Inheritance

Inheritance

- It support the creation of hierarchical classification and reusability
- A class that is inherited is called a superclass/base class/parent class.
- The class that does the inheriting is called a subclass/derived class/child class.
- Therefore, a subclass is a specialized version of a superclass.
- It inherits all of the instance variables and methods defined by the superclass and adds its own, unique elements.
- To inherit a class, you simply incorporate the definition of one class into another by using the ***extends*** keyword.

Syntax of Java Inheritance

```
class subclass-name extends superclass-name
{
    //variable declaration
    // method delaration
}
```

- Keyword extends signifies that the properties of the superclass are extended to the subclass.
- Subclass will now contain its own variables and methods as well as those in super class.

```
class A {
   int i, j;
   void showij() {
      System.out.println("i and j: " +
  i + " " + j);
 }
}
class B extends A {
  int k;
  void showk() {
    System.out.println("k: " + k);
   }
   void sum() {
      System.out.println("i+j+k: " +
  (i+j+k));
   }
}
```

```
class SimpleInheritance {
public static void main(String args[]) {
   A superOb = new A();
   B subOb = new B();

   superOb.i = 10;
   superOb.j = 20;

 System.out.println("Contents of superOb:");
  superOb.showij();
  System.out.println();

  subOb.i = 7;
  subOb.j = 8;
  subOb.k = 9;

  System.out.println("Contents of subOb: ");
  subOb.showij();
  subOb.showk();
  System.out.println();
  System.out.println("Sum of i, j and k in
    subOb:");
  subOb.sum();
}
}
```

Output

Contents of superOb:
i and j: 10 20

Contents of subOb:
i and j: 7 8
k: 9

Sum of i, j and k in subOb:
i+j+k: 24

- Subclass B includes all of the members of its superclass A.
- This is why subOb can access i and j and call showij( ).
- Also, inside sum( ), i and j can be referred to directly, as if they were part of B.
- Even though A is a superclass for B, it is also a completely independent, stand-alone class.
- A subclass can be a superclass for another subclass.

- Although a subclass includes all of the members of its superclass, it cannot access those members of the superclass that have been declared as *private*.

```
class A {
  int i;
  private int j;
  void setij(int x, int y) {
        i = x;
        j = y;
  }
}

class B extends A {
  int total;
  void sum() {
        total = i + j;
  }
}
```

```
class Access {
  public static void main(String
   args[]) {
        B subOb = new B();
        subOb.setij(10, 12);
        subOb.sum();
        System.out.println("Total
  is " + subOb.total);
  }
}
```

```
D:\java\bin>javac Access.java
Access.java:13: j has private access in A
total = i + j; // ERROR, j is not accessible here
              ^
1 error
```

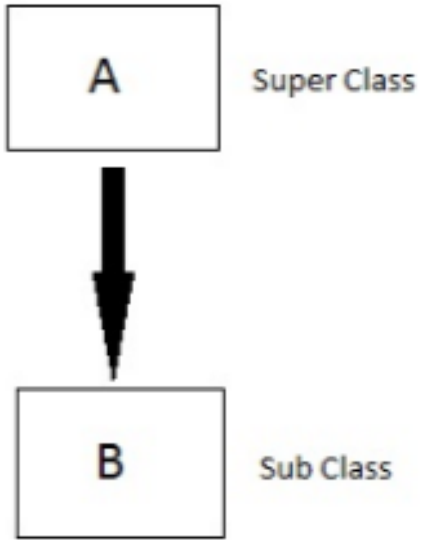
- **Since j is declared as private, it is only accessible by other members of its own class. Subclasses have no access to it.**

Types of Inheritance

1.Single Inheritance

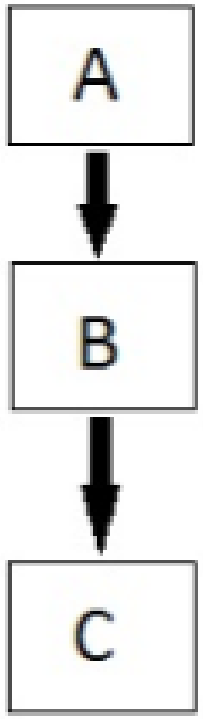
- Structure having one and only one super class as well as sub class

```
Class A
{
//variables and methods
}
Class B extends A
{
  //variables and methods
}
```



2.Multilevel Inheritance

```
Class A
{
//variables and methods
}
Class B extends A
{
  //variables and methods
}
Class C extends B
{
  //variables and methods
}
```



