Module 2: Inheritance, Interface and Packages

Inheritance basics, Types of inheritance, Member access rules, Usage of super key word, Method overriding, Usage of final, Abstract classes, Interfaces - differences between abstract classes and interfaces, defining an interface, implementing interface, applying interfaces, variables in interface and extending interfaces; Packages - defining, creating and accessing a package, importing packages, access control in packages.

Inheritance

2.1 Inheritance Basics

Inheritance is the concept of a child class (sub class) automatically inheriting the variables and methods defined in its parent class (super class). A primary feature of object-oriented programming along with encapsulation and polymorphism.
In java inheritance achieved by using the keyword EXTENDS.
Syntax:

```
class A
{
}
class B extends A                    // B inherits the features of class A
{
}
```

A---superclass
B---subclass

Advantages of Inheritance

- *Reusability of code*

Reusability--building new components by utilising existing components- is yet another important aspect of OO paradigm. It is always good/ "productive" if we are able to reuse something that is already exists rather than creating the same all over again. This is achieved by creating new classes, reusing the properties of existing classes.

- *Effort and time saving*

The above concept of reusability achieved by inheritance saves the programmer time and effort. Since the main code written can be reused in various situation as needed.

- *Increased reliability*

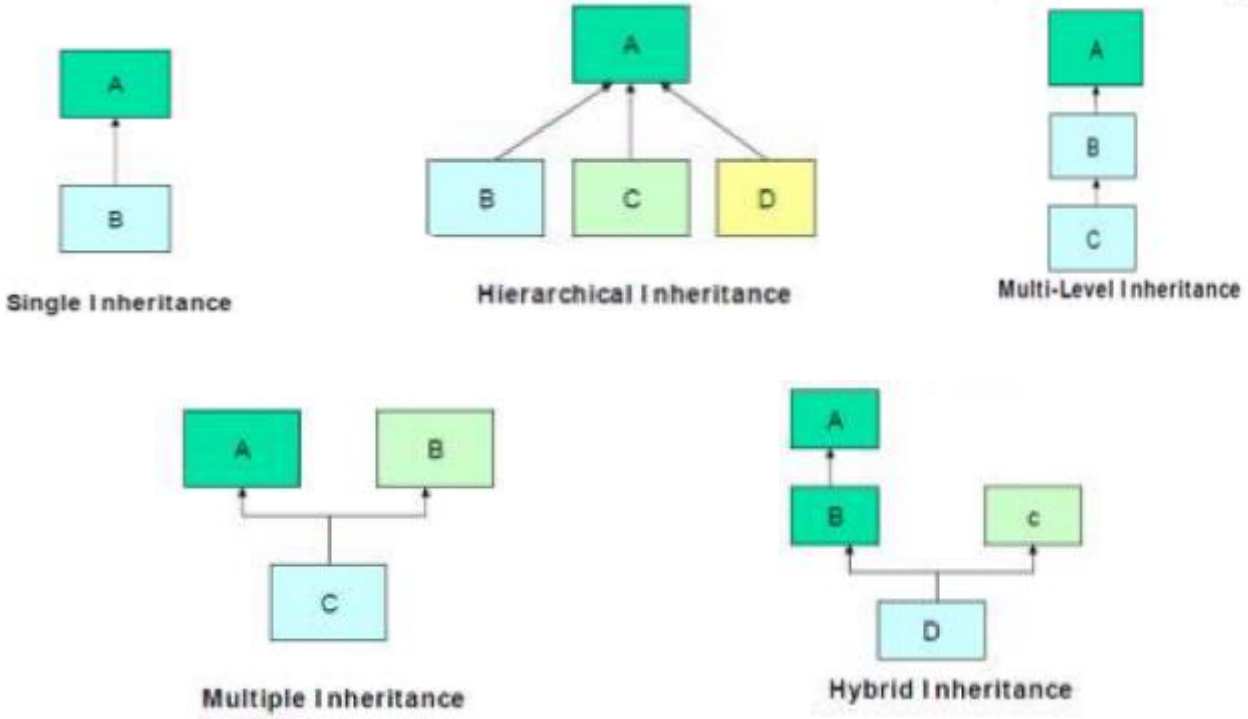
The program with inheritance become more understandable and easily maintainable as the sub classes are created from existing reliable working classes.

