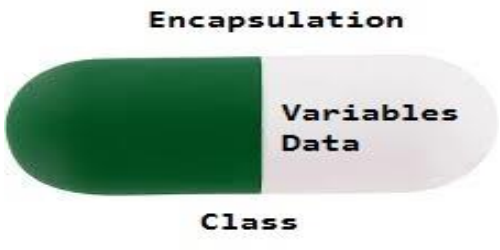
Week – 06

OOP concepts: Encapsulation

Encapsulation is one of the four fundamental OOP concepts. The other three are inheritance, polymorphism, and abstraction.

Encapsulation in Java is a mechanism of wrapping the data (variables) and code acting on the data (methods) together as a single unit. In encapsulation, the variables of a class will be hidden from other classes, and can be accessed only through the methods of their current class. Therefore, it is also known as **data hiding**.



Advantages of Encapsulation:

- **Data Hiding:** it is a way of restricting the access of our data members by hiding the implementation details. Encapsulation also provides a way for data hiding. The user will have no idea about the inner implementation of the class.
- **Increased Flexibility:** We can make the variables of the class read-only or write-only depending on our requirement. If we wish to make the variables read-only then we have to omit the setter methods like setName(), setAge(), or if we wish to make the variables write-only then we have to omit the get methods like getName(), getAge(), etc.
- **Reusability:** Encapsulation also improves the re-usability and is easy to change with new requirements.

To achieve encapsulation in Java −

- Declare the variables of a class as private.
- Provide public setter and getter methods to modify and view the variables values.

Example 1:

```
class Name {

    private int age; // Private is using to hide the data

    public int getAge() { return age; } // getter

    public void setAge(int age)
    {
        this.age = age;
    } // setter
}
```

```
class EncapDemo {
    public static void main(String[] args)
    {

        Name n1 = new Name();

        n1.setAge(19);

        System.out.println("The age of the person is: "+ n1.getAge());
    }
}
```

Example 2:

```
class Encapsulate {
    // private variables declared these can only be accessed by public methods of class
    private String name;
    private int roll;
    private int age;

    public int getAge() { return age; }

    public String getName() { return name; }

    public int getRoll() { return roll; }

    public void setAge(int newAge) { age = newAge; }

    public void setName(String newName)  { name = newName;   }

    public void setRoll(int newRoll) { roll = newRoll; }
}

public class EncapDemo2 {
    public static void main(String[] args)
    {
        Encapsulate obj = new Encapsulate();

        // setting values of the variables
        obj.setName("Harsh");
        obj.setAge(19);
        obj.setRoll(51);

        // Displaying values of the variables
        System.out.println("name: " + obj.getName());
        System.out.println("age: " + obj.getAge());
        System.out.println("roll: " + obj.getRoll());

        // Direct access of name is not possible due to encapsulation
        // System.out.println("Geek's roll: " + obj.name);
    }
}
```

In the above program, the class Encapsulate is encapsulated as the variables are declared as private. The get methods like getAge() , getName() , getRoll() are set as public, these methods are used to access these variables. The setter methods like setName(), setAge(), setRoll() are also declared as public and are used to set the values of the variables.

PACKAGES

java package is a group of similar types of classes, interfaces and sub-packages.

Package in java can be categorized in two form, built-in package and user-defined package.

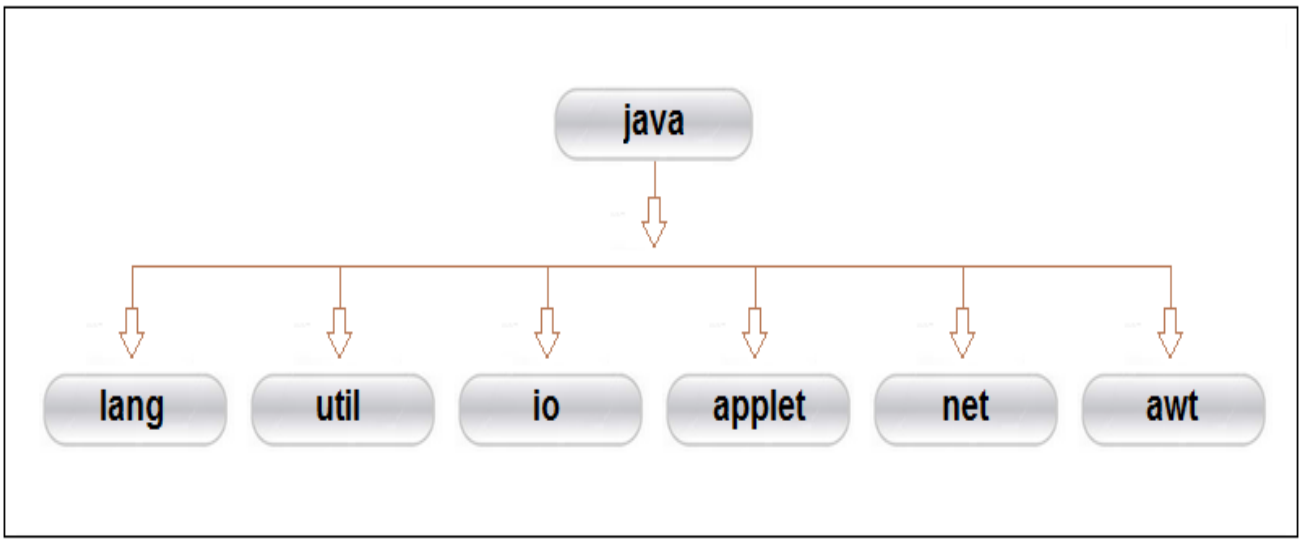
There are many built-in packages such as java, lang, awt, javax, swing, net, io, util, sql etc.

