Packages

Package Definition

A package is a group of related classes. A package is used to restrict access to a class and to create namespaces. If all reasonable class names are already used, then we can create a new package (new namespace) and reuse the class names again. For example a package mypackage1 can contain a class named MyClass and another package mypackage2 can also contain a class named MyClass.

Defining or Creating a Package

A package can be created by including the package statement as first line in a Java source file. Syntax for creating or defining a package is given below:

package mypackage;

In the above syntax, package is a keyword and mypackage is the name of our package. All the classes that follow the package statement are considered to be a part of the package mypackage.

Multiple source files can include the package statement with the same package name. Packages are maintained as regular directories (folders) in the machine. For example, we have to create a folder named mypackage and store the .class files in that folder.

Following example demonstrates creating a package:

```
ClassA.java
package mypackage;
public class ClassA
{
        public void methodA()
        {
                System.out.println("This is methodA in Class A");
        }
}
ClassB.java
package mypackage;

public class ClassB
{
        public void methodB()
        {
                System.out.println("This is methodB in Class B");
        }
}
```

In the above example, both ClassA and ClassB belong to the same package mypackage. Remember that the package creation statement should be the first line in the source file.

We can also create a hierarchy of packages as shown below:

package pack1.pack2.pack3;

Remember that packages are normal folders in the file system. So, pack3 is a sub folder in pack2 and pack2 is a sub folder in pack1.

Java Packages and CLASSPATH

It is not mandatory that the driver program (main program) and the package(s) to be at the same location. So how does JVM know the path to the package(s)? There are three options:

1. Placing the package in the current working directory.
2. Specifying the package(s) path using CLASSPATH environment variable.
3. Using the -classpath or -cp options with javac and java commands.

If no package is specified, by default all the classes are placed in a default package. That is why no errors are shown even if we don't use a package statement.

By default Java compiler and JVM searches the current working directory (option 1) for specified package(s) like mypackage. Let's assume that our package mypackage is stored at following location:

```
D:\packages\mypackage
```

Then we can set the CLASSPATH (option 2) environment variable (in command prompt) to the location of the package as shown below:

```
set CLASSPATH = .;D:\packages
```

The dot (.) before the path specifies the current working directory and multiple paths can be separated using semi-colon (;).

We can also use -classpath or -cp options (option 3) with javac and java commands as shown below:

```
javac  -classpath  .;D:\packages  Driver.java
```

or

```
javac  -cp  .;D:\packages  Driver.java
```

In the above example, Driver.java is our main program which utilizes the classes in the package mypackage.

Importing or Using Packages

There are multiple ways in which we can import or use a package. They are as follows:

1. Importing all the classes in a package.
2. Importing only necessary classes in a package.
3. Specifying the fully qualified name of the class.

First way is to import all the classes in a package using the import statement. The import statement is placed after the package statement if any and before all class declarations. We can import all the classes in a package as shown below:

```
import  mypackage.*;
```

* in the above line denotes all classes within the package mypackage. Now you are free to directly use all the classes within that package. A program which demonstrates importing all classes in a package is given below:

```
import mypackage.*;
public class Driver
{
        public static void main(String[] args)
        {
                ClassA obj1 = new ClassA();
                obj1.methodA();
                ClassB obj2 = new ClassB();
                obj2.methodB();
        }
}
```

If you want to use only one or two classes in a package, the second way is to specify the class names instead of * as shown below:

```
import  mypackage.ClassA;
import  mypackage.ClassB;
```

A program which demonstrates importing specific class from a package is given below:

```
import mypackage.ClassA;
public class Driver
{
        public static void main(String[] args)
        {
                ClassA obj1 = new ClassA();
                obj1.methodA();
        }
}
```

Suppose if two packages contain a class with same name, then it will lead to compile-time errors. To avoid this, we have to use the fully qualified name of the class. A fully qualified name of the class refers to the name of the class preceded by the package name. An example that demonstrates a fully qualified name is given below:

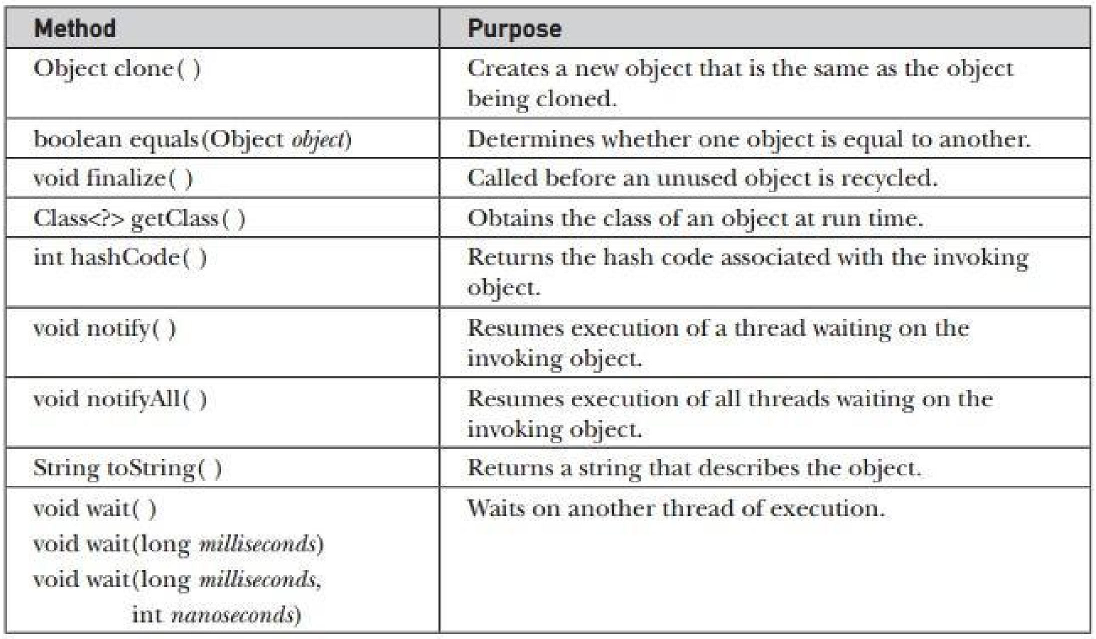
```
public class Driver
{
        public static void main(String[] args)
        {
                mypackage.ClassA obj1 = new mypackage.ClassA();
                obj1.methodA();
        }
}
```