2.2 Types of Inheritance

There are five types of inheritance. They are

1. Single Inheritance
2. Hierarchical Inheritance
3. Multi-Level Inheritance

4. Multiple Inheritance
5. Hybrid Inheritance



1. Single Inheritance

The process of creating only one sub class from only one super class knows as single inheritance.

Example:

```
class SuperClass
{
int a;
}
class SubClass extends SuperClass
{
int b;
SubClass()
{
a=10;
b=20;
}
void sum()
{
System.out.println("sum of a and b is:"+(a+b));
}
}
class Test
{
public static void main(String args[])
{
SubClass obj=new SubClass();
Obj.sum();
}
}
```

Output:

Sum of a and b is 30

2. Hierarchical Inheritance

The process of creating several sub classes from only one super class knows as Hierarchical inheritance.

Example:

```
class SuperClass
{
int a;
}
class SubClass1 extends SuperClass
{
int b;
SubClass1()
{
a=10;
b=20;
}
void sum()
{
System.out.println(“sum of a and b is:”+(a+b));
}
}
class SubClass2 extends SuperClass
{
int c;
SubClass2()
{
a=10;
c=20;
}
void mul()
{
System.out.println(“multiplication of a and c is:”+(a*c));
}
}
class Test
{
public static void main(String args[])
{
SubClass1 obj1=new SubClass1();
Obj1.sum();
SubClass2 obj2=new SubClass2();
Obj2.mul();
}
}
```

Output:

Sum of a and b is 30

Multiplication of a and c is 200

3. Multi-Level Inheritance

The process of creating new subclass from an already inherited sub class is known as Multilevel inheritance.

Example:

```
class SuperClass
{
int a;
}
class SubClass1 extends SuperClass
{
int b;
SubClass1()
{
a=10;
b=20;
}
void sum()
{
System.out.println("sum of a and b is:"+(a+b));
}
}
class SubClass2 extends SubClass1
{
int c;
SubClass2()
{
a=10;
b=10;
c=20;
}
void mul()
{
System.out.println("multiplication of a, b and c is:"+(a*b*c));
}
}
class Test
{
public static void main(String args[])
{
SubClass1 obj1=new SubClass1();
Obj1.sum();
SubClass2 obj2=new SubClass2();
Obj2.mul();
}
}
```

Output:

Sum of a and b is 30

Multiplication of a, b and c is 2000

4. Multiple Inheritance

Multiple inheritance in java is the capability of creating a single class with multiple super classes. Unlike some other popular object-oriented programming languages like C++, java doesn't provide support for multiple inheritance in classes. Java doesn’t support multiple inheritances in classes because it can lead to diamond problem and rather than providing some complex way to solve it, there are better ways through which we can achieve the same result as multiple inheritances.

5. Hybrid Inheritance

The process of creating more than one type of inheritance is known as hybrid inheritance.

Example:

```
class SuperClass
{
int a;
}
class SubClass1 extends SuperClass
{
int b;
SubClass1()
{
a=10;
b=20;
}
void sum()
{
System.out.println(“sum of a and b is:”+(a+b));
}
}
class SubClass2 extends SuperClass
{
int c;
SubClass2()
{
a=10;
c=20;
}
void mul()
{
System.out.println(“multiplication of a and c is:”+(a*c));
}
}
class SubClass3 extends SubClass1
{
int d;
SubClass3()
{
a=40;
b=30;
d=20;
}
```

```
void sub()
{
System.out.println("substraction of a, b and d is:"+(a-b-d));
}
}
class Test
{
public static void main(String args[])
{
SubClass1 obj1=new SubClass1();
Obj1.sum();
SubClass2 obj2=new SubClass2();
Obj2.mul();
SubClass3 obj3=new SubClass3();
Obj3.sub();
}
}
```

Output:

Sum of a and b is 30

Multiplication of a and c is 200

Substraction of a, b and d is 10

2.3 Member Access Rules

Member Access Rules in inheritance (or) Private Members in Inheritance (or) To show How to Access Members with Different Access Controls.

The access specifiers that are used to control member access are: public, private & protected. In addition to these three if no specifier is used, it results in default.

i. A private data member cannot be inherited in the subclass.
ii. A public and protected data members are inherited in the subclass.
iii. When no modifier is used, it results in default. Default members are also inherited in the subclass.