Advantage or Benefits of Java Package

1. The classes contained in the packages of other programs an be easily reused.
2. Packages provides a way to "hide" classes thus preventing other programs or packages from accessing classes that are meant for internal use only.
3. Java package is used to categorize the classes and interfaces so that they can be easily maintained.
4. Java package provides access protection.
5. Java package removes naming collision

JAVA API PACKAGES

Java **APl(Application Program Interface)** provides a large number of classes grouped into different packages according to functionality. Most of the time we use the packages available with the Java API. Following figure shows the system packages that are frequently used in the programs.



Java System Packages and Their Classes

java.lang	Language support classes. They include classes for primitive types, string, math functions, thread and exceptions.
java.util	Language utility classes such as vectors, hash tables, random numbers, data, etc.
java.io	Input/output support classes. They provide facilities for the input and output of data.
java.applet	Classes for creating and implementing applets.
java.net	Classes for networking. They include classes for communicating with local computers as well as with internet servers.
java.awt	Set of classes for implementing graphical user interface. They include classes for windows, buttons, lists, menus and so on.

USING SYSTEM PACKAGES

The packages are organized in a hierarchical structure as illustrated in fig below. This shows the package named java contains the package awt, which in turn contains various classes required for implementing GUI.

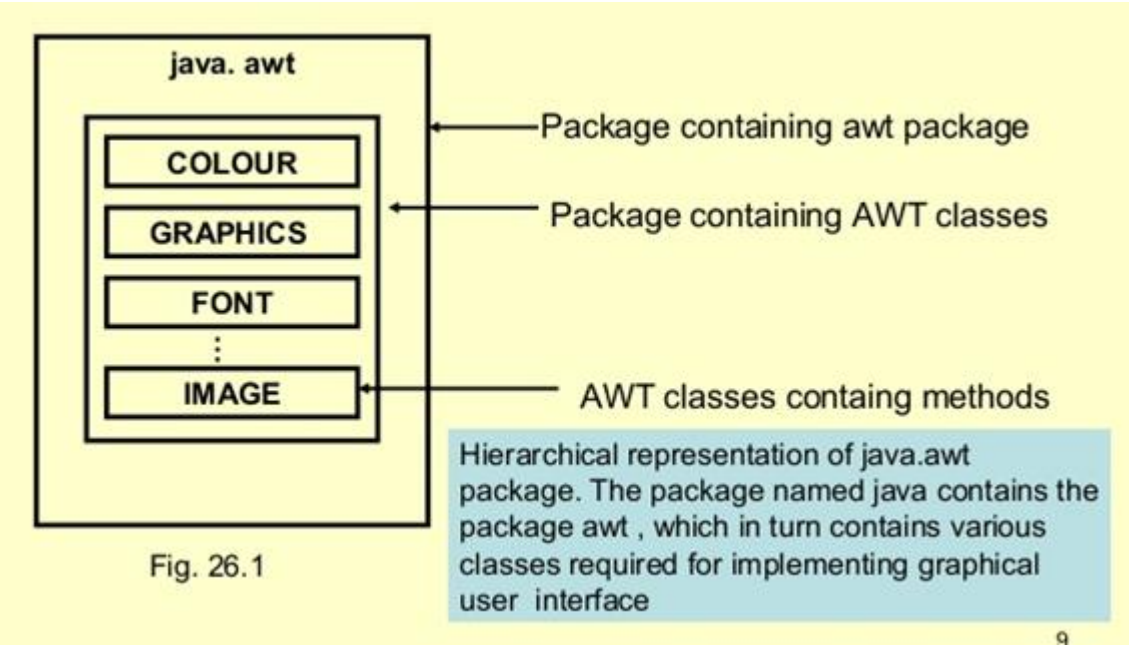


Fig. 26.1

In java, the **import** keyword used to import built-in and user-defined packages. When a package has imported, we can refer to all the classes of that package using their name directly.