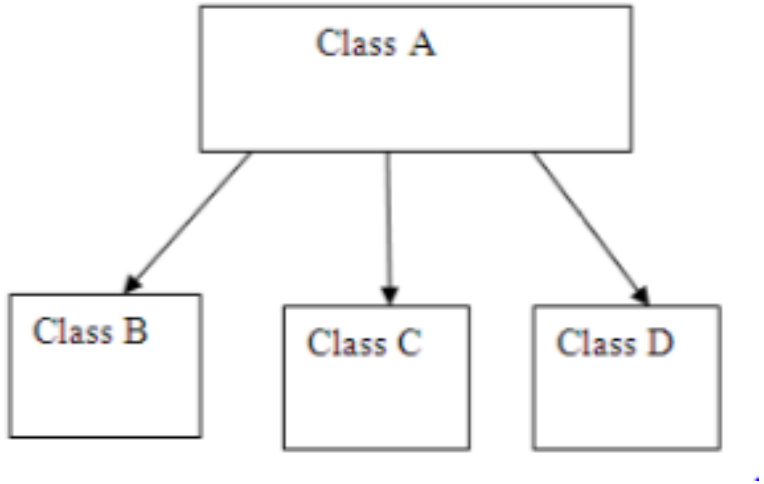
```
class Animal{
  void eat(){System.out.println("eating...");}
}
class Dog extends Animal{
  void bark(){System.out.println("barking...");}
}
class BabyDog extends Dog{
   void weep(){System.out.println("weeping...");}
}
class TestInheritance2{
  public static void main(String args[]){
            BabyDog d=new BabyDog();
            d.weep();
            d.bark();
            d.eat();
  }}
```

3.Hybrid Inheritance

- A structure having one parent and more child class

```
Class A {
//variables and methods
}
Class B extends A {
  //variables and methods
}
Class C extends A {
  //variables and methods
}
```

```
Class D extends A
{
  //variables and methods
}
```



Constructors in Multilevel Hierarchy

- In a class hierarchy, constructors are called in order of derivation, from superclass to subclass.
- Since super() must be the first statement executed in a subclass' constructor, this order is the same whether or not super() is used.
- If super() is not used, then the default or parameter-less constructor of each superclass will be executed.

```
class A {
A( ) {
System.out.println("Inside A's constructor.");
}
}

class B extends A {
B ( ) {
System.out.println("Inside B's constructor.");
}
}

class C extends B {
C( ) {
System.out.println("Inside C's constructor.");
}
}

class CallingCons {
public static void main(String args[]) {
C c = new C( );
}
}
```

```
class A {
A( ) {
System.out.println("Inside A's constructor.");
}
}
// Create a subclass by extending class A.
class B extends A {
B ( ) {
System.out.println("Inside B's constructor.");
}
}
// Create another subclass by extending B.
class C extends B {
C( ) {
System.out.println("Inside C's constructor.");
}
}
class CallingCons {
public static void main(String args[]) {
C c = new C( );
}
}
```

OUTPUT

```
D:\java\bin>java CallingCons
Inside A's constructor.
Inside B's constructor.
Inside C's constructor.
```

Using super() to call superclass constructor

- A subclass can call a constructor defined by its superclass by use of the following form of super

  super(parameter-list);

- Parameter-list specifies any parameters needed by the constructor in the super class

- The keyword super can be used with the following conditions:

a. Only the subclass constructor uses the super keyword
b. A call to the superclass constructor, super() must always be the first statement executed inside a subclass constructor
c. The parameter in the super() call must match the order and types declared in the superclass constructor

SUPER

BoxWeight now uses super to initialize its Box attributes.

```
Class Box{
   double length,width,depth;
   Box(int l,int w, int d){
            length=l;
            width=w;
            depth=d;                }
}
class BoxWeight extends Box {
   double weight;
   BoxWeight(double w, double h, double d, double m) {
            super(w, h, d);
            weight = m;          }
}
```

```
class A{
    int i,j;
    A(int a,int b){
        i=a;
        j=b;
    }
}
class B extends A{
    int k;
    B(int a,int b,int c){
        super(a,b);
        k=c;
    }
    void sum(){
        System.out.println(i+j+k);
    }
}
```

```
class SuperDemo {
    public static void main(String[] args) {
        B b=new B(10,20,30);
        b.sum();
    }
}
```

Method Overriding

Method Overriding

- Subclasses inherit all methods from their superclass
  - Sometimes, the implementation of the method in the superclass does not provide the functionality required by the subclass.
  - In these cases, the method must be overridden.
- To override a method, provide an implementation in the subclass.
  - The method in the subclass MUST have the exact same signature as the method it is overriding.

Method Overriding

- In a class hierarchy, when a method in a subclass has the same name and type signature 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 will always refer to the version of that method defined by the subclass.
- The version of the method defined by the superclass will be hidden.

Usage of Java Method Overriding

- Method overriding is used to provide the specific implementation of a method which is already provided by its superclass.
- Method overriding is used for runtime polymorphism

Rules for Java Method Overriding

- The method must have the same name as in the parent class
- The method must have the same parameter as in the parent class.
- There must be an IS-A relationship (inheritance).

```
class A {
        int i, j;
        A(int a, int b) {
                i = a;
                j = b;
        }
        void show() {
        System.out.println("i and j: " + i + " " + j);
}}

class B extends A {
        int k;
        B(int a, int b, int c) {
                super(a, b);
                k = c;
        }
        void show() {
                System.out.println("k: " + k);
}}
class Override {
public static void main(String args[]) {
        B subOb = new B(1, 2, 3);
        subOb.show(); // this calls show() in B
}}
```

```
class A {
int i, j;
A(int a, int b) {
i = a;
j = b;
}
void show() {
System.out.println("i and j: " + i + " " + j);
}}
class B extends A {
int k;
B(int a, int b, int c) {
super(a, b);
k = c;
}
void show() {
System.out.println("k: " + k);
}}
class Override {
public static void main(String args[]) {
B subOb = new B(1, 2, 3);
subOb.show(); // this calls show() in B
}
}
```

OUTPUT

```
D:\java\bin>java Override
k: 3
```

Method overriding

- Method overriding occurs only when the names and the type signatures of the two methods are identical.
- If they are not, then the two methods are simply overloaded.

```
class A {
int i, j;
A(int a, int b) {
i = a;
j = b;
}
void show() {
System.out.println("i and j: " + i + " " + j);
} }
class B extends A {
int k;
B(int a, int b, int c) {
super(a, b);
k = c;
}
void show(String msg) {
System.out.println(msg + k);
} }
class Override2 {
public static void main(String args[]) {
B subOb = new B(1, 2, 3);
subOb.show("This is k: ");
subOb.show();
} }
```

```
class A {
int i, j;
A(int a, int b) {
i = a;
j = b;
}
void show() {
System.out.println("i and j: " + i + " " + j);
} }
class B extends A {
int k;
B(int a, int b, int c) {
super(a, b);
k = c;
}
void show(String msg) {
System.out.println(msg + k);
} }
class Override2 {
public static void main(String args[]) {
B subOb = new B(1, 2, 3);
subOb.show("This is k: ");
subOb.show();
} }
```

OUTPUT

```
D:\java\bin>java Override2
This is k: 3
i and j: 1 2
```

Polymorphism

- **Polymorphism in Java** is a concept by which we can perform a *single action in different ways*.
- Polymorphism is derived from 2 Greek words: poly and morphs.
- The word "poly" means many and "morphs" means forms. So polymorphism means many forms.
- There are two types of polymorphism in Java: compile-time polymorphism and runtime polymorphism.
- We can perform polymorphism in java by method overloading and method overriding.

Dynamic Method Dispatch

- Method overriding forms the basis for one of Java's most powerful concepts: dynamic method dispatch.
- Dynamic method dispatch is the mechanism by which a call to an overridden method is resolved at run time, rather than compile time.
- Dynamic method dispatch is important because *this is how Java implements run-time polymorphism*.

Dynamic Method Dispatch

- A superclass reference variable can refer to a subclass object.
- Java uses this fact to resolve calls to overridden methods at run time.
- When an overridden method is called through a superclass reference, Java determines which version of that method to execute based upon the type of the object being referred to at the time the call occurs.
- Thus, this determination is made at run time.

Dynamic Method Dispatch

- When different types of objects are referred to, different versions of an overridden method will be called.
- In other words, *it is the type of the object being referred to* (*not the type of the reference variable*) that determines which version of an overridden method will be executed.
- Therefore, if a superclass contains a method that is overridden by a subclass, then when different types of objects are referred to through a superclass reference variable, different versions of the method are executed.

```
// Dynamic Method Dispatch
class A {
void callme() {
System.out.println("Inside A's callme
method");
}
}
class B extends A {
// override callme()
void callme() {
System.out.println("Inside B's
callme method");
}
}
class C extends A {
// override callme()
void callme() {
System.out.println("Inside C's
 callme method");
}
}
```

```
class Dispatch {
public static void main(String args[]) {
A a = new A(); // object of type A
B b = new B(); // object of type B
C c = new C(); // object of type C

A r; // obtain a reference of type A

r = a; // r refers to an A object
r.callme(); // calls A's version of callme

r = b; // r refers to a B object
r.callme(); // calls B's version of callme

r = c; // r refers to a C object
r.callme(); // calls C's version of callme
}
}
```

```
// Dynamic Method Dispatch
class A {
void callme() {
System.out.println("Inside A's callme
method");
}
}
class B extends A {
// override callme()
void callme() {
System.out.println("Inside B's
callme method");
}
}
class C extends A {
// override callme()
void callme() {
System.out.println("Inside C's
 callme method");
}
}
```

```
class Dispatch {
public static void main(String args[]) {
A a = new A(); // object of type A
B b = new B(); // object of type B
C c = new C(); // object of type C
A r; // obtain a reference of type A
r = a; // r refers to an A object
r.callme(); // calls A's version of callme
r = b; // r refers to a B object
r.callme(); // calls B's version of callme
r = c; // r refers to a C object
r.callme(); // calls C's version of callme
}
}
```

OUTPUT

```
D:\java\bin> java Dispatch
Inside A's callme method
Inside B's callme method
Inside C's callme method
```

Dynamic Method Dispatch

- Dynamic, run-time polymorphism is one of the most powerful mechanisms that object- oriented design brings to bear on code reuse and robustness.
- The ability of existing code libraries to call methods on instances of new classes without recompiling while maintaining a clean abstract interface is a profoundly powerful tool.

```
class Figure {
double dim1;
double dim2;
Figure(double a, double b) {
dim1 = a;
dim2 = b;
}
double area() {
System.out.println("Area for Figure is
undefined.");
return 0;
}
}
class Rectangle extends Figure {
Rectangle(double a, double b) {
super(a, b);
}
double area() {
System.out.println("Inside Area
for Rectangle.");
return dim1 * dim2;
}
}
```

```
class Triangle extends Figure {
Triangle(double a, double b) {
super(a, b);
}
double area() {
System.out.println("Inside Area for
Triangle.");
return dim1 * dim2 / 2; } }
class FindAreas {
public static void main(String args[]) {
Figure f = new Figure(10, 10);
Rectangle r = new Rectangle(9, 5);
Triangle t = new Triangle(10, 8);
Figure figref;
figref = r;
System.out.println("Area is " +
figref.area());
figref = t;
System.out.println("Area is " +
figref.area());
figref = f;
System.out.println("Area is " +
figref.area()); }}
```

Abstract Classes

- There are situations in which you will want to define a superclass that declares the structure of a given abstraction without providing a complete implementation of every method.
- Sometimes you will want to create a superclass that only defines a generalized form that will be shared by all of its subclasses, leaving it to each subclass to fill in the details.
- Such a class determines the nature of the methods that the subclasses must implement.

Abstract Classes

- One way this situation can occur is when a superclass is unable to create a meaningful implementation for a method.
- This is the case with the class Figure used in the preceding example.
- The definition of area( ) is simply a placeholder. It will not compute and display the area of any

  type of object.

Abstract Classes

- You may have methods that must be overridden by the subclass in order for the subclass to have any meaning.
- Consider the class Triangle. It has no meaning if area() is not defined.
- In this case, you want some way to ensure that a subclass does, indeed, override all necessary methods.
- Java's solution to this problem is the *abstract* method.

Abstract Classes

- You can require that certain methods be overridden by subclasses by specifying the abstract type modifier.
- These methods are sometimes referred to as *subclasser responsibility*
- because they have no implementation specified in the superclass.
- Thus, a subclass must override them—it cannot simply use the version defined in the superclass. To declare an abstract method, use this general form: *abstract type name(parameter-list);*

Abstract Classes

- **Any class that contains one or more abstract methods must also be declared abstract.**
- **To declare a class abstract, you simply use the *abstract* keyword in front of the class keyword.**
- **There can be no objects of an abstract class.**
- That is, an abstract class cannot be directly instantiated with the new operator.
- Such objects would be useless, because an abstract class is not fully defined.

Abstract Classes

- Also, **you cannot declare abstract constructors, or abstract static methods.**
- **Any subclass of an abstract class must either implement all of the abstract methods in the superclass, or be itself declared abstract.**

```
abstract class A {
abstract void callme();
void callmetoo() {
System.out.println("This is a concrete method.");
}
}

class B extends A {
void callme() {
System.out.println("B's implementation of callme.");
}
}

class AbstractDemo {
public static void main(String args[]) {
B b = new B();
b.callme();
b.callmetoo();
}
}
```

```
abstract class A {
abstract void callme();
void callmetoo() {
System.out.println("This is a concrete method.");
}
}

class B extends A {
void callme() {
System.out.println("B's implementation of callme.");
}
}

class AbstractDemo {
public static void main(String args[]) {
B b = new B();
b.callme();
b.callmetoo();
}
}
```

OUTPUT

```
D:\java\bin>java AbstractDemo
B's implementation of callme.
This is a concrete method.
```

Abstract Classes

- Class A implements a concrete method called callmetoo().
- Although abstract classes cannot be used to instantiate objects, they can be used to create object references
- because Java's approach to run-time polymorphism is implemented through the use of superclass references.
- Thus, it must be possible to create a reference to an abstract class so that it can be used to point to a subclass object.

```
abstract class Figure {
double dim1;
double dim2;
Figure(double a, double b) {
dim1 = a;
dim2 = b;
}
abstract double area();
}
class Rectangle extends Figure {
Rectangle(double a, double b) {
super(a, b);
}
double area() {
System.out.println("Inside Area for Rectangle.");
return dim1 * dim2;
} }
class Triangle extends Figure {
Triangle(double a, double b) {
super(a, b);
}
double area() {
System.out.println("Inside Area for Triangle.");
return dim1 * dim2 / 2;
} }
```

```
class AbstractAreas {
public static void main(String
args[]){
// Figure f = new Figure(10, 10);
Rectangle r = new Rectangle(9, 5);
Triangle t = new Triangle(10, 8);
Figure figref;
figref = r;
System.out.println("Area is " +
figref.area());
figref = t;
System.out.println("Area is " +
figref.area());
} }
```

FINAL

- A variable can be declared as *final*.
- Doing so prevents its contents from being modified.
- This means that you must initialize a final variable when it is declared.
- For example: *final int FILE_NEW = 1;*
- Subsequent parts of your program can now use FILE_NEW as a constant without fear that the value has been changed.

FINAL

- It is a common coding convention to choose all uppercase identifiers for *final* variables.
- A final variable is essentially a constant.

Uses of FINAL

- The keyword final has three uses.

1.To create the equivalent of a named constant.

2.To prevent overriding

3.To prevent inheritance

Using final to Prevent Overriding

- To disallow a method from being overridden, specify final as a modifier at the start of its declaration.
- Methods declared as final cannot be overridden.

```
class A {
final void meth() {
System.out.println("This is a final method.");
}
}
```

```
class B extends A {
void meth() {
System.out.println("Illegal!");
}
}
```

- **Because meth( ) is declared as final, it cannot be overridden in B**

Using final to Prevent Overriding

- Methods declared as final can sometimes provide a performance enhancement
- The compiler is free to inline calls to them because it "knows" they will not be overridden by a subclass.
- When a small final method is called, often the Java compiler can copy the bytecode for the subroutine directly inline with the compiled code of the calling method,
- Thus eliminating the costly overhead associated with a method call.

Using final to Prevent Overriding

- Inlining is only an option with final methods. Normally, Java resolves calls to methods dynamically, at run time.
- This is called late binding. However, since final methods cannot be overridden, a call to one can be resolved at compile time. This is called early binding.

Using final to Prevent Inheritance

- precede the class declaration with final.
- Declaring a class as final implicitly declares all of its methods as final, too.
- It is illegal to declare a class as both abstract and final

```
final class A {
// ...
}

// The following class is illegal.
class B extends A { // ERROR!Can't subclass A
// ...
}
```

- It is illegal for B to inherit A since A is declared as final.

Package

- A unique name had to be used for each class to avoid name collisions.
- After a while, without some way to manage the name space, you could run out of convenient, descriptive names for individual classes.
- You also need some way to be assured that the name you choose for a class will be reasonably unique and not collide with class names chosen by other programmers.

Package

- Java provides a mechanism for partitioning the class name space into more manageable chunks.
- This mechanism is the package.
- The package is both **a *naming* and a *visibility control mechanism***.
- You can define classes inside a package that are not accessible by code outside that package.
- You can also define class members that are only exposed to other members of the same package.

Package

- Packages are containers for classes that are used to keep the class name space compartmentalized.
- For example, a package allows you to create a class named ***List***, which you can store in your own package without concern that it will collide with some other class named ***List*** stored elsewhere.
- Packages are stored in a hierarchical manner and are explicitly imported into new class definitions.

Package

- To create a package - *include a package command as the first statement in a Java source file.*
- Any classes declared within that file will belong to the specified package.
- The package statement defines a name space in which classes are stored.
- If you omit the package statement, *the class names are put into the default package, which has no name.*

Package

- general form of the package statement:
  - *package pkg;*
- Java uses file system directories to store packages.
- For example, the .class files for any classes you declare to be part of *MyPackage* must be stored in a directory called *MyPackage*.
- Case is significant, and the directory name must match the package name exactly.

Package

- More than one file can include the same package statement.
- The package statement simply specifies to which package the classes defined in a file belong.
- It does not exclude other classes in other files from being part of that same package.
- You can create a hierarchy of packages. To do so, simply separate each package name from the one above it by use of a period. The general form of a multileveled package statement
  - *Package pkg1[.pkg2[.pkg3]];*

```
package MyPack;

class Balance {
String name;
double bal;
Balance(String n, double b) {
name = n;
bal = b;
}
void show() {
if(bal<0)
System.out.print("--> ");
System.out.println(name + ": $" + bal);
} }

class AccountBalance {
public static void main(String args[]) {
Balance current[] = new Balance[3];
current[0] = new Balance("K. J. Fielding", 123.23);
current[1] = new Balance("Will Tell", 157.02);
current[2] = new Balance("Tom Jackson", -12.33);
for(int i=0; i<3; i++) current[i].show();
} }
```

```
package MyPack;

class Balance {
String name;
double bal;
Balance(String n, double b) {
name = n;
bal = b;
}
void show() {
if(bal<0)
System.out.print("--> ");
System.out.println(name + ": $" + bal);
} }

class AccountBalance {
public static void main(String args[]) {
Balance current[] = new Balance[3];
current[0] = new Balance("K. J. Fielding",
123.23);
current[1] = new Balance("Will Tell", 157.02);
current[2] = new Balance("Tom Jackson", -12.33);
for(int i=0; i<3; i++) current[i].show();
} }
```

- **Compile the file. Make sure that the resulting *.class* file is also in the *MyPack* directory.**

- **Remember, you will need to be in the directory above *MyPack* when you execute this command.**

OUTPUT

```
D:\java\bin>java MyPack.AccountBalance
K. J. Fielding: $123.23
Will Tell: $157.02
--> Tom Jackson: $-12.33
```

Package

- Packages are a good mechanism for compartmentalizing diverse classes from each other
- So all of the built-in Java classes are stored in packages.
- There are no core Java classes in the unnamed default package; all of the standard classes are stored in some named package.

Package

- Since classes within packages must be fully qualified with their package name or names, it could become tedious to type in the long dot-separated package pathname for every class you want to use.
- For this reason, 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.

Package

- The import statement is a convenience to the programmer and is not technically needed to write a complete Java program.
- If you are going to refer to a few dozen classes in your application, 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.

Package

- 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 b
- There is no practical limit on the depth of a package hierarchy, except that imposed by the file system.
- Finally, you specify either an explicit class name or a star(*), which indicates that the Java compiler should import the entire package.

Package

- The star form may increase compilation time—especially if you import several large packages.
- For this reason it is a good idea to explicitly name the classes that you want to use rather than importing whole packages.
- However, the star form has absolutely no effect on the run-time performance or size of your classes.

Package

- All of the standard Java classes included with Java are stored in a package called java.
- The basic language functions are stored in a package inside of the java package called java.lang.
- Normally, you have to import every package or class that you want to use, but since Java is useless without much of the functionality in java.lang, it is implicitly imported by the compiler for all programs.

Package

- If a class with the same name exists in two different packages that you import using the star form, the compiler will remain silent, unless you try to use one of the classes.
- In that case, you will get a compile-time error and have to explicitly name the class specifying its package.

Package

- Packages add another dimension to access control.
- Java provides many levels of protection to allow fine-grained control over the visibility of variables and methods within classes, subclasses, and 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.

1. java.lang-contain classes for primitive types, strings, math functions, threads and exceptions
2. java.util-contains classes such as vectors, hash tables, date etc
3. java.io-stream classes for i/o
4. java.awt-classes for implementing GUI-windows, buttons, menus etc
5. java.net-classes for networking
6. java.applet-classes for creating and implementing applets

Package

- Java addresses four categories of visibility for class members:
  - Subclasses in the same package
  - Non-subclasses in the same package
  - Subclasses in different packages
  - Classes that are neither in the same package nor subclasses
- The three access specifiers, ***private, public***, and ***protected***, provide a variety of ways to produce the many levels of access required by these categories.

Package

	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

Package

- Anything declared ***public*** can be accessed from anywhere.
- 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***.

```
package MyPack2;
public class Balance2 {
String name;
double bal;
public Balance2(String n, double b) {
name = n;
bal = b;
}
public void show() {
if(bal<0)
System.out.print("--> ");
System.out.println(name + ": $" + bal);
}
}
```

```
import MyPack2.Balance2;
class TestBalance{
public static void main(String args[]) {
Balance2 test = new Balance2("xx",1.98);
test.show();
}
}
```

- **When a package is imported, only those items within the package declared as public will be available to non-subclasses in the importing code**.

```
package MyPack2;
public class Balance2 {
String name;
double bal;
public Balance2(String n, double b) {
name = n;
bal = b;
}
public void show() {
if(bal<0)
System.out.print("--> ");
System.out.println(name + ": $" + bal);
}
}
```

```
import MyPack2.Balance2;
class TestBalance{
public static void main(String args[]) {
Balance2 test = new Balance2("xx",1.98);
test.show();
}
}
```

OUTPUT

```
D:\java\bin>java TestBalance
xx: $1.98
```

Advantages of using packages

- Java package is used to categorize the classes and interfaces so that they can be easily maintained.
- Java package provides access protection.
- Java package removes naming collision.
- Java Package provides reusability of code.

Interface

- An **interface** is similar to a class that contains both variablesa and methods with a major difference
- Difference is that interface define only abstract methods and final fields.
- The interface in Java is *another 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?

- It is used to achieve abstraction.
- Java does not support "multiple inheritance" (a class can only inherit from one superclass). However, it can be achieved with interfaces, because the class can **implement** multiple interfaces.

Declaring 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.

access **interface** <interface_name>{

```
    // declare constant fields
    // declare methods that abstract
    // by default.
```

}

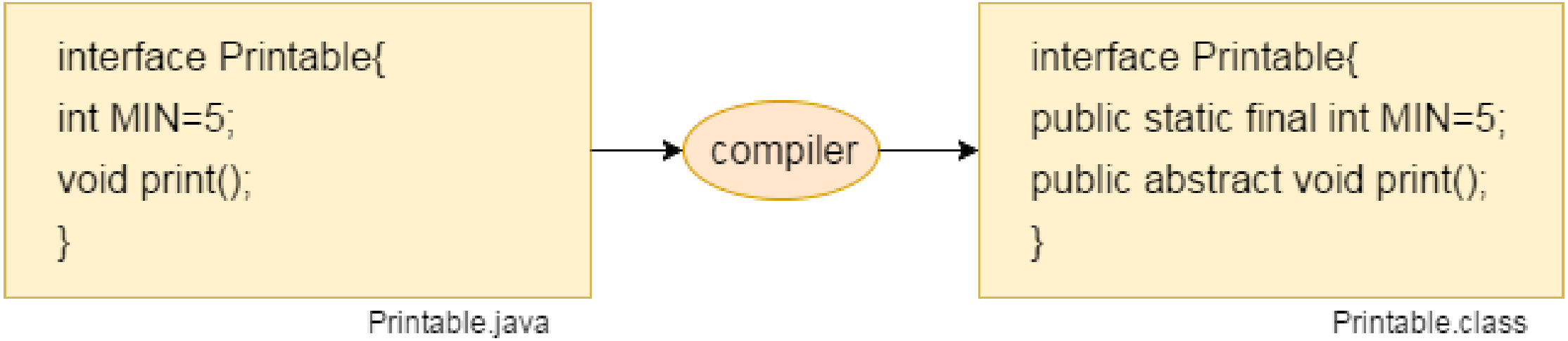
Interface

- An interface is defined much like a class.

```
access interface name {
return-type method-name1(parameter-list);
return-type method-name2(parameter-list);
type final-varname1 = value;
type final-varname2 = value;
// ...
return-type method-nameN(parameter-list);
type final-varnameN = value;
}
```

- Access indicate access specifier. It is either public or not used
- All variables declared inside of the interface are implicitly final and static, meaning they cannot be changed by the implementing class. They must also be initialized
- All methods and variables are implicitly public

- Interface fields are public, static and final by default, and the methods are public and abstract.



```
interface Callback {
        void callback(int param);
}
```

```
class Client implements Callback {
// Implement Callback's interface

        public void callback(int p) {
            System.out.println("callback called with " + p);
        }
 }
```

```
class Client implements Callback {
// Implement Callback's interface
        public void callback(int p) {
                System.out.println("callback called with " +
p);
        }
        void nonIfaceMeth() {
                System.out.println("Classes that implement
interfaces " + "may also define other members, too.");
} }
```

Interface

- Using the keyword interface, you can fully abstract a class' interface from its implementation.
- That is, using interface, you can specify what a class must do, but not how it does it.
- Interfaces are syntactically similar to classes, but they lack instance variables, and their methods are declared without any body.
- In practice, this means that you can define interfaces that don't make assumptions about how they are implemented.

Interface

- Once it is defined, any number of classes can implement an interface.
- Also, one class can implement any number of interfaces.
- To implement an interface, a class must create the complete set of methods defined by the interface.
- However, each class is free to determine the details of its own implementation.
- By providing the interface keyword, Java allows you to fully utilize the “one interface, multiple methods” aspect of polymorphism.

Interface

- 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.
- In this case, the interface must be the only public interface declared in the file, and the file must have the same name as the interface.

Interface

- Variables can be declared inside of interface declarations.
- They are implicitly *final* and *static*, meaning they cannot be changed by the implementing class.
- They must also be initialized.
- Once an interface has been defined, one or more classes can implement that interface.
- To implement an interface, include the *implements* clause in a class definition, and then create the methods defined by the interface.

Multiple Interfaces

- If a class implements more than one interface, the interfaces are separated with a comma.
- The methods that implement an interface must be declared *public*.
- Also, the type signature of the implementing method must match exactly the type signature specified in the interface definition.

```java
interface FirstInterface {
  public void myMethod(); // interface method
}

interface SecondInterface {
  public void myOtherMethod(); // interface method
}

class DemoClass implements FirstInterface, SecondInterface {
  public void myMethod() {
    System.out.println("Some text..");
  }
  public void myOtherMethod() {
    System.out.println("Some other text...");
  }
}

class Main {
  public static void main(String[] args) {
    DemoClass myObj = new DemoClass();
    myObj.myMethod();
    myObj.myOtherMethod();
  }
```

Interface

- In inheritance 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.

```
public interface Limits{
    int low=64;
    int high=94;
}
class Numbers implements Limits{
    int I;
    void series(){
        for(i=low;i<=high;i++){
            if(i%5==0)
             System.out.println("\t"+i);
        }
    }
}
```

```
public Class DemoVariable{
    public static void main(String args[]){
        Numbers obj=new Numbers();
        obj.series();
    }
}
```

Accessing implementations through interface references

- You can declare variables as object references that use an interface rather than a class type.
- Any instance of any class that implements the declared interface can be referred to by such a variable.
- When you call a method through one of these references, the correct version will be called based on the actual instance of the interface being referred to.
- The method to be executed is looked up dynamically at run time, i.e. allowing classes to be created later than the code which calls methods on them.

```
class TestIface {
public static void main(String args[]) {
        Callback c = new Client();
        c.callback(42);
}
}
```

Partial implemetations

If a class includes an interface but does not fully implement the methods defined by that interface, then that class must be declared as abstract.

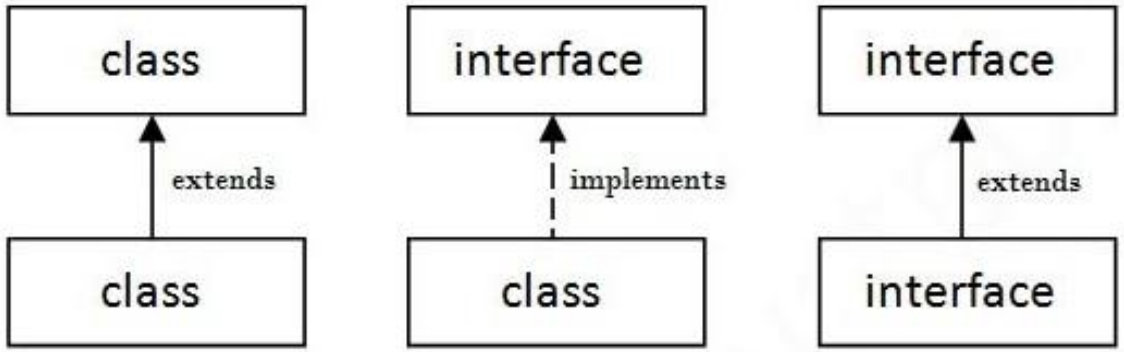
Example:

```
interface Callback {
        void callback(int param);
}

abstract class Incomplete implements Callback {
        int a, b;
        void show() {
                System.out.println(a + " " + b);
        }
        // ...
}
```

Relationship between class and interface

- a class extends another class, an interface extends another interface, but a **class implements an interface**.



Extending Interfaces

- Like classes, it is possible to extend interfaces
- We can create subinterfaces from superintefaces
- The new interface will inherit all the members of the superinterface

```
interface name2 extends name1{
    //Body of name2
}
```

Notes on Interface

- Like **abstract classes**, interfaces **cannot** be used to create objects
- Interface methods do not have a body - the body is provided by the "implement" class
- On implementation of an interface, you must override all of its methods
- Interface methods are by default abstract and public
- Interface attributes are by default public, static and final
- An interface cannot contain a constructor (as it cannot be used to create objects)

Nested Interface

- An interface can be declared a member of a class or another interface. Such an interface is called a member interface or a nested interface
- A nested interface can be declared as public, private or protected

```
class A{
  public interface NestedIF{
        Boolean isNotNegative(intx);
 }
}
class B implements A.NestedIF{
        public Boolean isNotNegative(int x){
                return x<0?false:true;
        }
}
```

Abstract Class	Interface
They may contain abstract as well as non-abstract methods.	They contain only abstract methods.
Variables may be final or not. Can have any access modifier.	Variables are by default public, static, and final.
An abstract class can be extended using “extends” keyword.	They can be implemented using “implements” keyword.
An abstract class can extend another class as well as implement an interface	An interface can only extend another interface.

super

- 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.

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.

```
class Animal{
        String color="white";
}
class Dog extends Animal{
        String color="black";
        void printColor(){
            System.out.println(color);//prints color of Dog class
            System.out.println(super.color);//prints color of Animal class
        }
}
class TestSuper1{
        public static void main(String args[]){
          Dog d=new Dog();
          d.printColor();
        }
}
```

Output

Black

White

In the above example, Animal and Dog both classes have a common property color. If we print color property, it will print the color of current class by default. To access the parent property, we need to use super keyword.

used to invoke parent class method

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

```
class Animal{
      void eat(){System.out.println("eating...");}
}
class Dog extends Animal{
          void eat(){System.out.println("eating bread...");}
          void bark(){System.out.println("barking...");}
          void work(){
                super.eat();
                bark();
          }
}
class TestSuper2{
      public static void main(String args[]){
          Dog d=new Dog();
          d.work();
       }
}
```

Output

Eating…
Barking…

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

used to invoke parent class constructor.

```
class Animal{
        Animal(){System.out.println("animal is created");}
}
class Dog extends Animal{
        Dog(){
            super();
            System.out.println("dog is created");
        }
}
class TestSuper3{
        public static void main(String args[]){
            Dog d=new Dog();
        }
}
```

Output

Animal is created

Dog is created

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

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