Unit 3
Package

Prog 1

```
// Name of the package must be same as the directory
// under which this file is saved
package myPackage;

public class MyClass
{
    public void getNames(String s)
    {
        System.out.println(s);
    }
}
```

```
/* import 'MyClass' class from 'names' myPackage */
import myPackage.MyClass;

public class PrintName
{
   public static void main(String args[])
   {
      // Initializing the String variable
      // with a value
      String name = "GeeksforGeeks";

      // Creating an instance of class MyClass in
      // the package.
      MyClass obj = new MyClass();

      obj.getNames(name);
   }
}
```

Note : MyClass.java must be saved inside the **myPackage** directory since it is a part of the package.

Prog 2

```java
package package_name;

public class ClassOne {
    public void methodClassOne() {
        System.out.println("Hello there its ClassOne");
    }
}
```

```java
package package_one;

public class ClassTwo {
    public void methodClassTwo(){
        System.out.println("Hello there i am ClassTwo");
    }
}
```

```java
import package_one.ClassTwo;
import package_name.ClassOne;

public class Testing {
    public static void main(String[] args){
        ClassTwo a = new ClassTwo();
        ClassOne b = new ClassOne();
        a.methodClassTwo();
        b.methodClassOne();
    }
}
```

Output:
Hello there i am ClassTwo Hello there its ClassOne

Access Protection

- Classes and packages are both means of encapsulating and containing the name space and scope of variables and methods. Packages act as containers for classes and other subordinate packages. Classes act as containers for data and code.
- The class is Java’s smallest unit of abstraction. Because of the interplay between classes and 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 modifiers, **private, public, and protected, provide a variety of ways to** produce the many levels of access required by these categories.

	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

Table 9-1 Class Member Access

Table 9-1 applies only to members of classes. A non-nested class has only two possible access levels: default and public. When a class is declared as **public, it is accessible by any** other code. If a class has default access, then it can only be accessed by other code within its same package. When a class is public, it must be the only public class declared in the file, and the file must have the same name as the class.

An Access Example

- This example has two packages and five classes. Remember that the classes for the two different packages need to be stored in directories named after their respective packages—in this case, p1 and p2.

  The source for the first package defines three classes:

- Protection,
- Derived,
- SamePackage.

  The first class defines four int variables in each of the legal protection modes. The variable

- n is declared with the default protection,
- n_pri is private,
- n_pro is protected, and
- n_pub is public.

- Each subsequent class in this example will try to access the variables in an instance of this class. The lines that will not compile due to access restrictions are commented out. Before each of these lines is a comment listing the places from which this level of protection would allow access.
- The second class, Derived, is a subclass of Protection in the same package, p1.
- This grants Derived access to every variable in Protection except for n_pri, the private one.
- The third class, SamePackage, is not a subclass of Protection, but is in the same package and also has access to all but n_pri.

P1(package)---> Protection —>Baseclass

subclass →Derived & SamePackage

P2(package) `(subclass) Protection2 extends p1.Protection`

`OtherPackage (class)`

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 = "4 + 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. The two classes defined in p2** cover the other two conditions that are affected by access control. The first class, **Protection2,** is a subclass of **p1.Protection. This grants access to all of p1.Protection's variables except** for **n_pri (because it is private) and n, the variable declared with the default protection.**
Remember, the default only allows access from within the class or the package, not extrapackage subclasses. Finally, the class **OtherPackage** has access to only one variable, **n_pub,** which was declared **public.**

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);
  }
}
```

If you want to try these two packages, 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();
  }
}
```

Importing Packages

- Java includes the **import** statement to bring certain classes, or entire packages, into visibility.
- Once imported, a class can be referred to directly, using only its name.
- The **import statement is a convenience to** the programmer and is not technically needed to write a complete Java program.
- the **import statement will** save a lot of typing.
- In a Java source file, **import statements occur immediately following the package** statement (if it exists) and before any class definitions.

This is the general form of the **import statement:**
import *pkg1 [.pkg2].(classname | *);*
Here, *pkg1 is the name of a top-level package, and pkg2 is the name of a subordinate*
package inside the outer package separated by a dot (**.**).

- This code fragment shows both forms in use:
- `import java.util.Date;`
- `import java.io.*;`

It must be emphasized that the **import** statement is optional. Any place you use a class name, you can use its *fully qualified name*, which includes its full package hierarchy. For example, this fragment uses an import statement:

```
import java.util.*;
class MyDate extends Date {
}
```

The same example without the **import** statement looks like this:

```
class MyDate extends java.util.Date {
}
```

In this version, **Date** is fully-qualified.

In this example, we created a class Demo stored into pack package and in another class Test, we are accessing Demo class by importing package name with class name.

```java
//save by Demo.java
package pack;
public class Demo {
    public void msg() {
        System.out.println("Hello");
    }
}

//save by Test.java
package mypack;
import pack.Demo;
class Test {
    public static void main(String args[]) {
        Demo obj = new Demo();
        obj.msg();
    }
}
```

In this example, we are creating a class A in package pack and in another class B, we are accessing it while creating object of class A.

```
//save by A.java
package pack;
public class A {
    public void msg() {
        System.out.println("Hello");
    }
}

//save by B.java
package mypack;
class B {
    public static void main(String args[]) {
        pack.A obj = new pack.A();  //using fully qualified name
        obj.msg();
    }
}
```