The import statement must be after the package statement, and before any other statement.

Using an import statement, we may import a specific class or all the classes from a package.

Syntax:

import packagename.classname;

Or

import packagename.*;

example:

import java.awt.Color; // imports only class Color from awt package.

or

import java.awt.*;

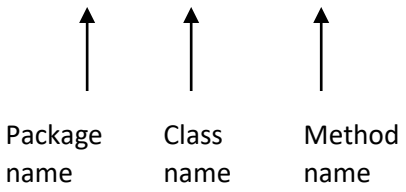
will bring all classes of java.awt package.

NAMING CONVENTIONS

Packages can be named using the standard Java naming rules.

By convention, packages begin with lowercase letters.

Double y = java.lang.Math.sqrt(x);



This statement uses a fully qualified class name Math to invoke the method sqrt(). Note that method begin with lowercase letters.

CREATING PACKAGES

We must first declare the name of the package using the package keyword followed by package name. This must be first statement in the java source file.

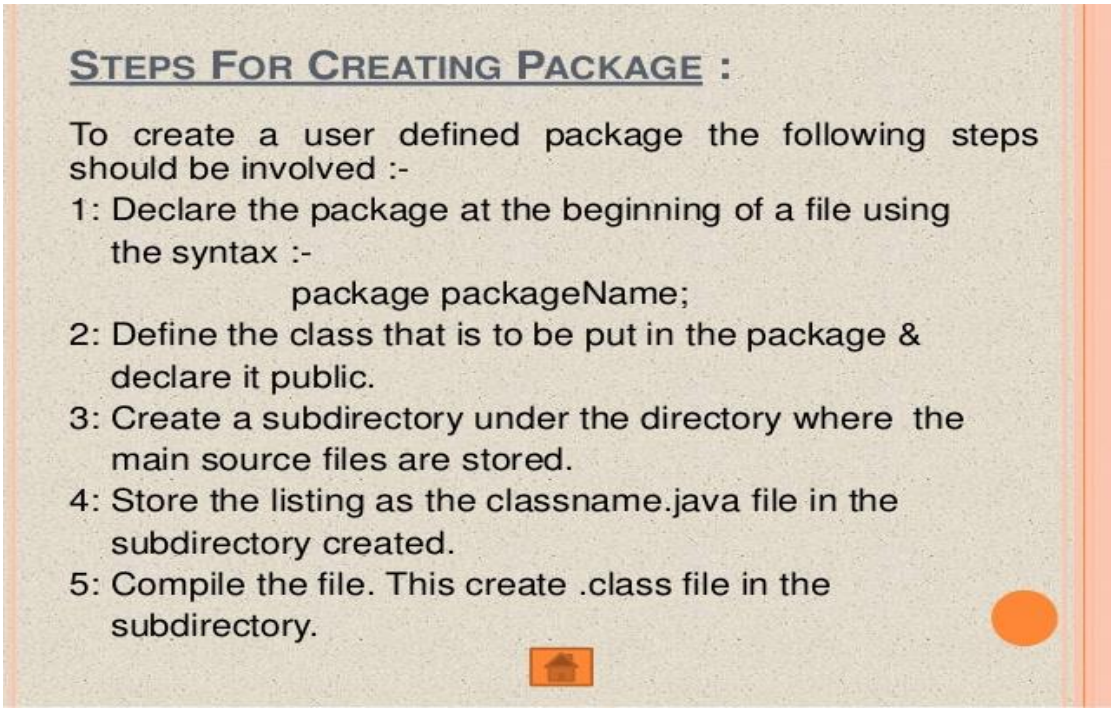
Package firstPackage;

Public class FirstClass

{

Body of the class

}



ACCESSING THE PACKAGES

We use import statement to access the packages.

Syntax:

import package1 [.package2] [.package2] [.package3] .classname;

example

import firstPackage.secondPackage.MyClass;

- Using one import statement, we may import only one package or a class.
- Using an import statement, we cannot import a class directly, but it must be a part of a package.
- A program may contain any number of import statements.

WAJP to implement the concept of importing classes from user defined package and creating packages

```
/* CREATE A NEW FOLDER WITH NAME pkg1 INSIDE THE pkg1 FOLDER SAVE A.java*/
package pkg1;
public class A
{
   public void displayA()
   {
    System.out.println("Class A");
   }
}
```

```
/* CREATE A NEW FOLDER WITH NAME pkg2 INSIDE THE pkg2 FOLDER SAVE B.java*/
package pkg2;
public class B
{
  protected int m=10;
  public void displayB()
  {
   System.out.println("class B");
   System.out.println("m="+m);
  }
}
```

```
import pkg1.A;
import pkg2.*;
class prog14demo
{
   public static void main(String args[])
   {
     A a= new A();
     B b= new B();
     a.displayA();
     b.displayB();
    }
}
```

```
C:\javaprg>javac prog14demo.java

C:\javaprg>java prog14demo
Class A
child class
y=20
```

During the compliation of prog14demo.java the complier checks for the A.class and B.class in the pkg1 and pkg2 packages or subdirectories. When prog14demo program is run, Java looks for the file prog14demo.class and loads it using something called class loader. Now the interpreter knows that it also needs the code in the file A.class and B.class and loads it as well.

HIDING CLASSES

When we import a package using asterisk (*), all public classes are imported. However, we may prefer to “not import”cerain classes. i.e., we may like to hide these classes from accessing from outside of the packages. Such classes should not be declared “not public”

Example

```
package p1;
public class X
{
// body of X
}
Class Y
{
//body of Y
}
```

Here class Y which is not declared public is hidden from outside of the package p1. This class can be seen and used only by other classes in the same package.

Note java source file should contain only one public class and may include any number of non-public classes.

```
Import p1.*;
X objectX;    // OK; class X is available here
Y object;      // NOT OK; Y is not available
```

Java complier would generate an error message for this code because the class Y, which has not been declared public.

Single Responsibility Principle (SRP)

Single Responsibility Principle. As the name indicates, it states that all classes and modules should have only one well-defined responsibility.

A class should have one, and only one reason to change.

This means when we design our classes, we need to ensure that our class is responsible only for 1 task or functionality and when there is a change in that task/functionality, only then, that class should change.

In the world of software, change is the only constant factor. When requirements change and when our classes do not adhere to this principle, we would be making too many changes to our classes to make our classes adaptable to the new business requirements. This could involve lots of side effects, retesting, and introducing new bugs. Also, our dependent classes need to change, thereby recompiling the classes and changing test cases. Thus, the whole application will need to be retested to ensure that new functionality did not break the existing working code.

Benefits of Single Responsibility Principle

- When an application has multiple classes, each of them following this principle, then the applicable becomes more maintainable, easier to understand.
- The code quality of the application is better, thereby having fewer defects.
- Onboarding new members are easy, and they can start contributing much faster.
- Testing and writing test cases is much simpler

Example: CALCULATOR PROGRAM

```
class Calculator {
        // this class is responsible for Arithmetic operations &  printing Values
  public static int add(int x, int y) { return x + y; }
  public static int sub(int x, int y) { return x - y; }
  public static int mul(int x, int y) { return x * y; }
  public static int div(int x, int y) { return x / y; }
  public static void display(int value)
  {
          System.out.println("The value is="+value);
  }
}

class CalcDemo1
{
   public static void main(String args[])
    {
            int a = Calculator.add(20, 30);
            Calculator.display(a);
            int b = Calculator.sub(20, 30);
            Calculator.display(b);
            int c = Calculator.mul(20, 30);
            Calculator.display(c);
            int d = Calculator.div(20, 30);
            Calculator.display(d);

          }
}
```

The above program violates SRP, since the class Calculator has 2 responsibilities Arithmetic operations & printing Values

To maintain SRP we have to write the program so that the class should have only one responsibility.

```
class Calculator1 { // this class is only responsible for Arithmetic operations
  public static int add(int x, int y) { return x + y; }
  public static int sub(int x, int y) { return x - y; }
  public static int mul(int x, int y) { return x * y; }
  public static int div(int x, int y) { return x / y; }
}
class ResultPrinter { // this class is only responsible for printing int values
  public static void printResult(int value)
  {
   System.out.println("The value is="+value);
  }
}
```

```
public class CalcDemo
{
   public static void main(String args[])
     {
            int a = Calculator.add(20, 30);
            ResultPrinter.printResult(a);
            int b = Calculator.sub(20, 30);
            ResultPrinter.printResult(b);
            int c = Calculator.mul(20, 30);
            ResultPrinter.printResult(c);
            int d = Calculator.div(20, 30);
            ResultPrinter.printResult(d);

          }
}
```

The above Calculator program satisfies SRP.