), **objb** is created to **B** class. In the statement **Class ob=objb.getClass();** the **getClass**() of Object class returns the object of Class type of **objb** and assigns to ob. In the **print(),** the **ob.getName**() returns the class name(**B**). In the statement, **ob=ob.getSuperclass();** the **getSuperclass()** returns the super class **Class** type object of **B** class. In the second **print(),** the **getName()** returns the super class name (**A**).

The newInstance( ) method

The newInstance( ) method of Class creates a new instance of the class represented by invoking Class object. The class is instantiated as if by a new expression with an empty argument list. The class is initialized if it has not already been initialized. The signature of this method is.

public Object newInstance() throws InstantiationException, IllegalAccessException

Bin>edit class3.java

```
class A
{
   int x;
}

class class3
{
```

```
    public static void main(String argv[]) throws
            InstantiationException,IllegalAccessException
    {
            A obj1=new A();
            obj1.x=100;
            Class ob=obj1.getClass();
            A obj2;
            try
            {
                    obj2=(A)ob.newInstance();
                    System.out.print("\n obj1...x=" + obj1.x);
                    System.out.print("\n obj2...x=" + obj2.x);
            }
            catch(Exception e)
            {
                    System.out.print("\n Error :" +e);
            }

    }
}
```

Bin>javac class3.java
Bin>java class3
obj1...x=100
obj2...x=0

Explanation:

In the above program, obj1 is an object created to **class A** and **x** is stored with **100**. In the statement, **Class ob=obj1.getClass();** the **getClass()** returns the object of **Class** type of **obj1** and assigns to **ob.**

In the statement, **obj2=(A)ob.newInstance();** the **newInstance()** method creates a new object(instance) of **ob** class i.e. **A** and the reference of newly created object is typecasted to **A** and assigns to **obj2**.

The **obj1.x** contains **100** where as **obj2.x** contains **0** because **obj2** is not initialized with any values.

The getMethods() method

The **getMethods()** of **Class** returns an array containing **Method** objects reflecting all the public member methods of the class or interface represented by invoking **Class** object, including those declared by the class or interface and those inherited from superclasses and superinterfaces. The elements in the array returned are not sorted and are not in any particular order. This method returns an array of length **0** if this **Class** object represents a class or interface that has no public member methods, or if this **Class** object represents an array class, **primitive type**, or **void**.

public Method[] getMethods()throws SecurityException

Bin>edit methods1.java

```
import java.lang.reflect.Method;
class A
{
   public void method1()
   {
         System.out.print("\n method1...");
   }

   public void method2()
   {
         System.out.print("\n method2...");
   }
}
class m ethods1
{
   public static void main(String argv[])
   {
         A obj1=new A();
         Class ob=obj1.getClass();
         Method m[]=ob.getMethods();
         System.out.print("\n methods of class A are..");
         for( int i=0; i<m.length;i++)
         {
               System.out.print("\n"+m[i].getName());
         }
   }
}
```

```
Bin>javac methods1.java
Bin>java methods1
        methods of class A are..
        method1
        method2
        hashCode
        getClass
        wait
        wait
        wait
        equals
        notify
        notifyAll
        toString
```

In the above output **method1** and **method2** are the methods of class A where as the remaining methods are inherited from the **Object** class which is the supermost class in java.

Math class

The class **Math** contains methods for performing basic numeric operations such as the elementary exponential, logarithm, square root, and trigonometric functions. The **Math** class contains all the methods as public and static therefore these methods can be accessed by using class name. **Math** class contains two constant variables(final variables), the following table shows these constants

Fields	Meaning
static double **E**	The double value that is closer than any other to *e*, the base of the natural logarithms.
static double **PI**	The double value that is closer than any other to *pi*, the ratio of the circumference of a circle to its diameter.

