PACKAGE INTRODUCTION

Files in one directory (or package) would have special functionality from those of another directory. For example, files in java.io package do something related to I/O, but files in java.net package give us the way to deal with the Network, meaning that this directory keeps files related to the presentation part of the application. Packaging also helps us to avoid class name collision when we use the same class name as that of others.
Simple package designed to include user interactivity. The package enables beginner programmers to build programs with GUI-like interactivity while maintaining good design principles. An advantage of this package is that it is easy to implement using the classes. Therefore, it can be used as a case study to demonstrate Java features.

How to create a package

Suppose we have a file called HelloWorld.java, and we want to put this file in a package **world**. First thing we have to do is to specify the keyword **package** with the name of the package we want to use (**world**in our case) on top of our source file, before the code that defines the real classes in the package, as shown in our HelloWorld class below:

```
// only comment can be here
package world;

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

One thing you must do after creating a package for the class is to create nested subdirectories to represent package hierachy of the class. In our case, we have the **world** package, which requires only one directory. So, we create a directory **world** and put our HelloWorld.java into it.

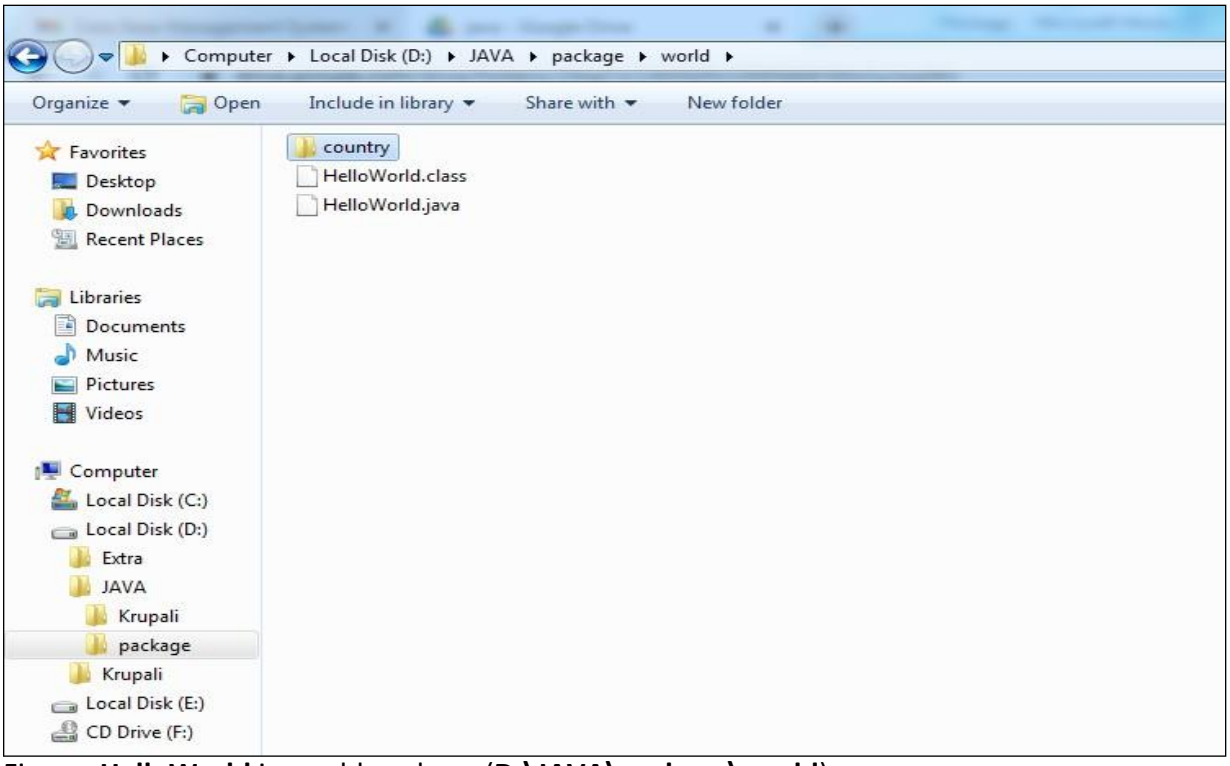

Figure: **HelloWorld** in world package (**D:\JAVA\package\world**)
That's it!!! Right now we have HelloWorld class inside world package. Next, we have to introduce the world package into our **CLASSPATH**.

Setting up the CLASSPATH

From figure 2 we put the package world under d:. So we just set our CLASSPATH as:
set CLASSPATH=.;D:\;
We set the CLASSPATH to point to 2 places, . (dot) and C:\ directory.

Note: If you used to play around with DOS or UNIX, you may be familiar with . (dot) and .. (dot dot). We use . as an alias for the current directory and .. for the parent directory. In our CLASSPATH we include this . for convenient reason. Java will find our class file not only from C: directory but from the current directory as well. Also, we use ; (semicolon) to separate the directory location in case we keep class files in many places.
When compiling HelloWorld class, we just go to the world directory and type the command:
D:\JAVA\package\world \javac HelloWorld.java
If you try to run this HelloWorld using **java HelloWorld**, you will get the following error:
D:\JAVA\package\world >java HelloWorld
Exception in thread "main" java.lang.NoClassDefFoundError: HelloWorld (wrong name:

world/HelloWorld)
    at java.lang.ClassLoader.defineClass0(Native Method)
    at java.lang.ClassLoader.defineClass(ClassLoader.java:442)
    at java.security.SecureClassLoader.defineClass(SecureClassLoader.java:101)
    at java.net.URLClassLoader.defineClass(URLClassLoader.java:248)
    at java.net.URLClassLoader.access$1(URLClassLoader.java:216)
    at java.net.URLClassLoader$1.run(URLClassLoader.java:197)
    at java.security.AccessController.doPrivileged(Native Method)
    at java.net.URLClassLoader.findClass(URLClassLoader.java:191)
    at java.lang.ClassLoader.loadClass(ClassLoader.java:290)
    at sun.misc.Launcher$AppClassLoader.loadClass(Launcher.java:286)
    at java.lang.ClassLoader.loadClass(ClassLoader.java:247)

The reason is right now the HelloWorld class belongs to the package world. If we want to run it, we have to tell JVM about its **fully-qualified class name** (world.HelloWorld) instead of its plain class name(HelloWorld).

D:\JAVA\package\world >java world.HelloWorld
D:\JAVA\package\world >Hello World

Note: **fully-qualified class name** is the name of the java class that includes its package name

Subpackage (package inside another package)

Assume we have another file called **HelloCountry.java**. We want to store it in a subpackage **"country"**, which stays inside package **world**. The HelloCountry class should look something like this:

```
package world.country;

public class HelloCountry {
  private String Name = "Jump to Learn";
  public getName() {
    return Name;
  }
  public setNAme(String SName) {
    this. SName = SName;
  }
}
```

If we store the package world under C: as before, the **HelloCountry.java** would be **D:\JAVA\package \world\country\ HelloCountry.java** as shown in Figure 4 below:

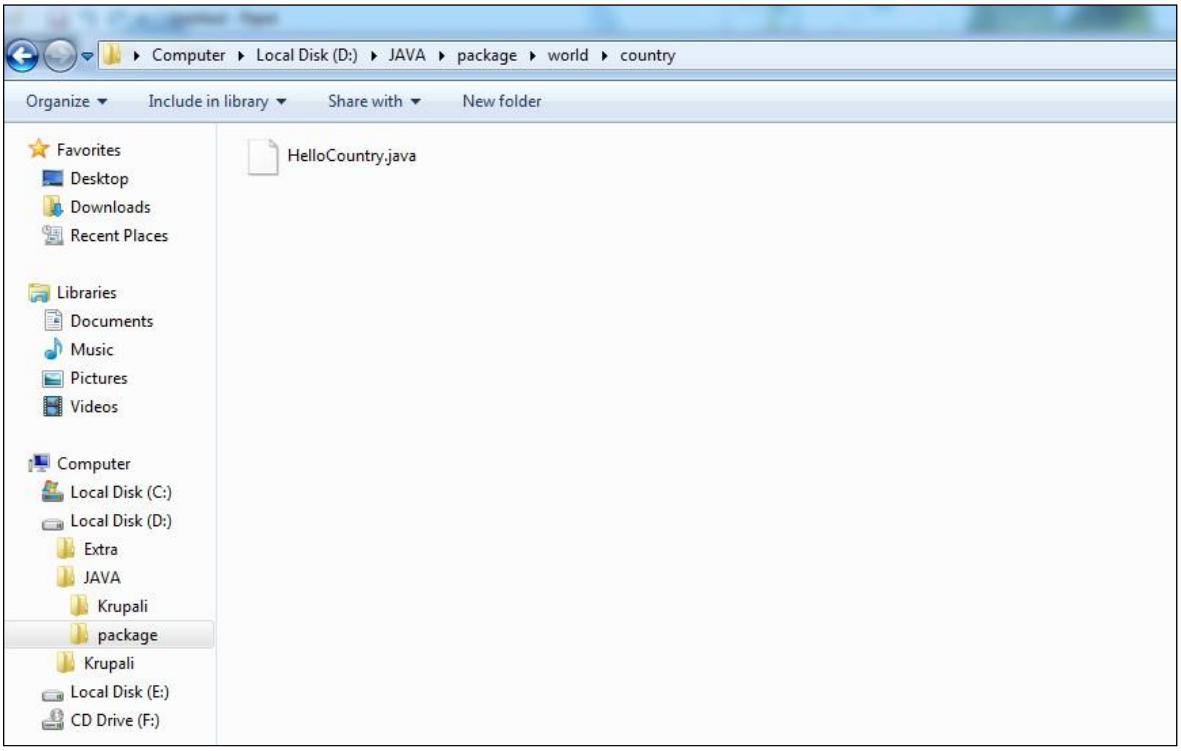

Figure: **HelloCountry** in **world.country** package

Although we add a subpackage under package world, we still don't have to change anything in our CLASSPATH. However, when we want to reference to the HelloCountry class, we have to use world.country.HelloCountry as its fully-qualified class name.

How to use package

There are 2 ways in order to use the public classes stored in package.

1. Declare the fully-qualified class name. For example,