java.lang package

java.lang is a special package, as it is imported by default in all the classes that we create. There is no need to explicitly import the lang package.It contains that form the basic building blocks of java

1. java.lang.Object class

Object class is present in **java.lang** package. Every class in Java is directly or indirectly derived from the **Object** class. If a Class does not extend any other class then it is direct child class of **Object** and if extends other class then it is an indirectly derived. Therefore the Object class methods are available to all Java classes. Hence Object class acts as a root of inheritance hierarchy in any Java Program.



Method	Purpose
Object clone( )	Creates a new object that is the same as the object being cloned.
boolean equals(Object *object*)	Determines whether one object is equal to another.
void finalize( )	Called before an unused object is recycled.
Class<?> getClass( )	Obtains the class of an object at run time.
int hashCode( )	Returns the hash code associated with the invoking object.
void notify( )	Resumes execution of a thread waiting on the invoking object.
void notifyAll( )	Resumes execution of all threads waiting on the invoking object.
String toString( )	Returns a string that describes the object.
void wait( ) void wait(long *milliseconds*) void wait(long *milliseconds*, int *nanoseconds*)	Waits on another thread of execution.

Example:

Illustrates toString( ) method

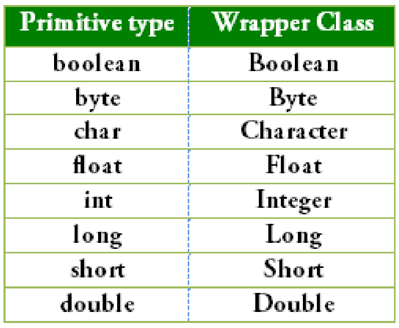
```
class Demo
{
        public String toString( )                          //this method is overriden
        {
                Return "My demo object created";
        }
        public static void main(String args[])
        {
                System.out.println(new Demo());
        }
}
```

2. Java Wrapper classes

A Wrapper class is a class whose object wraps or contains primitive data types. When we create an object to a wrapper class, it contains a field and in this field, we can store primitive data types. In other words, we can wrap a primitive value into a wrapper class object.

Need of Wrapper Classes

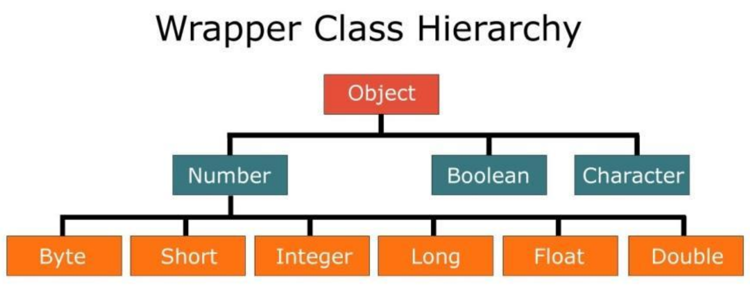
- They convert primitive data types into objects. Objects are needed if we wish to modify the arguments passed into a method (because primitive types are passed by value).
- The classes in java.util package handles only objects and hence wrapper classes help in this case also.



Primitive type	Wrapper Class
boolean	Boolean
byte	Byte
char	Character
float	Float
int	Integer
long	Long
short	Short
double	Double

- Data structures in the Collection framework, such as ArrayList and Vector, store only objects (reference types) and not primitive types.
- An object is needed to support synchronization in multithreading.

The eight classes of the *java.lang* package are known as wrapper classes in Java. The list of eight wrapper classes are given above:



Autoboxing

The automatic conversion of primitive data type into its corresponding wrapper class is known as autoboxing, for example, byte to Byte, char to Character, int to Integer, long to Long, float to Float, boolean to Boolean, double to Double, and short to Short.

Since Java 5, we do not need to use the valueOf() method of wrapper classes to convert the primitive into objects.

Wrapper class Example: Primitive to Wrapper

```
//Java program to convert primitive into objects -Autoboxing example of int to Integer
public class WrapperExample1
{
        public static void main(String args[])
        {
          //Converting int into Integer
          int a=20;
          Integer i=Integer.valueOf(a);//converting int into Integer explicitly
          Integer j=a;//autoboxing, now compiler will write Integer.valueOf(a) internally
          System.out.println(a+" "+i+" "+j);
        }
}
```

Unboxing

The automatic conversion of wrapper type into its corresponding primitive type is known as unboxing. It is the reverse process of autoboxing. Since Java 5, we do not need to use the intValue() method of wrapper classes to convert the wrapper type into primitives.

Wrapper class Example: Wrapper to Primitive

```
//Java program to convert object into primitives
//Unboxing example of Integer to int
```

```
public class WrapperExample2{
public static void main(String args[]){
        //Converting Integer to int
        Integer a=new Integer(3);
        int i=a.intValue();//converting Integer to int explicitly
        int j=a;//unboxing, now compiler will write a.intValue() internally
        System.out.println(a+" "+i+" "+j);
}}
```