The following table shows some of the methods of **Math** class

Methods	Meaning
static double **abs**(double a)	Returns the absolute value of a double value.
static float **abs**(float a)	Returns the absolute value of a float value.

static int **abs**(int a)	Returns the absolute value of an int value.
static long **abs**(long a)	Returns the absolute value of a long value.
static double **acos**(double a)	Returns the arc cosine of an angle, in the range of 0.0 through *pi*.
static double **asin**(double a)	Returns the arc sine of an angle, in the range of -*pi*/2 through *pi*/2.
static double **atan**(double a)	Returns the arc tangent of an angle, in the range of -*pi*/2 through *pi*/2.
static double **atan2**(double y, double x)	Converts rectangular coordinates (x, y) to polar (r, *theta*).
static double **ceil**(double a)	Returns the smallest (closest to negative infinity) double value that is not less than the argument and is equal to a mathematical integer.
static double **cos**(double a)	Returns the trigonometric cosine of an angle.
static double **exp**(double a)	Returns Euler's number *e* raised to the power of a double value.
static double **floor**(double a)	Returns the largest (closest to positive infinity) double value that is not greater than the argument and is equal to a mathematical integer.
static double **IEEEremainder**(double f1, double f2)	Computes the remainder operation on two arguments as prescribed by the IEEE 754 standard.
static double **log**(double a)	Returns the natural logarithm (base *e*) of a double value.
static double **max**(double a, double b)	Returns the greater of two double values.
static float **max**(float a, float b)	Returns the greater of two float values.
static int **max**(int a, int b)	Returns the greater of two int values.
static long **max**(long a, long b)	Returns the greater of two long values.
static double **min**(double a, double b)	Returns the smaller of two double values.

static float **min**(float a, float b)	Returns the smaller of two float values.
static int **min**(int a, int b)	Returns the smaller of two int values.
static long **min**(long a, long b)	Returns the smaller of two long values.
static double **pow**(double a, double b)	Returns the value of the first argument raised to the power of the second argument.
static double **random**()	Returns a double value with a positive sign, greater than or equal to 0.0 and less than 1.0.
static double **rint**(double a)	Returns the double value that is closest in value to the argument and is equal to a mathematical integer.
static long **round**(double a)	Returns the closest long to the argument.
static int **round**(float a)	Returns the closest int to the argument.
static double **sin**(double a)	Returns the trigonometric sine of an angle.
static double **sqrt**(double a)	Returns the correctly rounded positive square root of a double value.
static double **tan**(double a)	Returns the trigonometric tangent of an angle.
static double **toDegrees**(double angrad)	Converts an angle measured in radians to an approximately equivalent angle measured in degrees.
static double **toRadians**(double angdeg)	Converts an angle measured in degrees to an approximately equivalent angle measured in radians.

The sqrt() method

The sqrt() method of Math class is a public static method which returns the correctly rounded positive square root of a double value.

public static double sqrt(double a)

Bin>edit math1.java

```
class math1
{
      public static void main(String argv[])
      {
            double d=Math.sqrt(49.0);
```

```
            System.out.print("\n d =" +d);
        }
    }
```

```
Bin>javac math1.java
Bin>java math1
    d=7.0
```

Wrapper Classes

Java has eight primitive data types such as **byte**, **short**, **int**, **long**, **float**, **double**, **char** and **boolean**. These data types are keywords in java. Passing these type variables to methods are pass-by-value. There are some situations were we want to see these data types as user-defined types i.e. classes. Java provides user-defined types for primitive types and such classes are known as wrapper classes. Each wrapper class is enclosed with its respective primitive type to store the value. The objects of wrapper classes when passed to methods are pass-by-reference.

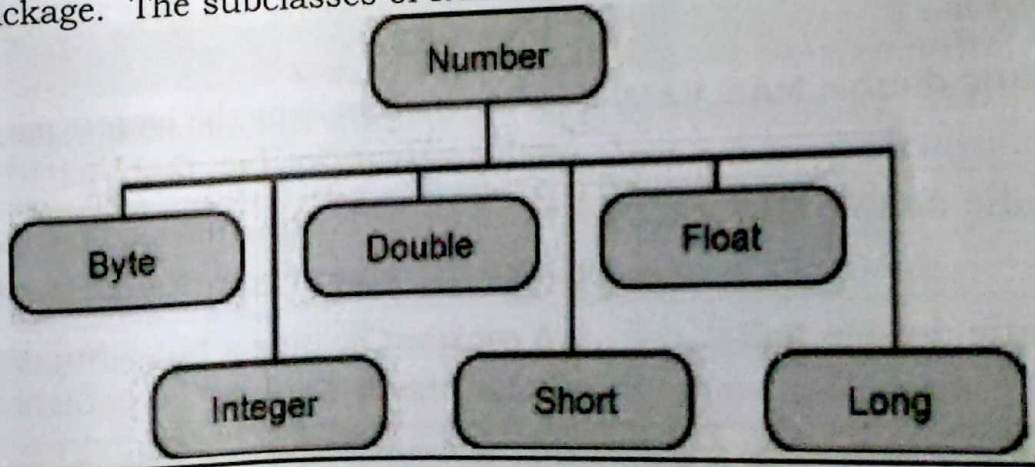
Java provides object wrappers for these primitive types, which helps to convert simple types to the corresponding objects. The objects of wrapper classes are immutable (constant) objects i.e. whose contents cannot be altered. These wrapper classes contain various methods that allow I/O operations and conversions from one form to another. The Collections(pre-defined data structures) in java operates on only objects therefore we have to use these wrapper classes instead of primitive types.

