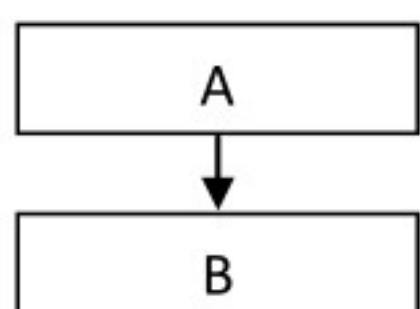


Inheritance

- **Inheritance** is the process, by which class can acquire the properties and methods of its parent class.
- The mechanism of deriving a **new child class** from **an old parent class** is called **inheritance**.
- The new class is called **derived** class and old class is called **base** class.
- When you inherit from an existing class, you can **reuse** methods and **fields** of **parent** class, and you can add new methods and fields also.
- All the properties of **superclass** except private properties can be inherit in its **subclass** using **extends** keyword.

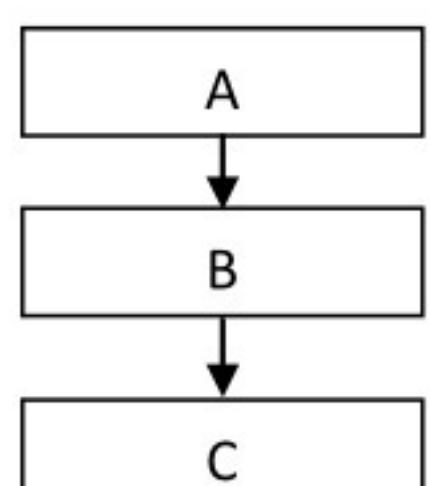
Types of Inheritance

Single Inheritance



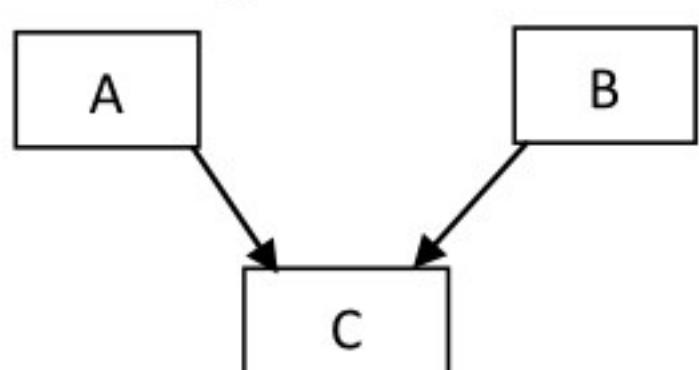
- If a class is derived from a single class then it is called **single inheritance**.
- Class **B** is derived from class **A**.

Multilevel Inheritance



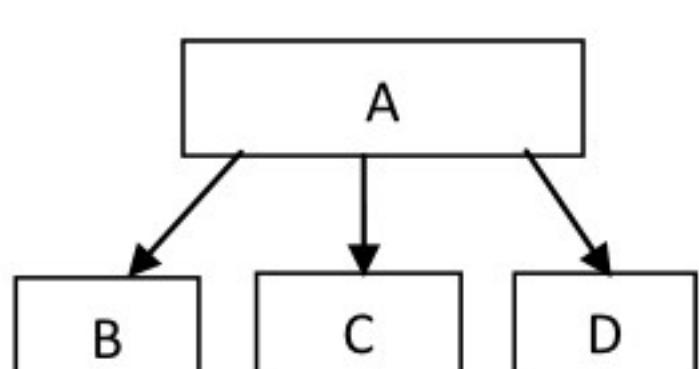
- A class is derived from a class which is derived from another class then it is called **multilevel inheritance**
- Here, class **C** is derived from class **B** and class **B** is derived from class **A**, so it is called multilevel inheritance.

Multiple Inheritance



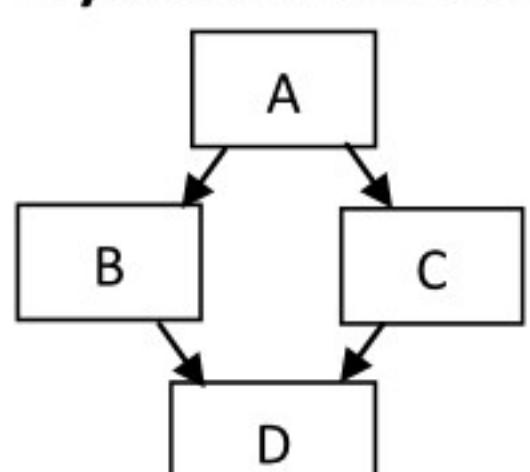
- If one class is derived from more than one class then it is called **multiple inheritance**.
- It is **not supported** in java through class.

Hierarchical Inheritance



- If one or more classes are derived from one class then it is called **hierarchical inheritance**.
- Here, class **B**, class **C** and class **D** are derived from class **A**.

Hybrid Inheritance



- Hybrid inheritance is combination of **single** and **multiple** inheritance.
- But java doesn't support multiple inheritance, so the hybrid inheritance is also **not possible**.

Example

```
class A
{
    public void displayA()
    {
        System.out.println("class A method");
    }
}

class B extends A          //Single Inheritance - class B is derived from class A
{
    public void displayB()
    {
        System.out.println("class B method");
    }
}

class C extends B // Multilevel Inheritance - class C is derived from class B
{
    public void displayC()
    {
        System.out.println("class C method");
    }
}

class D extends A //Hierarchical Inheritance - Class B and Class D are derived from Class A
{
    public void displayD()
    {
        System.out.println("class D method");
    }
}

class Trial
{
    public static void main(String []args)
    {
        B b=new B();
        C c=new C();
        D d=new D();
        b.displayB();
        c.displayC();
        d.displayD();
    }
}
```

- Class **B** and class **D** are derived from class **A** so it is example of **Hierachal Inheritance**.

Method Overriding

- When a method in a **subclass** has the **same name and type signature** as method in its **superclass**, then the method in the **subclass** is said to **override the method** of the super class.
- **The benefit of overriding is:** Ability to define a behavior that's specific to the **subclass** type. Which means a subclass can implement a superclass method based on its requirement.
- Method overriding is used for **runtime polymorphism**.
- In object oriented terms, overriding means to **override the functionality** of any existing method.

Example:

```

class A
{
    public void display()
    {
        System.out.println("Class A");
    }
}
class B extends A
{
    public void display()
    {
        System.out.println("Class B");
    }
}
class Trial
{
    public static void main(String args[])
    {
        A a = new A();          // A reference and object
        A b = new B();          // A reference but B object
        a.display();            // Runs the method in A class
        b.display();            // Runs the method in B class
    }
}

```

Output:

Class A

Class B

- In the above example, you can see that even though **b** is a type of **A** it runs the **display()** method of the **B** class.
- Because, at **compile** time reference type of object is checked. However, at the **runtime** JVM determine the object type and execute the method that belongs to that particular **object**.

Rules for method overriding

- The **argument list** should be exactly the **same** as that of the overridden method.

- The **return type** should be the **same** as the return type declared in the original overridden method in the superclass.
- **Instance methods** can be overridden only if they are **inherited** by the **subclass**.
- A method declared **final** cannot be overridden.
- A method declared **static** cannot be overridden but can be re-declared.
- If a method cannot be **inherited** then it cannot be **overridden**.
- A subclass within the same package as the instance's superclass can override any superclass method that is not declared private or final.
- A subclass in a different package can only override the non-final methods declared public or protected.
- **Constructors** cannot be overridden.

super keyword

- The **super** keyword in java is a reference variable that is used to refer **immediate parent** class object.

super To Access super-class Members

- If your method overrides one of its super-class methods, you can invoke the **overridden** method through the use of the keyword **super**.

Example:

```

class A
{
    String name = "Class A";
    public void display()
    {
        System.out.println("Class A display method called..");
    }
}
class B extends A
{
    String name = "Class B";
    public void display()
    {
        System.out.println("Class B display method called..");
    }
    void printName()
    {
        //this will print value of name from subclass(B)
        System.out.println("Name from subclass : " + name);

        // this will print value of name from superclass(A)
        System.out.println("Name from Superclass: " + super.name);
    }
}
```

```

//invoke display() method of Class B method
display();

// invoke display() method of class A(superclass) using super
super.display();
}

}

class SuperDemo
{
    public static void main(String args[])
    {
        B b1 = new B();
        b1.printName();
    }
}

```

Output:

Name from subclass : Class B
Name from Superclass: Class A
Class B display method called..
Class A display method called..

- Here, **display()** method of **subclass** overrides the **display()** method of its **super-class**.
- So, to call **superclass(A)** members within **subclass(B)** **super** keyword is used.

super To Call super-class Constructor

- Every time a parameterized or non-parameterized constructor of a **subclass** is created, by default a default constructor of **superclass** is called **implicitly**.
- The syntax for calling a **superclass** constructor is:

super();

OR

super(parameter list);

- The following example shows how to use the **super** keyword to invoke a superclass's constructor.

Example:

```

class A
{
    A()
    {
        System.out.println("Super class default constructor called..");
    }
    A(String s1)
    {
        System.out.println("Super class parameterized constructor called: "+s1);
    }
}
class B extends A
{
}

```

```

/* Implicitly default constructor of superclass(A) will be called. Whether you define
super or not in subclass(B) constructor */

B()
{
    System.out.println("Sub class default constructor called..");
}

/* To call a parameterized constructor of superclass(A) you must write super() with
same number of arguments*/
B(String s1)
{
    super("Class A");
    System.out.println("Sub class parameterized constructor called: " + s1);
}

class SuperConDemo
{
    public static void main(String args[])
    {
        B b1 = new B();
        B b2 = new B("Class B");
    }
}

```

Output:

```

Super class default constructor called..
Sub class default constructor called..
Super class parameterized constructor called: Class A
Sub class parameterized constructor called: Class B

```

- Here, implicitly a default constructor of **superclass** is called not a parameterized one. To call a **parameterized** constructor of a **superclass** we must use '**super(parameters..)**' with matching parameters.

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.
- It is also known as **run-time polymorphism**.
- A **superclass** reference variable can refer to a **subclass object**. It is known as **Upcasting**.
- When an overridden method is called through a **superclass reference**, the determination of the method to be called is based on **the object being referred to** by the reference variable.
- This determination is made at **run time**.

Example:

```

class A
{
    void callme()
    {

```

```

        System.out.println("Inside A's callme method");
    }
}
class B extends A
{
    void callme()
    {
        System.out.println("Inside B's callme method");
    }
}
class C extends A
{
    void callme()
    {
        System.out.println("Inside C's callme method");
    }
}
class DispatchDemo
{
    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 r 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:

Inside A's callme method
Inside B's callme method
Inside C's callme method

Object Class

- The **Object class** is the **parent** class (`java.lang.Object`) of all the classes **in java by default**.
- The Object class provides some common behaviors to all the objects must have, such as object can be compared, object can be converted to a string, object can be notified etc..
- Some Java object class methods are given below:**

Method	Description
<code>equals()</code>	To compare two objects for equality.

getClass()	Returns a runtime representation of the class of this object. By using this class object we can get information about class such as its name, its superclass etc.
toString()	Returns a string representation of the object.
notify()	Wakes up single thread , waiting on this object's monitor.
notifyAll()	Wakes up all the threads , waiting on this object's monitor.
wait()	Causes the current thread to wait , until another thread notifies.

Example:

```

class parent
{
    int i = 10;
    Integer i1 = new Integer(i);
    void PrintClassName(Object obj)      // Pass object of class as an argument
    {
        System.out.println("The Object's class name is ::" + obj.getClass().getName());
    }
}
class ObjectClassDemo
{
    public static void main(String args[])
    {
        parent a1 = new parent();
        a1.PrintClassName(a1);
        System.out.println("String representation of object i1 is:: "+a1.i1.toString());
    }
}

```

Output:

```

The Object's class name is :: parent
String representation of object i1 is:: 10

```

Packages

- A **java package** is a group of similar types of **classes, interfaces and sub-packages**.
- Packages are used to **prevent naming conflicts** and provides **access protection**.
- It is also used to **categorize the classes and interfaces**. so that, they can be easily **maintained**.
- We can also **categorize the package** further by using concept of **subpackage**. Package inside the package is called the **subpackage**.
- Package can be categorized in two form: **built-in package** and **user-defined package**.
 - **built-in packages** :Existing Java package such as **java.lang, java.util, java.io, java.net, java.awt**.
 - **User-defined-package** : Java package created by user to categorized **classes and interface**.
- Programmers can define their own packages to bundle group of **classes, interfaces** etc.

Creating a package

- To create a package, **package** statement followed by **the name of the package**.

- The **package** statement should be the first line in the source file. There can be only **one package statement** in each source file.
- If a package statement is **not used** then the class, interfaces etc. will be put into an unnamed package.

Example:

```

package mypack;
class Book
{
    String bookname;
    String author;
    Book()
    {
        bookname = "Complete Reference";
        author = "Herbert";
    }
    void show()
    {
        System.out.println("Book name is :: "+bookname+"\nand author name is :: "+
author);
    }
}
class DemoPackage
{
    public static void main(String[] args)
    {
        Book b1 = new Book();
        b1.show();
    }
}
  
```

Compile: javac -d . DemoPackage.java

To run the program : java mypack.DemoPackage

Output:

Book name is :: Complete Reference

and author name is :: Herbert

- The **-d** is a switch that tells the compiler where to put the class file. **Like**, /home (Linux), d:/abc (windows) etc.
- The **.** (dot) represents the **current folder**.

Import Package

- import** keyword is used to import **built-in and user-defined** packages into your java source file.
- If a **class** wants to use **another class** in the **same package**, no need to import the package.
- But, if a **class** wants to use **another class** that is **not exist in same package** then **import** keyword is used to import that package into your java source file.
- A class file can contain **any number of import** statements.
- There are three ways to access the package from outside the package:**

- 1) import package.*;
 - 2) import package.classname;
 - 3) fully qualified name
- 1) Using packagename.* :**
- If you use **packagename.*** then all the **classes and interfaces** of this package will be accessible but **not subpackages**.
- Syntax:** import packagename.*;
- 2) Using packagename.classname :**
- If you use **packagename.classname** then only declared **class** of this package will be accessible.
- Syntax:** import packagename.classname;
- 3) Using fully qualified name :**
- If you use **fully qualified name** then only **declared class** of this package will be accessible. So, no need to import the package.
 - 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:

File 1: **A.java**

```
package pack;
public class A
{
    public void display()
    {
        System.out.println("Welcome to package pack...");
    }
}
```

File 2: **Q.java**

```
package pack;
public class Q
{
    public void Q_display()
    {
        System.out.println("Welcome to package pack through qualified
name...");
    }
}
```

File 3: **B.java**

```
package mypack;
import pack.*;           // Here, you can also use import pack.A ;
class B
{
    public static void main(String args[])
    {
```

```

A a1 = new A();
pack.Q q1= new pack.Q();
/*Here Q is class in package A. If you want to access Q_display() method of class Q
using fully qualified name then no need to import package named pack */
    a1.display();
    q1.Q_display();
}
}

```

Output: java mypack.B

Welcome to package pack...

Welcome to package pack through qualified name...

- In above example, **class A, class Q and all its methods** must be declared as **public** otherwise they can be accessed in **class B**.

Visibility and Access Rights

class \ have access to	Private	Default	Protected	Public
own class	yes	yes	yes	yes
subclass - same package	no	yes	yes	yes
subclass - another package	no	no	yes	yes
class - another package	no	no	no	yes

Interfaces

- An **interface** is a collection of **abstract methods**. An interface is not a **class**.
- When you **create** an interface it defines what a class can do without saying anything about how the class will do it.
- Interface contains only **static constants** and **abstract methods** only.
- The interface in java is a **mechanism to achieve fully abstraction**. There can be only abstract methods in the java interface not method body.
- By default (Implicitly), an interface is **abstract**, interface **fields**(data members) are **public, static and final** and **methods** are **public and abstract**.
- It is **used** to achieve **fully abstraction** and **multiple inheritance** in Java.
- **Similarity between class and interface are given below:**
 - An interface can contain any **number of methods**.
 - An interface is written in a file with a **.java** extension, with the name of the interface matching the name of the file.
 - The **bytecode** of an interface appears in a **.class** file.
- **Difference between class and interface are given below:**

Class	Interface
<ul style="list-style-type: none"> • You can instantiate class. 	<ul style="list-style-type: none"> • You cannot instantiate an interface.
<ul style="list-style-type: none"> • It contains default as well as parameterize constructors. 	<ul style="list-style-type: none"> • It does not contain any constructors.

<ul style="list-style-type: none"> All the methods should have definition otherwise declare method as abstract explicitly. 	<ul style="list-style-type: none"> All the methods in an interface are abstract by default.
<ul style="list-style-type: none"> All the variables are instance by default. 	<ul style="list-style-type: none"> All the variables are static final by default, and a value needs to be assigned at the time of definition.
<ul style="list-style-type: none"> A class can inherit only one Class and can implement many interfaces. 	<ul style="list-style-type: none"> An interface cannot inherit any class while it can extend many interfaces.

Declaring Interfaces

- The **interface** keyword is used to **declare** an interface.

Syntax: NameOfInterface.java

```
import java.lang.*;
public interface NameOfInterface
{
    //Any number of final, static fields
    //Any number of abstract method declarations
}
```

Example: DemolInterface.java

```
interface DemolInterface
{
    int i = 10;
    void demo();
}
```

- In above example, name of interface is **DemolInterface** and it contains a variable **i** of integer type and an abstract method named **demo()**.

Implementing Interfaces

- A **class** uses the **implements** keyword to implement an **interface**.
- A **class** implements **an interface** means, 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**.

Example: DemolInterfacelmp.java

```
public class DemolInterfacelmp implements DemolInterface
{
    public void demo()
    {
        System.out.println("Value of i is :: "+i);
    }
    public static void main(String args[])
    {
        DemolInterfacelmp d = new DemolInterfacelmp();
        d.demo();
    }
}
```

```

    }
}
```

Output:

Value of i is :: 10

Inheritance on Interfaces

- We all knows a class can extend another class. Same way an **interface can extend another interface**.
- The **extends** keyword is used to extend an interface, and the **child interface** inherits the methods of the **parent interface**.

Example:

```

public interface A
{
    void getdata(String name);
}

public interface B extends A
{
    void setdata();
}

class InheritInterface implements B
{
    String display;
    public void getdata(String name)
    {
        display = name;
    }
    public void setdata()
    {
        System.out.println(display);
    }
    public static void main(String args[])
    {
        InheritInterface obj = new InheritInterface();
        obj.getdata("Welcome TO Heaven");
        obj.setdata();
    }
}
```

Output:

Welcome TO Heaven

- The interface **B** has one method, but it **inherits** one from interface **A**; thus, a class **InheritInterface** that implements **B** needs to implement **two methods**.

Multiple Inheritance using Interface

- If a **class** implements **multiple interfaces**, or an **interface** extends **multiple interfaces** known as **multiple inheritance**.

- A java **class** can only extend **one parent class**. Multiple inheritances are not allowed. However, an **interface** can extend **more than one parent interface**.
- The **extends** keyword is used once, and the **parent interfaces** are declared in a **comma-separated list**.

Example:

```

public interface A
{
    void getdata(String name);
}
public interface B /* Here we can also extends multiple interface like interface B
extends A,C*/
{
    void setdata();
}
class InheritInterface implements A, B
{
    String display;
    public void getdata(String name)
    {
        display = name;
    }
    public void setdata()
    {
        System.out.println(display);
    }
    public static void main(String args[])
    {
        InheritInterface obj = new InheritInterface();
        obj.getdata("Welcome TO Heaven");
        obj.setdata();
    }
}

```

Output:

Welcome TO Heaven

Abstract Class

- A **class** that is declared with **abstract** keyword, is known as an **abstract class** in java. It can have **abstract** and **non-abstract methods** (method with body).
- It needs to be **extended** and its method implemented. It cannot be instantiated means we can't create object of it.
- Any class that extends an **abstract** class must implement all the **abstract methods** declared by the super class.

Syntax:

```

abstract class class_name
{

```

//Number of abstract as well as non-abstract methods.

}

- A **method** that is declared as **abstract** and does not have implementation is known as **abstract method**.
- The method body will be defined by its **subclass**. Abstract method can never be **final** and **static**.

Syntax:

```
abstract return_typefunction_name(); // No definition
```

Example:

```
abstract class A
{
    abstract void abs_method();
    public void display()
    {
        System.out.println("This is concrete method..");
    }
}
class DemoAbstract extends A
{
    void abs_method()
    {
        System.out.println("This is an abstract method..");
    }
    public static void main(String[] args)
    {
        DemoAbstract abs = new DemoAbstract();
        abs.abs_method();
        abs.display();
    }
}
```

Output:

This is an abstract method..

This is concrete method..

Final Keyword

- The **final keyword** in java is used to restrict the user. The java final keyword can be used with:
 - 1) variable
 - 2) method
 - 3) class
- **Final Variable:** If you make any variable as **final**, you cannot change the value of that final variable (It will be constant).
 - A variable that is declared as **final** and **not initialized** is called a **blank final** variable. A blank final variable forces the **constructors to initialize it**.
- **Final Method:** Methods declared as **final** cannot be **overridden**.
- **Final Class:** Java classes declared as **final** cannot be extended means cannot **inherit**.
- If you declare any parameter as **final**, you cannot change the value of it.

Example:

```

class DemoBase
{
    final int i = 1;           //final variable must be initialize.
    final void display()
    {
        System.out.println("Value of i is :: "+ i);
    }
}
class DemoFinal extends DemoBase
{
    /*void display()      // Compilation error final method cannot override.
    {
        System.out.println("Value of i is :: "+ i);
    }*/
    public static void main(String args[])
    {
        DemoFinal obj = new DemoFinal();
        obj.display();
    }
}

```

Output:

Value of i is :: 1

Difference between Method Overloading and overriding

Overriding	Overloading
• The argument list must exactly match that of the overridden method.	• Overloaded methods MUST change the argument list .
• The return type must be the same as overridden method in the super class.	• Overloaded methods CAN change the return type .
• Private and final methods cannot be overridden.	• Private and final methods can be overloaded.
• Method overriding occurs in two classes that have inheritance.	• Method overloading is performed within class.
• Dynamic binding is being used for overridden methods.	• Static binding is being used for overloaded methods.
• Method overriding is the example of run time polymorphism .	• Method overloading is the example of compile time polymorphism .

Difference between Interface and Abstract class

Interface	Abstract class
• Interface can have only abstract methods.	• Abstract class can have abstract and non-abstract methods.
• Interface supports multiple inheritance .	• Abstract class doesn't support multiple inheritance .

<ul style="list-style-type: none">• Interface can't have static methods, main method or constructor.	<ul style="list-style-type: none">• Abstract class can have static methods, main method and constructor.
<ul style="list-style-type: none">• In interface all method should be define in a class in which we implement them.	<ul style="list-style-type: none">• This is not applicable for abstract classes.
<ul style="list-style-type: none">• Interface can't provide the implementation of abstract class.	<ul style="list-style-type: none">• Abstract class can provide the implementation of interface.