```
/*
world.HelloWorld HW  = new world.HelloWorld();
world.country. HelloCountry HC = new world.country.HelloCountry ();
String Name = HC.getName();
*/
```

2) Use an "import" keyword:

```
import world.*;  // we can call any public classes inside the world package
import world.country.*;  // we can call any public classes inside the world.country package
import java.util.*;  // import all public classes from java.util package
import java.util.Hashtable; // import only Hashtable class (not all classes in java.util package)
```

Thus, the code that we use to call the HelloWorld and HelloCountry class should be

```
/*
  HelloWorld HW = new HelloWorld(); // don't have to explicitly specify world.HelloWorld anymore
```

HelloCountry HC = new HelloCountry (); // don't have to explicitly specify world.country. HelloCountry anymore
*/
Note that we can call public classes stored in the package level we do the import only. We can't use any classes that belong to the subpackage of the package we import. For example, if we import package world, we can use only the HelloWorld class, but not the HelloCountry class.

Characteristics of packages

Packages are stored in hierarchical manner and are explicitly imported into new classes definition.
The package is both naming and visibility control mechanism.
If a source file doesn't contain a package statement, its classes and interfaces are placed in a default package.
Package statement must appear as the first statement.
Two different packages can declare classes have same, by providing this way JAVA provides way for partitioning the class name space into more manageable chunks.
We can define classes inside package that are not accessible by code outside that package.
We can also define class members that are only exposed to other members of the same package.

Benefits of Package

The classes contained in the packages of other programs can be easily reused.
In packages, classes can be unique compared with classes in other packages. i.e. two classes in two different packages can have same name.
Packages can hide classes that are meant to internal use only from other programs or packages.
Packages also provide a way for separating "design" and "coding". 1st we can design classes and decide their responsibility and then we can implement the java code needed for the methods. It is possible to change the implementation of any method without affecting the rest of the design.

Hierarchy of packages

One can create hierarchy of packages. To do this simply separate each package name from the one above it by use of a period.
Syntax: package pkg1[.pkg2[.pkg3]];
CLASSPATH variable
CLASSPATH is an environment variable that determines where the JDK tools such as the JAVA compiler and interpreter search for a .class file. It contains an ordered sequence of directories as well as .jar and .zip files. It can be set as:
set classpath=.;c:\\project1;c:\\project2
echo %classpath% displays the current settings of the classpath variable.
Example:

```
package MyPack;
class Balance
{
    String name;
    double bal;
    Balance(String n, double b)
    {
            name=n;
            bal=b;
    }
    void show()
    {
            If(bal<0)
            {
                    System.out.println("-> ");
                    System.out.println(name+":Rs."+bal);
            }
    }
}
class AccountBalance
{
    public static void main(String args[])
    {
            Balance current[]=new Balance[5];
            Current[0]=new Balance["liza",25000.02];
            Current[1]=new Balance["liza",17000.00];
Current[2]=new Balance["pushpendu",12000.00];
            for(i=0;i<3;i++)
            current[i].show();
    }
}
```

Package Access Modifier

As you will see, Java provides many levels of protection to allow fine-grained control over the visibility of variables and methods within classes, subclasses, and packages. Packages act as containers for classes and other subordinate packages.

Java addresses four categories of visibility for class members:

- Subclasses in the same package
- Non-subclasses in the same package
- Subclasses in different packages
- Classes that are neither in the same package nor subclasses

The three access specifiers, private, public, and protected, provide a variety of ways to produce the many levels of access required by these categories.

CLASS MEMBER ACCESS

No.	PRIVATE	NO MODIFIER	PROTECTED	PUBLIC
SAME CLASS	YES	YES	YES	YES
SAME PACKAGE SUBCLASS	NO	YES	YES	YES
SAME PACKAGE NON-SUBCLASS	NO	YES	YES	YES
DIFFERENT PACKAGE SUBCLASS	NO	NO	YES	YES
DIFFERENT PACKAGE NON-SUBCLASS	NO	NO	NO	YES

2. PUBLIC

A class has only two possible access levels: default and public.
When a class is declared as public, it is accessible by any other code.
DEFAULT [friendly access]
If a class has default access, then it can only be accessed by other code within its same package.
The difference between public access and friendly access is that the public modifier makes fields visible in all classes regardless of their packages, while friendly access modifier makes field's visibility only in the same package but not in other packages.

3. PRIVATE

Anything declared private cannot be seen outside of its class.
It can't be inherited by the subclasses of other packages.

4. PROTECTED

Protected modifier makes the field visible not only to all classes and subclasses in the same package as well as in other packages.

5. PRIVATE PROTECTED

This modifier makes the fields visible in all sub classes regardless of what package they are in. These fields are not accessible by other classes in the same package.

EXAMPLE

The following example shows all combinations of the access control modifiers.
Following is the source code for the other package, p1.

//This is file **Protection.java:**

```
package p1;
public class Protection
{
     int n = 1;
     private int n_pri = 2;
    protected int n_pro = 3;
    public int n_pub = 4;
    public Protection()
    {
     System.out.println("base constructor");
     System.out.println("n = " + n);
     System.out.println("n_pri = " + n_pri);
     System.out.println("n_pro = " + n_pro);
     System.out.println("n_pub = " + n_pub);
     }
}
```

//This is file **Derived.java:**

```
package p1;
class Derived extends Protection
{
     Derived()
     {
               System.out.println("derived constructor");
               System.out.println("n = " + n);
               // class only
               // System.out.println("n_pri = " + n_pri);
               System.out.println("n_pro = " + n_pro);
               System.out.println("n_pub = " + n_pub);
     }
}
```

This is file **SamePackage.java:**

```
package p1;
class SamePackage
{
SamePackage()
 {
Protection p = new Protection();
System.out.println("same package constructor");
System.out.println("n = " + p.n);
// class only
// System.out.println("n_pri = " + p.n_pri);
System.out.println("n_pro = " + p.n_pro);
```

```
System.out.println("n_pub = " + p.n_pub);
}
}
```

Following is the source code for the other package, p2.

//This is file **Protection2.java:**

```
package p2;
class Protection2 extends p1.Protection
{
    Protection2()
    {
    System.out.println("derived other package constructor");
    // class or package only
    // System.out.println("n = " + n);
    // class only
    // System.out.println("n_pri = " + n_pri);
    System.out.println("n_pro = " + n_pro);
    System.out.println("n_pub = " + n_pub);
    }
}
```

//This is file **OtherPackage.java:**

```
package p2;
class OtherPackage
 {
    OtherPackage()
    {
    p1.Protection p = new p1.Protection();
    System.out.println("other package constructor");
    // class or package only
    // System.out.println("n = " + p.n);
    // class only
    // System.out.println("n_pri = " + p.n_pri);
    // class, subclass or package only
    // System.out.println("n_pro = " + p.n_pro);
    System.out.println("n_pub = " + p.n_pub);
    }
}
```

Here are two test files you can use. The one for package p1 is shown here:

```
// Demo package p1.
package p1;
// Instantiate the various classes in p1.
public class Demo
```

```
{
    public static void main(String args[])
    {
            Protection ob1 = new
            Protection(); Derived
            ob2 = new Derived();
            SamePackage ob3 =
            new SamePackage();
    }
}
```

The test file for p2 is shown next:

```
// Demo package p2.package p2;
// Instantiate the
various classes in p2.
public class Demo
 {
    public static void main(String args[])
    {
            Protection2 ob1 = new
            Protection2();
            OtherPackage ob2 =
            new OtherPackage();
    }
}
```