Example:

```
class SuperClass
{
private int a;
public int b;
protected int c;
int d;
void set_a(int x)
{
a=x;
}
int get_a()
{
return a;
```

```
}
}
class SubClass extends SuperClass
{
void set(int x1,int x2,int x3,int x4)
{
set_a(x1);
b=x2;
c=x3;
d=x4;
}
void display()
{
System.out.println("a="+get_a()+"b="+b+"c="+c+"d="+d);
}
}
class Test
{
public static void main(String args[])
{
SubClass obj=new SubClass();
obj.set(10,20,30,40);
obj.display();
}
}
```

Output:

```
a=10
b=20
c=30
d=40
```

2.4 Usage of Super Keyword

The super keyword in Java is a reference variable which is used to refer immediate parent class object. Whenever you create the instance of subclass, an instance of parent class is created implicitly which is referred by super reference variable.

Usage of Java super Keyword

1. super can be used to refer immediate parent class instance variable.
2. super can be used to invoke immediate parent class method.
3. super() can be used to invoke immediate parent class constructor.

1. super can be used to refer immediate parent class instance variable

We can use super keyword to access the data member or field of parent class. It is used if parent class and child class have same fields.

Example:

```
class Parent
{
int a=10;
}
class Child extends Parent
{
int a=20;
void display()
{
System.out.println(“a is:”+a);
System.out.println(“super class a is:”+super.a);
}
}
class Test
{
public static void main(String args[])
{
Child obj=new Child();
obj.display();
}
}
```

Output:

```
a is 20
super class a is 10
```

In the above example, Parent and Child both classes have a common property ‘a’. If we print ‘a’ property, it will print the value of ‘a’ class by default. To access the parent property, we need to use super keyword.

2. super can be used to invoke immediate parent class method

The super keyword can also be used to invoke parent class method. It should be used if child class contains the same method as parent class. In other words, it is used if method is overridden.

Example:

```
class Parent
{
void show()
{
System.out.println(“Parent Class”);
}
}
class child extends Parent
{
void show()
{
```

```
System.out.println("Child Class");
}
void display()
{
super.show();
show();
}
}
class Test
{
public static void main(String args[])
{
Child obj=new Child();
obj.display();
}
}
```

Output:

```
Parent Class
Child Class
```

In the above example Parent and Child both classes have show() method if we call show() method from Child class, it will call the show() method of Child class by default because priority is given to local. To call the Parent class method, we need to use super keyword.

3. super() can be used to invoke immediate parent class constructor

The super keyword can also be used to invoke the parent class constructor.

Example:

```
class Parent
{
Parent(int i)
{
System.out.println("Parent Class Constructor Value is:"+i);
}
}
class Child extends Parent
{
Child()
{
super(100);
System.out.println("Child Class Constructor");
}
}
class Test
{
public static void main(String args[])
```

```
{
Child obj=new Child();
}
}
```

Output:

Parent Class Constructor Value is: 100
Child Class Constructor

super() is added in each class constructor automatically by compiler if there is no super() or this().

As we know well that default constructor is provided by compiler automatically if there is no constructor. But, it also adds super() as the first statement.

2.5 Method Overriding

When a method in a subclass has the same name and type as a method in its superclass then the method in the subclass is said to override the method in the superclass.

When an overridden method is called from within a subclass, it always refers to the version of that method defined by the subclass.

Example:

```
class SuperClass
{
int a=10;
int b=20;
void display()
{
System.out.println(“Parent Class”);
}
}
class SubClass extends SuperClass
{
void display()
{
System.out.println(“Child Class”);
}
}
class Test
{
public static void main(String args[])
{
SubClass obj=new SubClass();
obj.display();
System.out.println(“ a and b values are:”+obj.a+” “+obj.b);
}
}
```

Output:

Child Class
a and b values are: 10 20

Differences between Method Overloading and Method Overriding

Method Overloading	Method Overriding
It is performed within a class.	It is performed between two classes that have inheritance relationship.
Signature must be different.	Signature must be the same.
It is example of compile time polymorphism.	It is example of run time polymorphism.
It is used to increase readability of program.	It is used to provide different implementation of super class method.
Static methods can be overloaded.	Static methods can't be override.
Private methods can be overloaded.	Private methods can't be override.
Final methods can be overloaded.	Final methods can't be override.
Return type of method may be same or different.	Return type must be the same.

2.6 Usage of Final Keyword

The final keyword in java is used to restrict the user. The java final keyword can be used in many contexts. Final can be:

1. variable
2. method
3. class

The final keyword can be applied with the variables, a final variable that have no value it is called blank final variable or uninitialized final variable. It can be initialized in the constructor only. The blank final variable can be static also which will be initialized in the static block only. We will have detailed learning of these. Let's first learn the basics of final keyword.

1. Java Final Variable

If you make any variable as final, you cannot change the value of final variable (It will be constant).

Example:

There is a final variable 'a', we are going to change the value of this variable, but it can't be changed because final variable once assigned a value can never be changed.

```
class Sample
{
final int a=10;
Sample()
{
a=20;
}
}
class Test
{
```

```
public static void main(String args[])
{
Sample obj=new Sample();
}
}
```

Output:

Compilation Time Error

2. Java Final Method

If you make any method as final, you cannot override it.

Example:

```
class Base
{
final void show()
{
System.out.println(“show() in Base Class”);
}
}
class Derived extends Base
{
void show()
{
System.out.println(“show() in Derived Class”);
}
}
class Test
{
public static void main(String args[])
{
Derived obj=new Derived();
}
}
```

Output:

Compilation Time Error

3. Java Final Class

If you make any class as final, you cannot extend it.

Example:

```
final class Base
{
void show()
{
```

```
System.out.println(“show() in Base”);
}
}
class Derived extends Base
{
void show()
{
System.out.println(“Show() in Derived”):
}
}
class Test
{
public static void main(String args[])
{
Derived obj=new Derived();
}
}
```

Output:

Compilation Time Error

2.7 Abstract Classes

Abstract Class:

An abstract class is declared using abstract keyword. An abstract class is a class definition that is incomplete in the sense that it has some abstract methods. It is not compulsory that an abstract class has an abstract method but if a class has an abstract method, the class has to be declared as abstract. Abstract classes can never be initiated but they can be subclassed.

Abstract Method:

Abstract method is declared using abstract keyword. Abstract methods are defined as part of the class but the inner workings (body of methods) are not provided. Abstract methods end with semicolon instead of curly brackets.

Syntax:

```
public abstract class Bike
{
abstract void gear();
abstract void topSpeed();
}
```

Why we need an Abstract Class:

The idea behind an abstract class is putting common functionality to an abstract class that other classes will extend. For example, we have an abstract class named fan with a method for turning the fan on and off but not defining the procedure to do that i.e., with no body declaration. Now we can extend the fan class into table fan and define on and off procedure accordingly or we can extend the fan class to ceiling fan and define on and off procedure according to ceiling fan.

Example:

```
abstract class Shape
{
abstract void draw();
}
class Rectangle extends Shape
{
void draw()
{
System.out.println("Drawing Rectangle");
}
}
class Circle extends Shape
{
void draw()
{
System.out.println("Drawing Circle");
}
}
class Test
{
public static void main(String args[])
{
Shape s;
Rectangle obj1=new Rectangle();
s=obj1;
s.draw();
Circle obj2=new Circle();
obj2.draw();
}
}
```

Output:

```
Drawing Rectangle
Drawing Circle
```

Interfaces

2.8 Differences between Abstract Classes and Interfaces:

Abstract Classes	Interfaces
Abstract class does not provide full abstraction i.e., it may contain concrete methods also.	Interface provides full abstraction i.e., it contains abstract methods only.
We can't achieve multiple inheritance using abstract class.	We can implement multiple inheritance using interface.
Every method present in abstract class need not be public and abstract.	Every method present in interface is by default public and abstract.

No restrictions on abstract class method modifiers.	We can't declare interface methods with modifiers private, protected, final and static.
Every variable present in abstract class need not be public, static and final.	Every Variable present in interface is always public, static and final.
Not necessary to initialize variables during declaration in abstract class.	We should perform variable initialization during declaration other wise compilation error will gets displayed.
An abstract class can be extended using "extends" keyword.	An interface can be extended using "implements" keyword.
Abstract class can provide the implementation of interface.	Interface can't provide the implementation of abstract class.

Example:
public abstract class Shape
{
public abstract void draw();
}
	Example:
	public interface Drawable
	{
	Void draw();
	}

2.9 Defining Interface

An interface in Java is a blueprint of a class. It has static constants and abstract methods. The interface in Java is a mechanism to achieve abstraction. There can be only abstract methods in the Java interface, not method body.

It is used to achieve abstraction and multiple inheritance in Java. In other words, you can say that interfaces can have abstract methods and variables. It cannot have a method body.

Why use Java Interface?

There are mainly three reasons to use interface. They are given below.

- It is used to achieve abstraction.
- By interface, we can support the functionality of multiple inheritance.
- It can be used to achieve loose coupling.

How to declare an interface?

An interface is declared by using the interface keyword. It provides total abstraction; means all the methods in an interface are declared with the empty body, and all the fields are public, static and final by default. A class that implements an interface must implement all the methods declared in the interface.

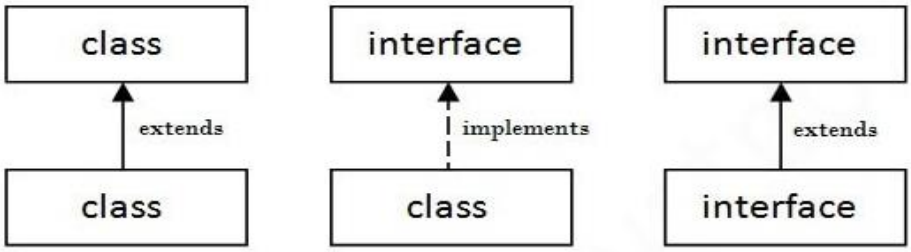
Syntax:

interface <interface_name>

{
// declare constant fields
// declare methods that abstract
// by default.
}

The relationship between Classes and Interfaces

As shown in the figure given below, a class extends another class, an interface extends another interface, but a class implements an interface**.**



2.10 Implementing Interface

When a class implements an interface, you can think of the class as signing a contract, agreeing to perform the specific behaviors of the interface. If a class does not perform all the behaviors of the interface, the class must declare itself as abstract.

A class uses the implements keyword to implement an interface. The implements keyword appears in the class declaration following the extends portion of the declaration.

Example:

```
interface Animal
{
void eat();
void travel();
}
class MammalInt implements Animal
{
public void eat()
{
System.out.println(“Mammal Eats”);
}
public void travel()
{
Sysytem.out.println(“Mammal Travel”);
}
public static void main(String args[])
{
MammalInt obj=new MammalInt();
obj.eat();
obj.travel();
}
}
```

Output:

```
Mammal Eats
Mammal Travels
```

When overriding methods defined in interfaces, there are several rules to be followed:

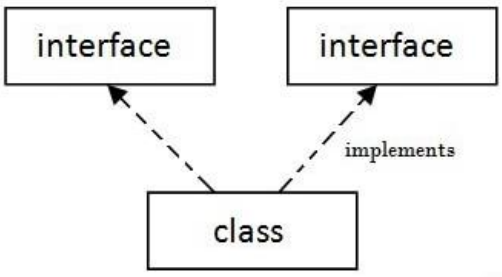
- Checked exceptions should not be declared on implementation methods other than the ones declared by the interface method or subclasses of those declared by the interface method.
- The signature of the interface method and the same return type or subtype should be maintained when overriding the methods.
- An implementation class itself can be abstract and if so, interface methods need not be implemented.

When implementation interfaces, there are several rules:

- A class can implement more than one interface at a time.
- A class can extend only one class, but implement many interfaces.
- An interface can extend another interface, in a similar way as a class can extend another class.

Implementing Multiple Interfaces ***(Multiple Inheritance by implementing Multiple Interfaces)***

If a class implements multiple interfaces, then it is known as multiple inheritance. When a class wants to implement more than one interface, we use the implements keyword is followed by a comma-separated list of the interfaces implemented by the class.



Example:

```
interface A
{
int a=10;
void show();
}
interface B
{
int a=20;
void show();
}
class C implements A,B
{
int c;
public void show()
{
c=A.a+B.a;
System.out.println("A.a="+A.a);
System.out.println("B.a="+B.a);
```

```
System.out.println("c="+c);
}
}
class Test
{
public static void main(String args[])
{
C.obj=new C();
obj.show();
}
}
```

Output:

```
A.a=10
B.a=20
c=30
```

2.11 Applying Interface

To understand the power of interfaces, let's look at a more practical example.

Example:

```
interface shape
{
void area();
}
class Rectangle implements Shape
{
public void area()
{
System.out.println("Drawing Rectangle");
}
}
class Circle implements Shape
{
public void area()
{
System.out.println("Drawing Circle");
}
}
class Square implements Shape
{
public void area()
{
System.out.println("Drawing Square");
}
}
```

```
class Test
{
public static void main(String args[])
{
Rectangle obj1=new Rectangle();
Shape S=new Shape();
S=obj1;          //S now refers to Rectangle object
S.area();
Circle obj2=new Circle();
obj2.area();
Square obj3=new Square();
obj3=area();
}
}
```

Output:

```
Drawing Rectangle
Drawing Circle
Drawing Square
```

2.12 Variables in Interface

In java, an interface is a completely abstract class. An interface is a container of abstract methods and static final variables. The interface contains the static final variables. The variables defined in an interface cannot be modified by the class that implements the interface, but it may use as it defined in the interface.

- The variable in an interface is public, static, and final by default.
- If any variable in an interface is defined without public, static, and final keywords then, the compiler automatically adds the same.
- No access modifier is allowed except the public for interface variables.
- Every variable of an interface must be initialized in the interface itself.
- The class that implements an interface cannot modify the interface variable, but it may use as it defined in the interface.

Example:

```
interface Sample
{
int speed=100;
}
class InterfaceVariable implements Sample
{
public static void main(String args[])
{
//speed=150; // cannot be modified because interface variable is by default final
System.out.println("The value of speed is:"+speed);
}
}
```

Output:

The value of speed is 100

2.13 Extending Interface

In java, an interface can extend another interface. When an interface wants to extend another interface, it uses the keyword extends. The interface that extends another interface has its own members and all the members defined in its parent interface too. The class which implements a child interface needs to provide code for the methods defined in both child and parent interfaces, otherwise, it needs to be defined as abstract class.

Example:

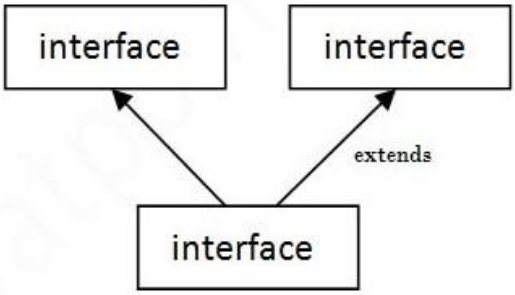
```
interface A
{
void m1();
void m2();
}
interface B extends A
{
void m3();
}
Class C implements B
{
public void m1()
{
System.out.println(“Interface A m1() definition”);
}
public void m2()
{
System.out.println(“Interface A m2() definition”);
}
public void m3()
{
System.out.println(“Interface B m3() definition”);
}
}
class Test
{
public static void main(String args[])
{
C obj=new C();
obj.m1();
obj.m2();
obj.m3();
}
}
```

Output:

Interface A m1() definition
Interface A m2() definition
Interface B m3() definition

Extending Multiple Interfaces ***(Multiple Inheritance by extending Multiple Interfaces)***

A Java class can only extend one parent class. Multiple inheritance is not allowed. Interfaces are not classes, however, and an interface can extend more than one parent interface. Multiple inheritance can be achieved by extending multiple interfaces (interface extends multiple interfaces). The extends keyword is used once, and the parent interfaces are declared in a comma-separated list.



Example:

```
interface A
{
int a=10;
void show();
}
interface B
{
int a=20;
void show();
}
interface C extends A,B
{
int a=30;
void show();
}
class D implements C
{
int d;
public void show()
{
d=A.a+B.a+C.a;
System.out.println("A.a="+A.a);
System.out.println("B.a="+B.a);
System.out.println("C.a="+C.a);
System.out.println("d="+d);
```

```
}
}
class Test
{
public static void main(String args[])
{
D.obj=new D();
obj.show();
}
}
```

Output:

```
A.a=10
B.a=20
C.a=30
d=60
```

Packages

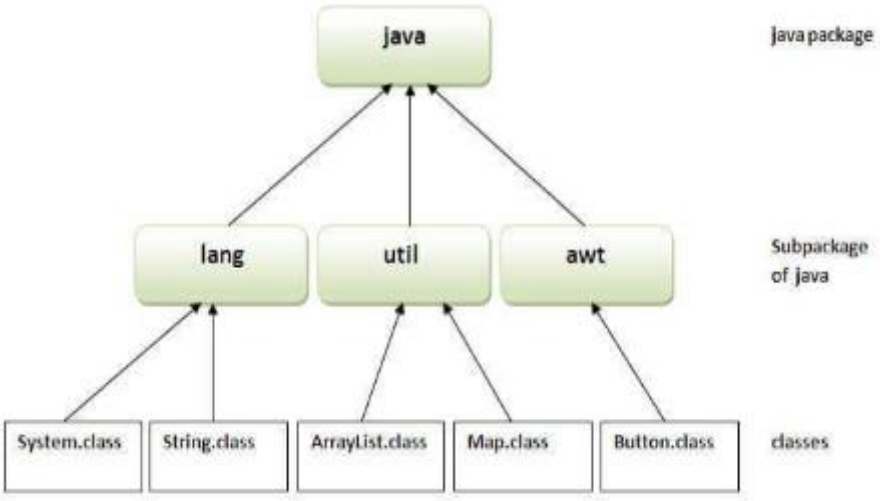
2.14 Defining Packages

A 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.

- Built-in Package: A package which is already declared is called built-in packages. There are many built-in packages such as java, lang, awt, javax, swing, net, io, util, sql etc.
- User-defined Packages: A package which is declared by user is called user-defined packages. Here, we will have the detailed learning of creating and using user-defined packages.

Advantages of Java Package

1) Java package is used to categorize the classes and interfaces so that they can be easily maintained.
2) Java package provides access protection.
3) Java package removes naming collision.



2.15 Creating Packages

The package keyword is used to create a package in java.

Syntax: package packagename;

Example:

```
//save as Simple.java
package mypack;
public class Simple
{
public static void main(String args[])
{
System.out.println("Welcome to package");
}
}
```

How to compile java package

If you are not using any IDE, you need to follow the syntax given below:

javac -d directory javafilename

Example: java -d . Simple.java

The -d switch specifies the destination where to put the generated class file. You can use any directory name like /home (in case of Linux), d:/abc (in case of windows) etc. If you want to keep the package within the same directory, you can use . (dot).

How to run java package program

You need to use fully qualified name e.g. mypack.Simple etc to run the class.

To Compile: javac -d . Simple.java

To Run: java mypack.Simple

Output: Welcome to Package

The -d is a switch that tells the compiler where to put the class files i.e., it represents the destination. The . represents the current folder.

2.16 Accessing and Importing Packages

There are three ways to access and import the package from outside the package.

1. import package.*;
2. import package.classname;
3. fully qualified name.

1. Using packagename.*

- If you use package.* then all the classes and interfaces of this package will be accessible but not subpackages.
- The import keyword is used to make the classes and interface of another package accessible to the current package.

Example:

```
//save by A.java
package pack;
public class A
{
public void msg(){System.out.println("Hello");}
}
//save by B.java
package mypack;
import pack.*;
class B{
public static void main(String args[]){
A obj = new A();
obj.msg();
}
}
```

Output:

Hello

2. **Using packagename.classname**

If you import package.classname then only declared class of this package will be accessible.

Example:

```
//save by A.java
package pack;
public class A
{
public void msg(){System.out.println("Hello");}
}
//save by B.java
package mypack;
import pack.A;
class B{
public static void main(String args[]){
A obj = new A();
obj.msg();
}
}
```

Output:

Hello

3. Using Fully Qualified Name

If you use fully qualified name then only declared class of this package will be accessible. Now there is no need to import. But you need to use fully qualified name every time when you are accessing the class or interface.

It is generally used when two packages have same class name e.g. java.util and java.sql packages contain Date class.

Example:

```
//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();
}
}
```

Output:

Hello

2.17 Access Control in Packages

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. As it relates to 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

	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

While Java's access control mechanism may seem complicated, we can simplify it as follows. Anything declared public can be accessed from different classes and different packages. Anything declared private cannot be seen outside of its class. When a member does not have an explicit access specification, it is visible to subclasses as well as to other classes in the same package. This is the default access. If you want to allow an element to be seen outside your current package, but only to classes that subclass your class directly, then declare that element protected.

The above table 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 outside its package. 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.

Example:

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);
  }
}
```

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);
  }
}
```

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);
  }
}
```

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);
  }
}
```

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();
  }
}
```