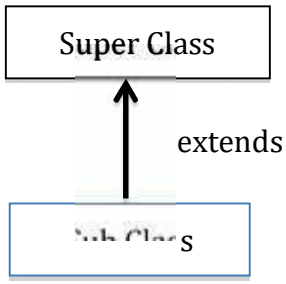
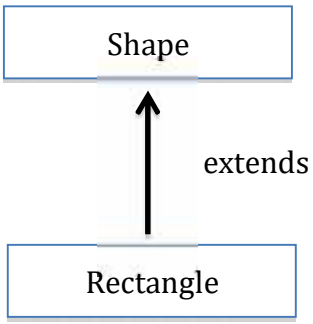
Chapter 2
Inheritance Basics, Member Access and Inheritance, Constructors and Inheritance, Using super to Call Superclass constructors, Using super to Access Superclass Members, Creating a Multilevel Hierarchy, When are Constructors Executed, Superclass References and Subclass Objects, Method Overriding, Overridden Methods support polymorphism, Why Overridden Methods, Using Abstract Classes, Using final, The Object Class.

## Inheritance Basics

- **What is inheritance?**
  - Inheritance is a mechanism in which one object acquires all the properties and behaviors of parent object.




(a)




(b)

  - Inheritance represents the IS-A relationship, Example Rectangle is a shape

- **Why use Inheritance**
  - To achieve Runtime Polymorphism using method overriding
  - For code reusability.

- **Methods of Inheritance**
  - By extending a class
  - By implementing an interface

**i) Inheritance By extending a class:**

**Syntax**

```
classSubclassName extends SuperclassName
{
    //methods and fields
}
```

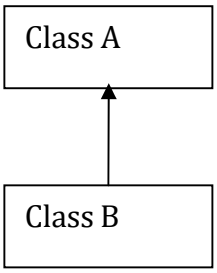
**ii) Inheritance By implementing a interface:**

Syntax:

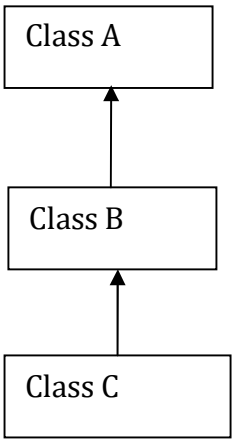
```
Class SubclassName implements interfaceName
{
    //methods and fields
```

*}*

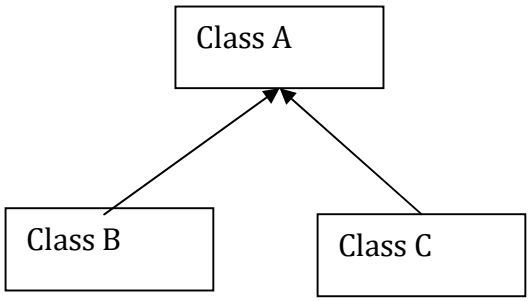
- **Types of Inheritance**
  - On the basis of class, there can be three types of inheritance: single, multilevel and hierarchical, multiple and hybrid is supported through interface only. We will learn about interface later.




**1.Simple Inheritance**




**2. Multilevel Inheritance**




**3. Hierarchical Inheritance**

- **Simple Inheritance**

```
Class ClassA {
     //methods and fields of ClassA class
}
class ClassB extends ClassA {
     //methods and fields of ClassB class
}
```

- **Multilevel Inheritance**

```
Class ClassA {
     //methods and fields of ClassA class
}
class ClassB extends ClassA {
     //methods and fields of ClassB class
}
Class classC extends ClassB{
//methods and fields of ClassC class
}
```

- **Hierarchical Inheritance**

```
Class ClassA {
     //methods and fields of ClassA class
}
class ClassB extends ClassA {
     //methods and fields of ClassB class
}
Class classC extends ClassA{
//methods and fields of ClassC class
}
```

**Member Access and Inheritance**

- 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.
- In a class hierarchy, private members remain private to their class. It is not accessible by any code outside its class, including subclasses.
- Example

```
// Create a superclass.
class A {
         int i; // public by default
         private int j; // private to A
         voidsetData (int x, int y) {
              i = x; j = y;
         }
 }
class B extends A {   // A's j is not accessible here.
     int total;
     void sum() {
          total = i + j;       // ERROR, j is not accessible here
     }
}
class Access {

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

- Java has a special access modifier known as ***protected*** which is meant to support Inheritance in Java. Any protected member including protected method and field are accessible for classes of same package and only Sub class outside the package.

## Constructors and Inheritance

- When both the superclass and subclass define constructor, the process is a bit more complicated because both the superclass and subclass constructor must be executed .
- In this case java's keyword '***super***' can be used.
    - First calls the superclass constructor
    - Second is used to access a member of the superclass that has been hidden by a member of a subclass.

## Using super to call superclass constructor

- A subclass can call a constructor defined by its superclass by use of the following form of super
- Syntax
    - Super(*parameter_list*);
        - *parameter_list* specifies any parameter needed by the constructor in superclass().
        - super() must be the first statement executed inside a subclass constructor.

- **Example**

```
class SuperClass {
        SuperClass( ) {
                System.out.println("SuperClass is created");
        }
}
class Subclass extends SuperClass {
        SubClass( ) {
        super( );
                System.out.println("SubClass is created");
        }
}
class Demo {
        public static void main(String args[]){
        Demo d=new Demo();
}
}
```

### Output :

```
SuperClass is created
SubClass is created
```

## Example 2:

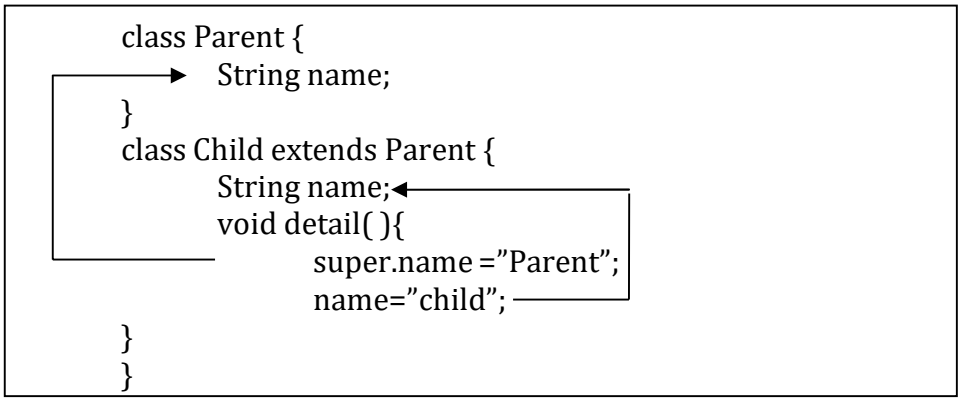
```
class Person{
      int id;
      String name;
      Person(intid,String name){
            this.id=id;
            this.name=name;
      }
}
class Emp extends Person{
      float salary;
      Emp(intid,Stringname,float salary){
            super(id,name); //reusing parent constructor
            this.salary=salary;
            }
      void display( ){
            System.out.println(id+" "+name+" "+salary);}
      }

class TestSuper {
      public static void main(String[] args){
      Emp e1=new Emp(1,"ankit",45000f);
      e1.display();
      }
}

Output:
1 ankit 45000
```

**Using super to access super class members**

- In Java 'super' keyword can be used to refer super class instance in a subclass.
- 'super' when used to access members need not be first line.
- **Syntax :**

  **Super.memberVariable**
  **Super.memberMethod( )**

- **Example 1:**
  accessing Shadowing instance variable
  In java super keyword is used to refer to immediate parent class of a class. In other words super keyword is used by subclass whenever it need to refer to its immediate super class.



- The 'name' instance variable in class child shadows the instance variable of parent class as both are having same name.

- If we display instance variable 'name' in sub class, it displays data of subclass. To access super class data from shadowed variable, we use '***super.name***'.

- **Example 2:**
  - **Accessing overridden instance method**

```
class Bank { String name;
      void showBank ( )  {
            System.out.println("Customer Name = " + name);
      }
}
class Customer extends Bank {
String name;
      void setName(String custName, String bankName) {
            super.name = bankName; //sets instance variable of super
            name = custName;
      }
      void show () {
            super.showBank();  //calls method of super class
            System.out.println("Customer Name = " + name);
      }

}
class BankAccount{
Customer cust1 = new Customer();
cust1.setName ( "Anil", "ICICI" );
cust1.show ( );

}
```

## Creating a Multilevel Hierarchy

- You can build hierarchies that contain as many layers of inheritance as you like. As mentioned, it is perfectly acceptable to use a subclass as a superclass of another**.**
- **Order of constructor execution in multilevel inheritance**
  - When a class hierarchy is created, then the *"constructors are called in order of derivation, from super class to subclass"*.
  - If super is used to call constructor, 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 constructor*** of each super class will be executed. Subclass constructor is the last to execute.

**Example:**

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

class ClassC extends ClassB {
      ClassC( ) {
      System.out.println("Inside C's constructor.");
     }
}
class Demo {
      public static void main(String[] args) {
      ClassC obj = new ClassC( );
      }
}
```

**Output:**

```
Inside A's constructor.
Inside B's constructor.
Inside C's constructor.
```

## When are Constructors Executed

- Constructors are executed in order of derivation.
  - Because a superclass has no knowledge of any subclass, any initialization it needs to perform is separate from and possibly prerequisite to any initialization performed by subclass.

```
class ClassB
{
      ClassB( ) {
            System.out.println("Inside B's constructor.");
      }
}
class ClassC extends ClassB {
      ClassC( ) {
            System.out.println("Inside C's constructor.");
     }
}
class Demo {
      public static void main(String[ ] args) {
      ClassC obj = new ClassC( ); //constructs a C object
      }
}
```

**Output:**

```
Inside B's constructor.
Inside C's constructor.
```

## Superclass Reference and Subclass Objects

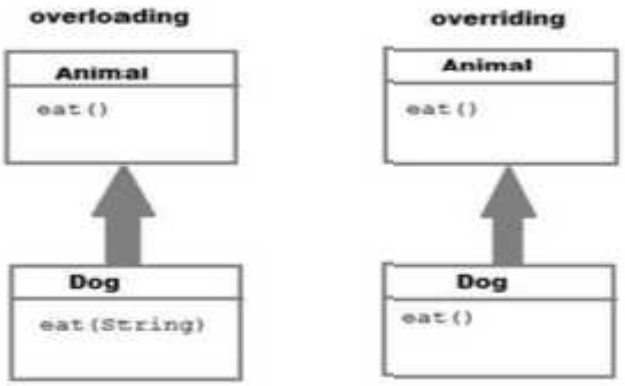
- Upper class reference variable can point to Sub Class Object e.g.
  **SuperClassparentOb = new SubClass();**
- If you want to store object of Sub class, which is stored in super class reference variable as above, back to Sub class reference then you need to use casting, as shown below :
  **SubClasschildOb = (SubClass) parent;** *//since parent variable pointing to SubClass object*
- Accessing subclass overloaded methods using super reference
- **Example**

```
class SuperA {
      void showSuper( ) {
            System.out.println("Super class"); }
}
class SubB extends SuperA {
      void showSub( ) {
            System.out.println("Sub class");
      }
 }
class Demo {
      SuperA SA = new SubB();
      SA.showSub();//ERROR: super class don"t know what subclass has extended
}
```

- Super class reference can't access subclass methods even if they are overloaded.
- Super class reference can access subclass methods only if they are overridden which implements runtime polymorphism
- Private members of Super class are not visible to Sub class even after using Inheritance in Java.
- Java has a special access modifier known as protected which is meant to support Inheritance in Java. Any protected member including protected method and field are accessible for classes of same package and only Sub class outside the package.

## Method Overriding

Difference between overloading and overriding



| Overloading | Overriding |
|---|---|
| Static polymorphism | Runtime polymorphism |
| Same name different signature | Same name same signature |
| Return type can be different | Return type also should be same |
| Can be overloaded in same or in a subclass | Only be overridden in a subclass |

- **Rules for overriding**
  - Instance methods can be overridden only if they are inherited by the subclass.
  - private, static and final methods cannot be overridden in Java..
  - Constructors cannot be overridden.

```
class Animal{
public void move(){
          System.out.println("Animals can move"); } }

class Dog extends Animal{
public void move(){
     System.out.println("Dogs can walk and run"); }

}
public class TestDog{
public static void main(String args[]){
          Animal bullDog = new Dog(); // Animal reference but Dog object
          bullDog.move(); //Runs the method in Dog class

         }
}
```

**Overridden Methods support polymorphism or (Dynamic Method Dispatch)**

- Method overriding is a Means of run time polymorphism. Which method to call is based on object type, at runtime time
- Dynamic method dispatch is the mechanism by which a call to an overridden method is resolved at run time, rather than compile time.
- Method to execute is based upon the type of the object being referred to at the time the call occurs.

```
class Animal {
public void move(){
System.out.println("Animals can move");
  }
}
class Dog extends Animal{
public void move(){
System.out.println("Dogs can walk and run");
   }
}
class Snake extends Animal{
public void move(){
System.out.println("Snake can't walk But run");
}
public class TestAnimal{
public static void main(String args[]){
      Animal animal = new Animal();
Snake cobra = new Snake();
      Dog bullDog = new Dog();
animal.move();   //Runs move in Animal class
animal = bullDog; // animal refers to Dog object
animal.move();   //Runs move in Dog class
animal = cobra; // animal refers to Snake object
animal.move();   //Runs move in Snake class
  }
}
```

## Why Overridden Methods

- It allows a general class to specify methods that will b common to all derivaties, while allowing subclasses to define the specific implementation of some or all of those methods.
- Overridden methods are another way that java implements the "one interface, multiple methods"

## Using Abstract Classes

- An abstract method is a method that is declared without an implementation (without braces, and followed by a semicolon)
- An abstract method has no implementation. It just has a method signature.
- **Syntax:** *abstract return-type function-name( );*
- You declare a method abstract by adding the abstract keyword in front of the method declaration.
- Example: *abstract public double area( );*

- *Abstract Class:*
  - The purpose of an abstract class is to specify the default functionality of an object and let its sub-classes to explicitly implement that functionality.
  - ***Java Abstract classes are*** used ***to declare common characteristics of subclasses.***
  - It can only be used as a superclass for other classes that extend the abstract class.

- **Syntax:** Abstract classes are declared with the abstract keyword

```
public abstract class MyabstractClass
{
        //code
}
```

An abstract class cannot be instantiated.

*MyAbstractClass myClassinstance = new MyAbstractClass( );//not valid*

- Abstract classes are used to provide a template or design for concrete subclasses down the inheritance tree.

**Example:**

```
 abstract class A
{
 abstract void display(); //abstract method
 public void normal()
 {
  System.out.println("this is concrete method");
 }
}
class B extends A
{
 void display()
 {
  System.out.println("this is callme.");
 }
 public static void main(String[] args)
 {
  B b = new B();
  b.callme();
  b.normal();
 }
}
```

**Using final**

a) final class
b) final method
c)final variable

**final class:**

- Restrict the inheritance, A final class cannot be subclassed.
- All methods in a final class are implicity final.
- Syntax:

```
final class className {
        //code
}
```

- **Example**

```
public final class MyFinalClass {
//some code
}
public class WrongSubClass extends MyFinalClass {      // forbidden

.....
}
```

- Final keyword helps to write immutable class.
- Immutable classes are the one which cannot be modified once it gets created
- String is primary example of immutable and final class .

**Final method**

- A final method cannot be overridden by subclasses.
- This is used to prevent unexpected behaviour from a subclass altering a method that may becrucial to the function or consistency of the class
- You should make a method final in java if you think it's complete and its behavior should remain constant in sub-classes
- Final methods are faster than non-final methods because they are not required to be resolved during run-time and they are bonded on compile time.
- Final methods are bonded during compile time also called static binding.
- **Example:**

```
public final class Bank {
        public final double getRateOfInterest()
        {
                return 8.9;
        }
}
```

**Final variable and static final variable**

- Once the value assigned to that variable, it can't be changed.
- