Inheritance: Extending a Class

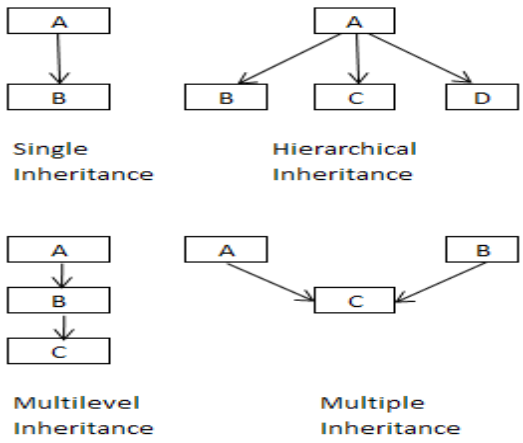
Inheritance is the capability of a class to use the properties and methods of another class while adding its own functionality. Java supports the concept of Reusability means reuse something that already exists rather than creating the same all over again. The mechanism of deriving a new class from an old one is called ***inheritance***. The old class is known as the ***base class*** or ***super class*** or ***parent class*** and the new one is called the ***subclass*** or ***derived class*** or ***child class*.** The inheritance allows subclasses to inherit all the variables and methods of their parent classes.

Types of Inheritance:

- Single or simple Inheritance (only one super class)
- Multiple Inheritance (several super classes)
- Hierarchical Inheritance (one super class, many subclasses)
- Multiple Inheritance (Derived from a derived class)

Java does not directly implement multiple inheritances. This concept is implemented using a secondary inheritance part in the form of *interfaces*.

Different forms of inheritance



Types of Inheritance

Defining a subclass:

Class subclassname extends superclassname
{
variable declaration;
methods declaration;
}

The keyword extends signifies that the properties of the superclassname are extended to the subclassname. The subclass will now contain its own variables and methods as well those of the superclass.

Super

When we create super class that keeps the details of its implementation to itself (making it private). Sometime the variable and methods of super class are private, so they will be available to subclass but subclass cannot access it. In such cases there would be no way for a subclass to directly access these variables on its own. Whenever a subclass needs a reference to it immediate super class, it can do so by use of the keyword super.

Super have two general forms:

1. The first calls the super class's constructor.
2. The second is used to access a member (variable or method) of the super class that has been declared as private.

Program for use of super:

```
class Base1
{
protected int num;
public Base1(int n)
{
num = n;
System.out.println("\n\t1-parameter constructor of Base class.");
}
public void display()
{
System.out.println("\n\tDisplay of Base class: " + num);
}
}
class Derived1 extends Base1
{
private int num;
public Derived1(int n1,int n2)
{
super(n1);
//super(n);
num = n2;
System.out.println("\n\t1-parameter constructor of Derived class.");
}
public void display()
{
super.display();
System.out.println("\n\tDisplay of Derived class: " + num);
}
}
public class UseOfSuper
{
public static void main(String args[])
{
Derived1 ob = new Derived1(10,20);
ob.display();
}
}
```

Output:

```
1-parameter constructor of Base class.
1-parameter constructor of Derived class.
Display of Base class: 10
Display of Derived class: 20
```

Subclass Constructor:

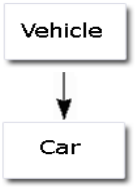
A subclass constructor is used to construct the instance variables of both the subclass and the superclass. The subclass constructor uses the keyword super to invoke the constructor method of the superclass. Conditions for using the super keyword are:

- Super may only be used within a subclass constructor method.
- The call to superclass constructor must appear as the first statement within the subclass constructor.
- The parameters in the super call must match the order and type of the instance variable declared in the superclass.

Types of Inheritance

1. Simple Inheritance (Single Level Inheritance):

When a subclass is derived simply from its parent class then this mechanism is known as ***simple inheritance.*** In case of simple inheritance there is only a sub class and its parent class. It is also called ***single inheritance*** or ***one level inheritance***.



Simple Inheritance

Program of single inheritance

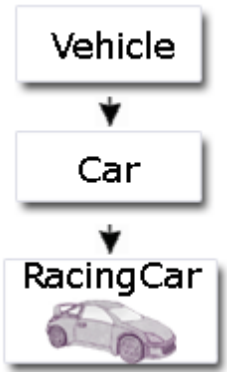
```
class A
{
int length, breadth;
A(int x, int y)
{
 length = x;
breadth = y;
}
int area()
{ return (length * breadth);
}
}

class B extends A          //inheriting class A
{
int height;
B(int x, int y, int z)
{
super (x, y);                  //pass values to superclass
height = z;
}
int volume()
{
return (length * breadth * height);
}
}
```

```
class SimpleInherTest
{
public static void main(String args[ ])
{
B ob1 = new B(14, 12, 10);
int area1 = ob1.area();              //superclass method
int volume1 = ob1.volume();      //baseclass method
System.out.println("Area = " + area1);
System.out.println("Volume = " + volume1);
}
}
```

Output:
Area = 168
Volume = 1680

2. Multilevel Inheritance:
When a subclass is derived from a derived class then this mechanism is known as the ***multilevel inheritance***. The derived class is called the subclass or child class for its parent class and this parent class works as the child class for it's just above (parent) class. Multilevel inheritance can go up to any number of levels. This concept allows built a chain of classes as Grandfather -> father -> child.
The class 'A' serves as a base class for the derived class 'B' which in turn serves as a base class for the derived class 'C'. the chain ABC is known as *inheritance path*.
Example of Multilevel inheritance

```
Class A
{…….……..}
Class B extends A   //first level
{…….……..}
Class C extends B   //second level
{…….……..}
```



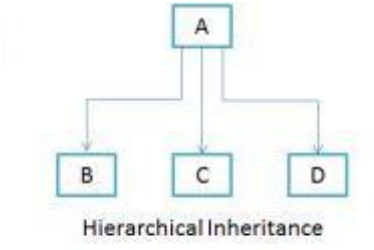
Program to implement Multilevel Inheritance

```
class Vehicle
{
public void display()
{
System.out.println("Super Class is Vehicle");
}
}
```

```
class Car extends Vehicle
{
public void disp()
{
System.out.println("Car acquires properties of Vehicle");
}
}

public class RacingCar extends Car
{
public void dis()
{
System.out.println("Racing Car is sub class");
}
public static void main(String args[])
{
RacingCar c1 = new RacingCar();
c1.dis();              // calling its own
c1.disp();             // calling super class Car method
c1.display();          // calling super class Vehicle
}
}
```

Output:
Racing Car is sub class
Car acquires properties of Vehicle
Super Class is Vehicle

3. Hierarchical Inheritance: In hierarchical type of inheritance, **one class is extended by many subclasses**. It is **one-to-many** relationship. In Simple sentence, Hierarchical inheritance is *"creating one or more child classes from the parent class".* Many programming problems can be cast into a hierarchy where certain features of one level are shared by many others below the level.



Example:

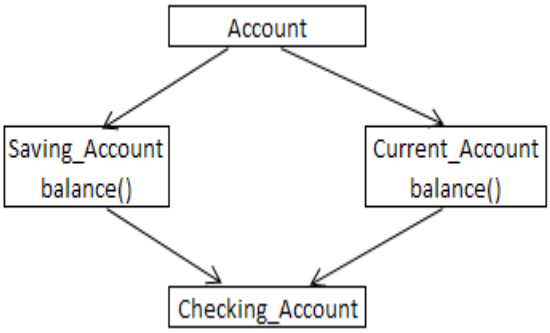
```
Class A
{
public void methodA()
{
System.out.println("method of Class A");
}
}
Class B extends A
{
public void methodB()
{
System.out.println("method of Class B");
}
}
Class C extends A
{
public void methodC()
{
System.out.println("method of Class C");
}
}
Class D extends A
{
public void methodD()
{
System.out.println("method of Class D");
}
}
Class MyClass
{
public void methodB()
{
System.out.println("method of Class B");
}
public static void main(String args[])
{
B obj1 = new B();
C obj2 = new C();
D obj3 = new D();
obj1.methodA();
obj2.methodA();
obj3.methodA();
}
}
```

Output:
Method of Class A
Method of Class A
Method of Class A

Problem with Multiple Inheritances:

As the name suggests, inheritance means to take something that is already made. It is the concept that is used for reusability purpose. The mechanism of inheriting the features of more than one base class into a single class is known as multiple inheritance. Java does not support multiple inheritance but the multiple inheritance can be achieved by using the interface. In Java Multiple Inheritance can be achieved through use of Interfaces by implementing more than one interfaces in a class.

When one subclass is having two or more super classes then such type of inheritance is called multiple inheritance. The subclass will acquires all properties of super classes. Saving_Account and Current_Account are both derived from Account. However, a Checking_Account incorporates the functionality of both Saving_Account and Current_Account. Both the subclasses Saving_Account and Current_Account have one method balance() for checking account balance. Checking_Account get all the features of both the classes. When we are creating object of class Checking_Account and calling balance() method, then there will be problem of which balance() method will be called. This is problem because of multiple inheritance as subclass Checking_Account have two different implementation of balance() method. Here complier will confused which method is follows for execution. This leads ambiguity problem. This is the reason java does not support multiple inheritance.



Multiple Inheritance

- Java does not support multiple inheritances.
- Classes in java cannot have more than one superclass.
- class A extends B extends C
  { ----- ----- }
  Is not permitted in java.

Java Interface

An interface is a collection of abstract methods. An interface is basically a kind of class. Writing an interface is similar to writing a class, with abstract methods and final fields. This means that interfaces do not specify any code to implement these methods and data fields contain only constants. Therefore it is the responsibility of the class that implements an interface to define the code for implementation of these methods. Interfaces cannot be instantiated—they can only be *implemented* by classes or *extended* by other interfaces.

Interfaces are designed to support dynamic method resolution at run time. Normally, in order for a method to be called from one class to another, both classes need to be present at compile time so the Java compiler can check to ensure that the method signatures are compatible. This requirement by itself makes for a static and non extensible classing environment. Inevitably in a system like this, functionality gets pushed up higher and higher in the class hierarchy so that the mechanisms will be available to more and more subclasses. Interfaces are designed to avoid this problem. They disconnect the definition of a method or set of methods from the inheritance hierarchy. Since interfaces are in a different hierarchy from classes, it is possible for classes that are unrelated in terms of the class hierarchy to implement the same interface. This is where the real power of interfaces is realized.

Note: Interfaces add most of the functionality that is required for many applications which would normally resort to using multiple inheritance in a language such as C++.

Defining an Interface:

An interface is defined much like a class. This is the general form of an interface:

Syntax:

```
access interface interfacename
{
Variable declaration;
Method declaration;
}
```

Access is either public or not used. When no access specifier is included, then default access results, and the interface is only available to other members of the package in which it is declared. When it is declared as **public**, the interface can be used by any other code. The **interface** keyword is used to declare an interface. **Interfacename** is any valid java variable. Methods are declared have no bodies and essentially abstract method. Methods end with a semicolon after the parameter list. They are, essentially, abstract methods; there can be no default implementation of any method specified within an interface. Each class that includes an interface must implement all of the methods. Variables can be declared inside of interface declarations. Variables are implicitly final and static, meaning they can't be change by the implementing class. They must also be initialized with a constant value. All methods and variables are implicitly **public** if the interface, itself, is declared as **public**.

Variables are declared as follows: static final type variable = value;

- All variables are declared as constants.
- Methods declaration will contain only a list of methods without body statements.
- Example: return_type methodname (parameter_list);

Example of interface definition:

```
interface person
{
static final int code=11;
static final string name="Vijay";
void display();
}
```

```
interface Area
{
final static float pi=3.14f;
float compute (float x, float y);
void show();
}
```

Extending Interfaces

An interface can extend another interface, similarly to the way that a class can extend another class. The **extends** keyword is used to extend an interface, and the child interface inherits the methods of the parent interface.

Syntax:

```
interface class2 extends class1
{
Body of class2
}
```

For example, we can put all the constants in one interface and the methods in the other. This will enable us to use the constants in classes where the methods are not required.

Example:

```
interface A
{
int code=11;
string name=”Computer”;
}
interface B extends  A
{
void display();
}
```

The interface B inherits both the constants code and name into it. Variables name **code** and **name** are declared like simple variables. It is allowed because all the variables in an interface are treated as constants although the keywords **final** and **static** are not present. We can also combine several interfaces together into a single interface.

Program for extending interface:

```
import java.lang.*;
interface A
{
void show();
}

interface B extends A
{
void display();
}

class C implements B
{
public void show()
{
System.out.println("Calling interface A");
}
```

```
public void display()
{
System.out.println("Calling interface B");
}
public void disp()
{
System.out.println("We are in Class C");
}
}

class D
{
public static void main(String args[])
{
C ob1=new C();
ob1.show();
ob1.display();
ob1.disp();
}
}
```

```
Calling interface A
Calling interface B
We are in Class C
```

While interfaces are allowed to extend to other interfaces, sub-interface cannot define the methods declared in the super-interface. Because sub-interface are still interface, not a classes. It is the responsibility of any class that implements the all methods of derived interface. When an interface extends two or more interfaces, they are separated by commas. Note that Interface cannot extend classes. This would violate the rule that an interface can have only abstract methods and constants.

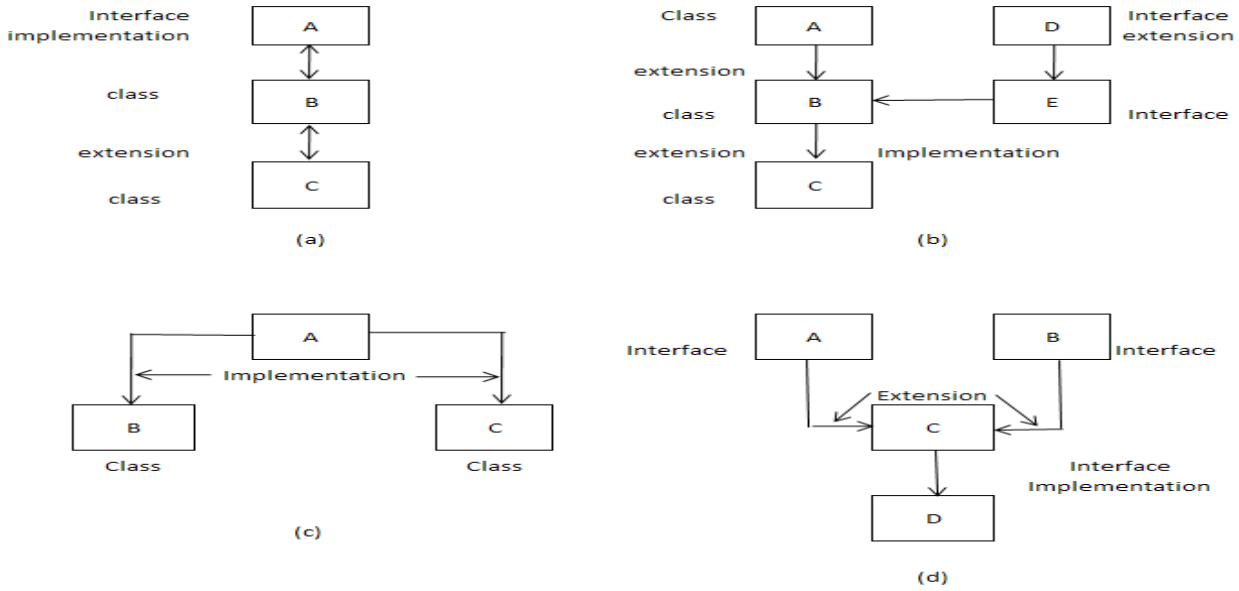
Implementing Interfaces

A Class can only inherit from one super class. However, a class may implement several Interfaces. To declare a class that implements an interface, you include an `implements` clause in the class declaration. Your class can implement more than one interface, so the `implements` keyword is followed by a comma-separated list of the interfaces implemented by the class. By convention, the `implements` clause follows the `extends` clause, if there is one.

Syntax:

```
class classname implements interfacename
{
Body of classname;
}
Or
class classname extends superclass implements interface1, interface2, ---
{
Body of classname;
}
```

In the above syntax, the class can extends another class while implements one or more interface, separated by a comma. The implementation of interface can take various forms as below:



Various forms of interface implementation

Example program of implementing interface:

```
public class Main
{
public static void main(String[] args)
{
shape circleshape=new circle();
circleshape.Draw();
}
}

interface shape
{
public   String baseclass="shape";
public void Draw();
}

class circle implements shape
{
public void Draw()
{
System.out.println("Drawing Circle here");
}
}
```

Accessing interface variables

Interfaces can be used to declare a set of constants that can be used in different classes. This is similar to creating header files in C++ to contain a large number of constants. Such interface does not contain methods; there is no need to worry about implementing any methods. The constant values will be available to any class that implements the interface. The values can be used in any method, as part of any variable declaration, or anywhere where we can use a final value. All variables declared in the interface must be constant. So if any interface contains the variable and if a class implements that interface then all of the variables will be available to the class implementing that interface.

Example:

```
interface A
{
int x=10;
int y=20;
}
class B implements A
{
int m=x;
void methodB(int size)
{…….}
}
```

Program for implementing interface and accessing interface variables:

```
/* Program to show a implementing two interfaces */
 /* First interface */
 interface One
{
 int x=12;
 }

 /* Second interface */
 interface Two
 {
 int y=10;
 void display();
 }

/* Class implementing the interfaces */
 class Demo implements One,Two
 {
int a=x;            // accessing value from interface one
int b=y;            // accessing value from interface two
public void display()
{
 System.out.println("X in inteface One ="+x);
 System.out.println("Y in interface Two="+y);
 System.out.println("X+Y= "+(x+y));
 }
```

```
public void disp()
{
System.out.println("Value access from interface one is="+a);
System.out.println("Value access from interface two is="+b);
 }
 }
  /* Main class */
 class TwoInterface
 {
 public static void main(String args[])
 {
 Demo d=new Demo();
 d.display();
 d.disp();
 }
 }
```

```
X in inteface One =12
Y in interface Two=10
X+Y= 22
Value access from interface one is=12
Value access from interface two is=10
```

Similarities between classes and interfaces

- They are both java basic object types.
- They both can contain variables and methods (With difference being class methods have implementation code whereas the interface methods can only have declarations).
- They can both be inherited using Inheritance (extends keyword for classes and implements keyword for interfaces).
- A class can be instantiated but an interface cannot.
- Classes and Interfaces are both Java files. They are saved as .java files with the name as the public class or public interface name.
- The difference is that, classes are fully coded. That is, they have all the necessary variables and methods coded in them. But in case of Interfaces, only methods are declared. They are not defined. Interfaces are only like skeletons whereas classes can implement interfaces and complete the skeleton.

Differences between an interface and a class.

Class	Interface
Classes are not used to implement multiple inheritence.	The interfaces are used in java to implementing the concept of multiple inheritance.
The member of a class can be constant or variables.	The members of an interface are always declared as constant i.e. their values are final.
The class definition can contain the code for each of its methods. That is the methods can be abstract or non-abstract.	The methods in an interface are abstract in nature. I.e. there is no code associated with them. It is defined by the class that implements the interface.
Class contains executable code.	Interface contains no executable code.
Memory is allocated for the classes.	We are not allocating the memory for the interfaces.
We can create object of class.	We can't create object of interface.
Classes can be instantiated by declaring objects.	Interface cannot be used to declare objects. It can only be inherited by a class.

Classes can use various access specifiers like public, private or protected.	Interface can only use the public access specifier.
Class contains constructors.	An interface does not contain any constructor.
Classes are always extended.	Interfaces are always implemented.
A class can extend only one class (no multiple inheritance), but it can implement many interfaces.	Interfaces can extend one or more other interfaces. Interfaces cannot extend a class, or implement a class or interface.
An abstract class is fast.	An interface requires more time to find the actual method in the corresponding classes.

Difference between Interface and Abstract class

Interface	Abstract class
The **interface keyword** is used to declare interface.	The **abstract keyword** is used to declare abstract class.
Interface **supports multiple inheritance**	Abstract class **doesn't support multiple inheritance**.
Interface **can't provide the implementation of abstract class**.	Abstract class **can provide the implementation of interface**.
Implements keyword is used to inherit an interface.	Extends keyword is used to inherit a class.
A class can implement any number of interfaces	A class can extend only one abstract class
All variables declared in an interface are by default public, static or final.	Abstract class **can have final, non-final, static and non-static variables**.
Interface can have only public abstract methods i.e. by default	abstract class can have protected , public and public abstract methods
Interfaces do not have any constructors.	Abstract classes can have constructors
Methods in an interface cannot be static	Non-abstract methods can be static

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

Nested Interface:

An interface which is declared within another interface or class is known as nested interface. The nested interfaces are used to group related interfaces so that they can be easy to maintain. The nested interface must be referred by the outer interface or class. It can't be accessed directly.

- Nested interface must be public if it is declared inside the interface but it can have any access modifier if declared within the class.
- Nested interfaces are declared static implicitly.

Syntax: Nested interface which is declared within the interface.

```
interface interface_name
{
...
```

```
interface nested_interface_name
{
...
}
}
```

Example of Nested interface which is declared within the interface.

```
interface display
{
void show();
interface Message
{
void msg();
}
}
class Test implements display.Message
{
public void msg()
{
System.out.println("Hello nested interface");
}
public static void main(String args[])
{
display.Message message=new Test();     //upcasting here
message.msg();
}
}
```

Output:

Hello nested interface

Syntax: Nested interface which is declared within the class.

```
class class_name
{
...
interface nested_interface_name
{
...
}
}
```

Example of Nested interface which is declared within the class.

```
class A
{
interface Message
{
void msg();
}
```

```
}

class Test implements A.Message
{
public void msg()
{
System.out.println("Hello nested interface");}
public static void main(String args[])
{
A.Message message=new Test();          //upcasting here
message.msg();
}
}
```

Output:

Hello nested interface

Object Cloning

The object cloning is a way to create exact copy of an object. For this purpose, clone() method of Object class is used to clone an object. The **java.lang.Cloneable interface** must be implemented by the class whose object clone we want to create.

```
class Student implements Cloneable
{
int rollno;
String name;

Student(int rollno,String name)
{
this.rollno=rollno;
this.name=name;
}



public Object clone()throws CloneNotSupportedException
{
return super.clone();
}

public static void main(String args[])
{
Try
{
Student s1=new Student(11,"Amit");
Student s2=(Student)s1.clone();
System.out.println(s1.rollno+" "+s1.name);
System.out.println(s2.rollno+" "+s2.name);
```

```
}
catch(CloneNotSupportedException c)
{}
}
}
```

Output:

```
11 Amit
11 Amit
```

Java Packages

Packages are nothing more than the way we organize files into different directories according to their functionality, usability as well as category they should belong to. Packaging also helps us to avoid class name collision when we use the same class name as that of others. Packages are similar to "class libraries". Packages act as "containers" for classes.

The Benefits of organizing our classes into packages are:

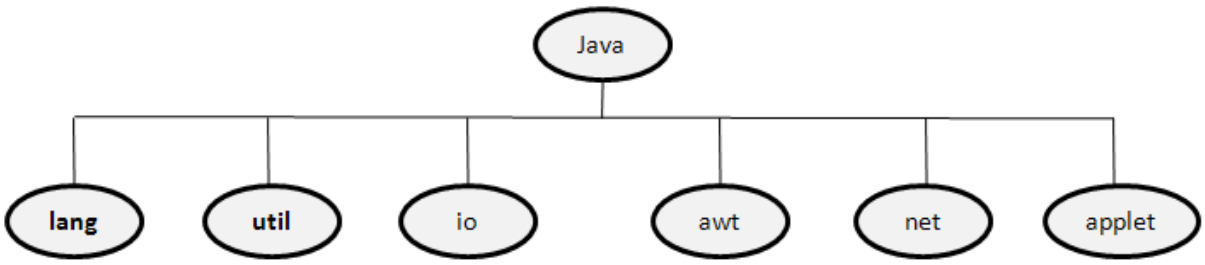
1. Packages are used to bundle classes and interfaces.
2. The classes contained in the packages of other programs/applications can be easily reused.
3. In packages, classes can be unique compared with classes in other classes. I.e. two classes in two different packages can have the same name.
4. Packages provide a way to hide classes thus preventing other programs or packages from accessing classes that are meant for internal use only.
5. Packages also provide a way for separating design from coding. First we design classes and decide their relationships, 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.
6. Name space collision is minimized.

Java packages are classified into two types.

- Java API Packages
- User defined Packages

Java API Packages

Java API is actually a huge collection of library routines that performs basic programming tasks such as looping, displaying GUI form etc. Most of the time we use the packages available with the java API. In the Java API classes and interfaces are packaged in packages. All these classes are written in Java programming language and runs on the JVM.

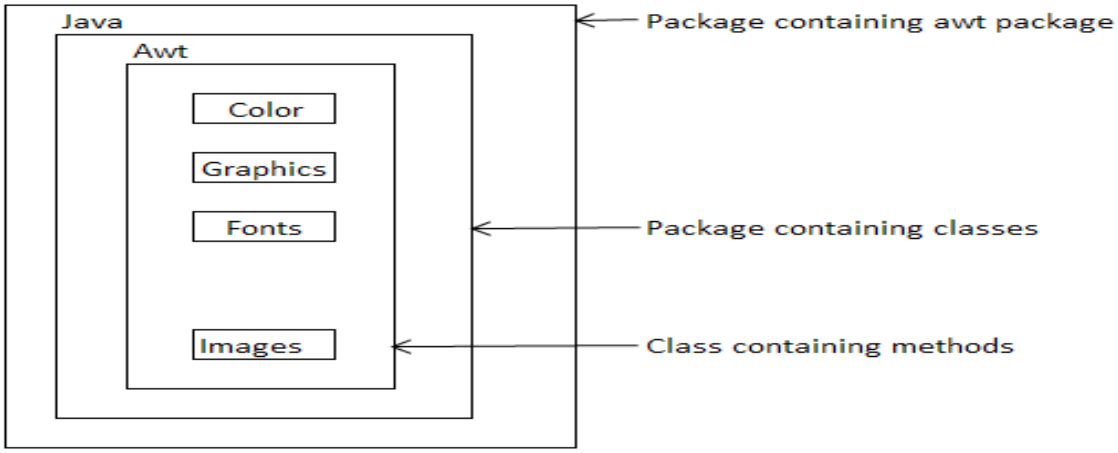


Frequently used API packages

Java System packages and their classes are:

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

Using System Packages



Hierarchical representation of java.awt package

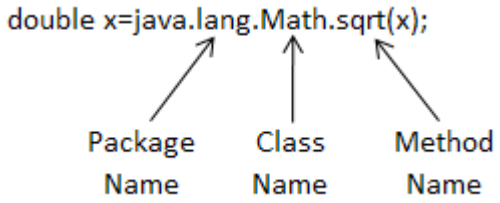
The package names java contains the package awt, which contains various classes required for implementing graphical user interface.

Classes stored in packages can be accessed in two ways:

1) To use the fully qualified class name of the class that we want to use. This is done by using the package name containing the class and then appending the class name to it using the dot operator. Example **java.awt.color**
2) In many situations, we might want to use a class in a number of places in the program or we may like to use many of the classes contained in a package. This is achieve as follows:
   import packagename.classname or import packagename.*
   Example: import java.awt.*;
   These are known as import statements and must appear at the top of the file, before any class declarations, import is a keyword.

Naming Conventions

Package names are written in all lower case to avoid conflict with the names of classes or interfaces. All class names begin with an uppercase letters and method names begin with lowercase letters.
Example:



Creating Packages

A package is a namespace that organizes a set of related classes and interfaces. Conceptually you can think of packages as being similar to different folders on your computer. We declare the name of the package using the package keyword followed by a package name. The **package** statement should be the first line in the source file.
Example:

```
package world                    //package declaration
public class HelloWorld          //class definition
{
public static void main(String args[])
{
System.out.println("Hello World");      //body of class
}
```

Here the package name is **world.** The class **HelloWorld** is considered a part of this package. This listing would be saved as a file called **HelloWorld.java**, and located in directory named **world.** When the source file is compiled, java will create a **.class** file and store in the same directory.

Steps for creating User defined package:

- Declare the package at the beginning of a file using the form: package packagename;
- Define the class that is to be put in the package and declare it **public.**
- Create a subdirectory under the directory where the main source files are stored.
- Store the listing as the classname.java file in the subdirectory created.
- Compile the file. This creates **.class** file in the subdirectory.

Accessing Package

If a class wants to use another class in the same package, the package name does not need to be used. Classes in the same package find each other without any special syntax.

Two ways to accessed java system package are:

1. Java system package can be accessed using a fully qualified class name through the **import** statement. The import statement can be used to search a list of package for a particular class.
   Syntax: import package1 [.package2] [.package3].classname;

Here package1 is the name of the top level package, package2 is the name of the package that is inside the package1 and so on. Finally the explicit classname is specified. The statement must end with a semicolon (;) The import statement should appear before any class definitions in a source file.

2. Another approach The package can be imported using the import keyword and the wild card (*)
   Syntax: import packagename.*;
   Drawback of the shortcut approach is that it is difficult to determine from which package a particular member came.

Using a Package

- We can use classes from other packages. Let store the code listing below in the file "ClassA.java" within the subdirectory named "package1" within the current directory.

```
package package1;
public class ClassA
{
Public void displayA()
{
System.out.println("Class A");
}
}
```

- Compile this "ClassA.java" file and ClassA. class file will be stored in the same subdirectory.

```
import package1. ClassA;
class PackageTest1
{
Public static void main(String args[]);
{
ClassA objectA= new ClassA();
objectA.displayA();
}
}
```

- In above program the class ClassA import from the package package1. The source file saved as "PackageTest1.java" and compiled. The source file and compiled file would be saved in the directory of which package1 was a subdirectory. Now run the program and obtain the result.
- During compilation of PackageTest1.java the compiler checks for the file "ClassA.class" in package1 directory for information it needs, but it does not actually include the code from ClassA.class in the file PackageTest1.class. When the PackageTest1 program is run, java looks for the file PackageTest1.class and loads it using something called class loader. Now the interpreter knows that it also needs the code in the file ClassA.class and loads it as well.
- Let us consider another package named is package2

```
package package2;
public class ClassB
{
protected int m=10;
Public void displayB()
{
System.out.println("Class B");
```

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

- Importing classes from other packages:

```
import package1.ClassA;
import package2.*;
class PackageTest2
{
Public static void main(String args[]);
{
ClassA objectA= new ClassA();
ClassB objectB= new ClassB();
objectA.displayA();
objectB.displayB();
}
}
```

This program are saved as PackageTest2.java, compiled and run to obtain result:
Output:
Class A
Class B
m=10

When we import packages it is likely that two or more packages contain classes with identical names.
For example:

```
package p1;
public class Teacher
{……}
public class Student
{……}
package p2
public class courses
{……}
public class Student
{……}
```

We can import and use these packages like:

```
import p1.*;
import p2.*;
student student1;
```

Both the p1 and p2 packages contain the class student, compiler cannot understand which one to use and generate error. We can specify the package name more explicitly.
Example:

```
import p1.*;
import p2.*;
p1.student student1;     //OK
p2.student student2;     //OK
```

Teacher teacher1; //No error
Courses course1; // No error

Adding a class to a Package

Consider the following package

```
package p1;
public ClassA
{
// body of A
}
```

The package p1 contains on public class by name A. suppose we want to add another class B to this package. This can be done as follows:

- Define the class and make it public.
- Place the package statement.

```
package p1;
```

Before the class definition as follows

```
package p1;
public class B
{
// body of B
}
```

- Store this as B.java file under the directory p1
- Compile B.java file. This will create a B.class file and place it in the directory p1.

Note that we can also add a non public class to a package using the same procedure. Now the package p1 will contain both the classes A and B. A statement like

```
import p1.*;
```

Will import both class A and B from package p1. Remember that, since a java source file can have only one class declared as **public,** we cannot put two or more public classes together in a **.java** file. This is because of the restriction that the file name should be same as the name of the public class with **.java** extension.

Following are the steps to create a package with multiple public classes in it.

1. Decide the name of the package.
2. Create a subdirectory with this name under the directory where main source files are stored.
3. Create classes that are to be placed in the package in separate source files and declare the package statement: ***package packagename;*** at the top of each source file.
4. Switch to the subdirectory created earlier and compile each source file. When completed, the package would contain **.class** files of all the source files.

Import Statement

Import keyword is used to import built-in and user defined packages into your java source file. So that your class can refer a class that is in another package by directly using its name.

Different way to refer to class that is present in different packages.

1) Import the only class you want to use

```
Import java.util.Date;
Class MyDate extends Date
{
```

```
// Statements
}
```

2) import all the classes from the particular package

```
Import java.util.*;
Class MyDate extends Date
{
// Statements
}
```

Import statement is kept after the package statement. For example

```
Package P1;
Import java.util.*;
```

If you are not creating a package then import statement will be the first statement of your java source file.

Static import

Static import is a feature that expands the capabilities of import keyword. It is used to import static member of a class. Using static import, it is possible to refer to the static member directly without its class name. The name of the class and a dot(.) are not required to use an imported static member. There are to general form of static import statement.

1) Import only a single static member of a class

Syntax: import static package.class-name.static-member-name;

Example: import static java.lang.Math.sqrt;

2) Imports all the static member of a class

Syntax: import static package.class-name.*;

Example: import static java.lang.Math.*;

Example without using static import	Example using static import

Public class Test
{
Public static void main (String args[])
{
System.out.println(Math.sqrt(144));
}
}
	Import static java.lang.Math.*;
	Public class Test
	{
	Public static void main (String args[])
	{
	System.out.println(sqrt(144));
	}
	}

Package Visibility

- Packages acts as container for classes and other package and classes acts as container for data and methods. Data member and methods can be declared with the access protection modifiers such as private, protected and public.
- Access level modifiers determine whether other classes can use a particular field or invoke a particular method.
- A class may be declared with the modifier `public`, in which case that class is visible to all classes everywhere. Applied to method or variable completely visible

- The `private` modifier specifies that the member or methods can visible or only be accessed within that class. Private members are not visible within subclasses and are not inherited.
- The `protected` modifier specifies that the member can only be accessed within its own package and, in addition, by a subclass of its class in another package.

Access protection table

Access Modifer → / Access Location ↓	Public	Protected	Friendly (default)	Private Protected	Private
Same Class	Yes	Yes	Yes	Yes	Yes
Subclass in same package	Yes	Yes	Yes	Yes	No
other classes in same package	Yes	Yes	Yes	No	No
subclass in other package	Yes	Yes	No	Yes	No
non-subclasses in other package	Yes	No	No	No	No