This wrapping is taken care of by the compiler The process is called boxing. So when a primitive is used when an object is required the compiler boxes the primitive type in its wrapper class. Similarly the compiler **unboxes** the object to a primitive as well.

Number

All the wrapper classes (**Integer**, **Long**, **Byte**, **Double**, **Float**, **Short**) are subclasses of the abstract class **Number**. The **Number** is part of the **java.lang** package. The subclasses of **Number** are:

Byte
Short
Integer
Long
Float
Double



The **Number** abstract class contains abstract methods that return the value of the object in each of the different number formats. These methods are inherited into all its subclasses and they are:

Methods	Meaning
byte **byteValue**()	Returns the value of the specified number as a byte.
abstract double **doubleValue**()	Returns the value of the specified number as a double.
abstract float **floatValue**()	Returns the value of the specified number as a float.
abstract int **intValue**()	Returns the value of the specified number as an int.
abstract long **longValue**()	Returns the value of the specified number as a long.
short **shortValue**()	Returns the value of the specified number as a short.

All the wrapper classes have two constants, **MIN_VALUE** and **MAX_VALUE** representing minimum value and maximum value of the respective type.

Double

The Double class wraps a value of the primitive type double in an object. An object of type Double contains a single field whose type is double.

In addition, this class provides several methods for converting a double to a String and a String to a double, as well as other constants and methods useful when dealing with a double.

Following table shows constants of **Double** class.

Fields	Meaning
static double **MAX_VALUE**	A constant holding the largest positive finite value of type double, $(2-2^{-52})\cdot 2^{1023}$.
static double **MIN_VALUE**	A constant holding the smallest positive nonzero value of type double, 2^{-1074}.
static double **NaN**	A constant holding a Not-a-Number (NaN) value of type double.

static double **NEGATIVE_INFINITY**	A constant holding the negative infinity of type double.
static double **POSITIVE_INFINITY**	A constant holding the positive infinity of type double.

Following table shows constru1ctors of **Double** class.

Constructors	**Meaning**
Double(**double** value)	Constructs a newly allocated Double object that represents the primitive double argument.
Double(**String** s)	Constructs a newly allocated Double object that represents the floating-point value of type double represented by the string.

Following table shows methods of **Double** class.

Methods	**Meaning**
byte **byteValue**()	Returns the value of this Double as a byte (by casting to a byte).
static int **compare**(double d1, double d2)	Compares the two specified double values.
int **compareTo**(Double anotherDouble)	Compares two Double objects numerically.
int **compareTo**(Object o)	Compares this Double object to another object.
static long **doubleToLongBits**(double v alue)	Returns a representation of the specified floating-point value according to the IEEE 754 floating-point "double format" bit layout.
static long **doubleToRawLongBits**(dou ble value)	Returns a representation of the specified floating-point value according to the IEEE 754 floating-point "double format" bit layout, preserving Not-a-Number (NaN) values.
double **doubleValue**()	Returns the double value of this Double object.
boolean **equals**(Object obj)	Compares this object against the specified object.

float **floatValue**()	Returns the float value of this Double object.
int **hashCode**()	Returns a hash code for this Double object.
int **intValue**()	Returns the value of this Double as an int (by casting to type int).
boolean **isInfinite**()	Returns true if this Double value is infinitely large in magnitude, false otherwise.
static boolean **isInfinite**(double v)	Returns true if the specified number is infinitely large in magnitude, false otherwise.
boolean **isNaN**()	Returns true if this Double value is a Not-a-Number (NaN), false otherwise.
static boolean **isNaN**(double v)	Returns true if the specified number is a Not-a-Number (NaN) value, false otherwise.
static double **longBitsToDouble**(long bits)	Returns the double value corresponding to a given bit representation.
long **longValue**()	Returns the value of this Double as a long (by casting to type long).
long **longValue**()	Returns the value of this Double as a long (by casting to type long).
static double **parseDouble**(String s)	Returns a new double initialized to the value represented by the specified String, as performed by the valueOf method of class Double.
short **shortValue**()	Returns the value of this Double as a short (by casting to a short).
String **toString**()	Returns a string representation of this Double object.
static String **toString**(double d)	Returns a string representation of the double argument.
static Double **valueOf**(String s)	Returns a Double object holding the double value represented by the argument string s.

A program to demonstrate Double class

Bin>edit double1.java

```
class sample
{
      public void show(Double ob)
      {
            System.out.print("\n ob =" + ob);
            System.out.print("\n ob.doubleValue()="+
                                    ob.doubleValue());
      }
}
class double1
{
      public static void main(String argv[])
      {
            Double d1=new Double(10.5);
            Double d2=new Double("5.6");

            System.out.print("\n d1.doubleValue() =" +
                                    d1.doubleValue());
            System.out.print("\n d1.floatValue()=" +
                                    d1.floatValue());
            System.out.print("\n d1.intValue()=" +
                                    d1.intValue());
            System.out.print("\n d2.doubleValue() =" +
                                     d2.doubleValue());
            sample obj=new sample();
            obj.show(d1);
       }
}
```

Bin>javac double1.java

Bin>java double1

```
d1.doubleValue() =10.5
d1.floatValue()=10.5
d1.intValue()=10
d2.doubleValue() =5.6
ob =10.5
ob.doubleValue()=10.5
```

Float

The Float class wraps a value of primitive type float in an object. An object of type **Float** contains a single field whose type is float.

In addition, this class provides several methods for converting a float to a **String** and a **String** to a float, as well as other constants, constructors and methods useful when dealing with a float.

Following table shows constants of **Float** class.

Fields	Meaning
static float **MAX_VALUE**	A constant holding the largest positive finite value of type float, $(2-2^{-23})\cdot 2^{127}$.
static float **MIN_VALUE**	A constant holding the smallest positive nonzero value of type float, 2^{-149}.
static float **NaN**	A constant holding a Not-a-Number (NaN) value of type float.
static float **NEGATIVE_INFINITY**	A constant holding the negative infinity of type float.
static float **POSITIVE_INFINITY**	A constant holding the positive infinity of type float.
static int **SIZE**	The number of bits used to represent a float value.
static Class<Float> **TYPE**	The Class instance representing the primitive type float.

Following table shows constructors of **Float** class.

Constructors	Meaning
Float(double value)	Constructs a newly allocated Float object that represents the argument converted to type float.
Float(float value)	Constructs a newly allocated Float object that represents the primitive float argument.
Float(String s)	Constructs a newly allocated Float object that represents the floating-point value of type float represented by the string.

Methods	Meaning
byte **byteValue**()	Returns the value of this Float as a byte (by casting to a byte).
static int **compare**(float f1, float f2)	Compares the two specified float values.
int **compareTo**(Float anotherFloat)	Compares two Float objects numerically.
Double **doubleValue**()	Returns the double value of this Float object.
Boolean **equals**(Object obj)	Compares this object against the specified object.
static int **floatToIntBits**(float value)	Returns a representation of the specified floating-point value according to the IEEE 754 floating-point "single format" bit layout.
static int **floatToRawIntBits**(float value)	Returns a representation of the specified floating-point value according to the IEEE 754 floating-point "single format" bit layout, preserving Not-a-Number (NaN) values.
float **floatValue**()	Returns the float value of this Float object.
int **hashCode**()	Returns a hash code for this Float object.
static float **intBitsToFloat**(int bits)	returns the float value corresponding to a given bit representation.
int **intValue**()	Returns the value of this Float as an int (by casting to type int).
Boolean **isInfinite**()	Returns true if this Float value is infinitely large in magnitude, false otherwise.
static Boolean **isInfinite**(float v)	Returns true if the specified number is infinitely large in magnitude, false otherwise.
Boolean **isNaN**()	Returns true if this Float value is a Not-a-Number (NaN), false otherwise.
static Boolean **isNaN**(float v)	Returns true if the specified number is a Not-a-Number (NaN) value, false otherwise.
long **longValue**()	Returns value of this Float as a long (by casting to type long).

static float **parseFloat**(String s)	Returns a new float initialized to the value represented by the specified String, as performed by the valueOf method of class Float.
short **shortValue**()	Returns the value of this Float as a short (by casting to a short).
static String **toHexString**(float f)	Returns a hexadecimal string representation of the float argument.
String **toString**()	Returns a string representation of this Float
static String **toString**(float f)	Returns a string representation of the float argument.
static Float **valueOf**(float f)	Returns a Float instance representing the specified float value.
static Float **valueOf**(String s)	Returns a Float object holding the float value represented by the argument string s.

A program to demonstrate Float class

Bin> edit float1.java

```
class float1
{
      public static void main(String argv[])
      {
            Float f1=new Float(10.5);
            Float f2=new Float("5.6");
            System.out.print("\n f1.doubleValue() =" +
                  f1.doubleValue());
            System.out.print("\n f1.floatValue()=" +
                  f1.floatValue());
            System.out.print("\n f1.intValue()=" +
                  f1.intValue());
            System.out.print("\n f2 = " + f2);
      }
}
```

Bin>javac float1.java

Bin>java float1

f1.doubleValue() =10.5

f1.floatValue()=10.5

f1.intValue()=10

f2=5.6

Integer

The Integer class wraps a value of the primitive type **int** in an object. An object of type **Integer** contains a single field whose type is int.

In addition, this class provides several methods for converting an **int** to a **String** and a **String** to an **int**, as well as other constants and methods useful when dealing with an int.

Following table shows constants of **Integer** class.

Fields	Meaning
static int **MAX_VALUE**	A constant holding the maximum value an int can have, $2^{31}-1$.
static int **MIN_VALUE**	A constant holding the minimum value an int can have, -2^{31}.
static int **SIZE**	The number of bits used to represent an int value in two's complement binary form.
static Class<Integer> **TYPE**	The Class instance representing the primitive type int.

Following table shows constructors of **Integer** class.

Constructors	Meaning
Integer(int value)	Constructs a newly allocated Integer object that represents the specified int value.
Integer(String s)	Constructs a newly allocated Integer object that represents the int value indicated by the String parameter.

Following table shows methods of **Integer** class.

Methods	Meaning
static int **bitCount**(int i)	Returns the number of one-bits in the two's complement binary representation of the specified int value.
byte **byteValue**()	Returns the value of this Integer as a byte.

int **compareTo**(Integer anotherInteger)	Compares two Integer objects numerically.
static Integer **decode**(String nm)	Decodes a String into an Integer.
double **doubleValue**()	Returns the value of this Integer as a double.
boolean **equals**(Object obj)	Compares this object to the specified object.
float **floatValue**()	Returns the value of this Integer as a float.
static Integer **getInteger**(String nm)	Determines the integer value of the system property with the specified name.
static Integer **getInteger**(String nm, int val)	Determines the integer value of the system property with the specified name.
int **intValue**()	Returns the value of this Integer as an int.
long **longValue**()	Returns the value of this Integer as a long.
static int **parseInt**(String s)	Parses the string argument as a signed decimal integer.
static int **parseInt**(String s, int radix)	Parses the string argument as a signed integer in the radix specified by the second argument.
short **shortValue**()	Returns the value of this Integer as a short.
static String **toBinaryString**(int i)	Returns a string representation of the integer argument as an unsigned integer in base 2.
static String **toHexString**(int i)	Returns a string representation of the integer argument as an unsigned integer in base 16.
static String **toOctalString**(int i)	Returns a string representation of the integer argument as an unsigned integer in base 8.
String **toString**()	Returns a String object representing this Integer's value.

static String **toString**(int i)	Returns a String object representing the specified integer.
static String **toString**(int i, int radix)	Returns a string representation of the first argument in the radix specified by the second argument.
static Integer **valueOf**(int i)	Returns a Integer instance representing the specified int value.
static Integer **valueOf**(String s)	Returns an Integer object holding the value of the specified String.
static Integer **valueOf**(String s, int radix)	Returns an Integer object holding the value extracted from the specified String when parsed with the radix given by the second argument.

MIN_VALUE

A constant holding the minimum value an int can have, -2^{31}.

public static final int **MIN_VALUE**

MAX_VALUE

A constant holding the maximum value an int can have, $2^{31}-1$.

public static final int **MAX_VALUE**

TYPE

The Class instance representing the primitive type int.

public static final Class<Integer> **TYPE**

SIZE

The number of bits used to represent an int value in two's complement binary form.

public static final int **SIZE**

A program to demonstrate Integer class

Bin>edit integer1.java

```
class integer1
{
   public static void main(String argv[])
   {
```

```
        Integer i1=new Integer(105);
        Integer i2=new Integer("223");
        System.out.print("\n i1.doubleValue() =" +
              i1.doubleValue());
        System.out.print("\n i1.floatValue()=" +
              i1.floatValue());
        System.out.print("\n i1.intValue()=" +
              i1.intValue());
        System.out.print("\n i2 = " + i2);
    }
}
```

Bin>javac float1.java
Bin>java float1
i1.doubleValue() =105.0
i1.floatValue()=105.0
i1.intValue()=105
i2 = 223

Byte

The Byte class wraps a value of primitive type byte in an object. An object of type **Byte** contains a single field whose type is byte.

In addition, this class provides several methods for converting a **byte** to a **String** and a **String** to a **byte**, as well as other constants and methods useful when dealing with a byte.

Following table shows constants of **Byte** class.

Fields	Meaning
static byte **MAX_VALUE**	A constant holding the maximum value a byte can have, 2^7-1.
static byte **MIN_VALUE**	A constant holding the minimum value a byte can have, -2^7.
static Class **TYPE**	The Class instance representing the primitive type byte.

Following table shows constructors of **Byte** class.

Constructors	Meaning
Byte(byte value)	Constructs a newly allocated Byte object that represents the specified byte value.
Byte(String s)	Constructs a newly allocated Byte object that represents the byte value indicated by the String parameter.

Following table shows methods of **Byte** class.

Methods	Meaning
byte **byteValue**()	Returns the value of this Byte as a byte.
int **compareTo**(Byte anotherByte)	Compares two Byte objects numerically.
int **compareTo**(Object o)	Compares this Byte object to another object.
static Byte **decode**(String nm)	Decodes a String into a Byte.
double **doubleValue**()	Returns the value of this Byte as a double.
Boolean **equals**(Object obj)	Compares this object to the specified object.
float **floatValue**()	Returns the value of this Byte as a float.
int **intValue**()	Returns the value of this Byte as an int.
long **longValue**()	Returns the value of this Byte as a long.
static byte **parseByte**(String s)	Parses the string argument as a signed decimal byte.
static byte **parseByte**(String s, int radix)	Parses the string argument as a signed byte in the radix specified by the second argument.
Short **shortValue**()	Returns the value of this Byte as a short.
String **toString**()	Returns a String object representing this Byte's value.
static String **toString**(byte b)	Returns a new String object representing the specified byte.
static Byte **valueOf**(String s)	Returns a Byte object holding the value given by the specified String.

static Byte **valueOf**(String s, int radix)	Returns a Byte object holding the value extracted from the specified String when parsed with the radix given by the second argument.

MIN_VALUE

A constant holding the minimum value a byte can have, -2^7.

public static final byte **MIN_VALUE**

MAX_VALUE

A constant holding the maximum value a byte can have, 2^7-1.

public static final byte **MAX_VALUE**

Short

The **Short** class wraps a value of primitive type short in an object. An object of type Short contains a single field whose type is short.

In addition, this class provides several methods for converting a **short** to a **String** and a **String** to a **short**, as well as other constants and methods useful when dealing with a short.

Following table shows constants of **Short** class.

Fields	Meaning
static short **MAX_VALUE**	A constant holding the maximum value a short can have, $2^{15}-1$.
static short **MIN_VALUE**	A constant holding the minimum value a short can have, -2^{15}.
static int **SIZE**	The number of bits used to represent a short value in two's complement binary form.
static Class<Short> **TYPE**	The Class instance representing the primitive type short.

Following table shows constructors of **Short** class.

Constructors	Meaning
Short(short value)	Constructs a newly allocated Short object that represents the specified short value.
Short(String s)	Constructs a newly allocated Short object that represents the short value indicated by the String parameter.

Following table shows methods of **Short** class.

Methods	Meaning
byte **byteValue**()	Returns the value of this Short as a byte.
int **compareTo**(Short anotherShort)	Compares two Short objects numerically.
static Short **decode**(String nm)	Decodes a String into a Short.
Double **doubleValue**()	Returns the value of this Short as a double.
Boolean **equals**(Object obj)	Compares this object to the specified object.
float **floatValue**()	Returns the value of this Short as a float.
int **intValue**()	Returns the value of this Short as an int.
long **longValue**()	Returns the value of this Short as a long.
static short **parseShort**(String s)	Parses the string argument as a signed decimal short.
static short **parseShort**(String s, int radix)	Parses the string argument as a signed short in the radix specified by the second argument.
static short **reverseBytes**(short i)	Returns the value obtained by reversing the order of the bytes in the two's complement representation of the specified short value.
short **shortValue**()	Returns the value of this Short as a short.
String **toString**()	Returns a String object representing this Short's value.
static String **toString**(short s)	Returns a new String object representing the specified short.
static Short **valueOf**(short s)	Returns a Short instance representing the specified short value.
static Short **valueOf**(String s)	Returns a Short object holding the value given by the specified String.
static Short **valueOf**(String s, int radix)	Returns a Short object holding the value extracted from the specified String when parsed with the radix given by the second argument.

Long

The Long class wraps a value of the primitive type long in an object. An object of type Long contains a single field whose type is long.

In addition, this class provides several methods for converting a **long** to a **String** and a **String** to a **long**, as well as other constants and methods useful when dealing with a long.

Following table shows constants of **Long** class.

Fields	Meaning
static long **MAX_VALUE**	A constant holding the maximum value a long can have, $2^{63}-1$.
static long **MIN_VALUE**	A constant holding the minimum value a long can have, -2^{63}.
static int **SIZE**	The number of bits used to represent a long value in two's complement binary form.
static Class<Long> **TYPE**	The Class instance representing the primitive type long.

Following table shows constructors of **Long** class.

Constructors	Meaning
Long(long value)	Constructs a newly allocated Long object that represents the specified long argument.
Long(String s)	Constructs a newly allocated Long object that represents the long value indicated by the String parameter.

Following table shows methods of **Long** class.

Methods	Meaning
static int **bitCount**(long i)	Returns the number of one-bits in the two's complement binary representation of the specified long value.
byte **byteValue**()	Returns the value of this Long as a byte.
int **compareTo**(Long anotherLong)	Compares two Long objects numerically.
static Long **decode**(String nm)	Decodes a String into a Long.

Double **doubleValue**()	Returns the value of this Long as a double.
Boolean	Compares this object to the specified object.
float **floatValue**()	Returns the value of this Long as a float.
static Long **getLong**(String nm)	Determines the long value of the system property with the specified name.
static Long **getLong**(String nm, long val)	Determines the long value of the system property with the specified name.
static Long **getLong**(String nm, Long val)	Returns the long value of the system property with the specified name.
int **intValue**()	Returns the value of this Long as an int.
long **longValue**()	Returns the value of this Long as a long value.
static long **parseLong**(String s)	Parses the string argument as a signed decimal long.
static long **parseLong**(String s, int radix)	Parses the string argument as a signed long in the radix specified by the second argument.
static long **reverse**(long i)	Returns the value obtained by reversing the order of the bits in the two's complement binary representation of the specified long value.
static long **reverseBytes**(long i)	Returns the value obtained by reversing the order of the bytes in the two's complement representation of the specified long value.
short **shortValue**()	Returns the value of this Long as a short.
static int **signum**(long i)	Returns the signum function of the specified long value.
static String **toBinaryString**(long i)	Returns a string representation of the long argument as an unsigned integer in base 2.
static String **toHexString**(long i)	Returns a string representation of the long argument as an unsigned integer in base 16.

static String **toOctalString**(long i)	Returns a string representation of the long argument as an unsigned integer in base 8.
String **toString**()	Returns a String object representing this Long's value.
static String **toString**(long i)	Returns a String object representing the specified long.
static String **toString**(long i, int radix)	Returns a string representation of the first argument in the radix specified by the second argument.
static Long **valueOf**(long l)	Returns a Long instance representing the specified long value.
static Long **valueOf**(String s)	Returns a Long object holding the value of the specified String.
static Long **valueOf**(String s, int radix)	Returns a Long object holding the value extracted from the specified String when parsed with the radix given by the second argument.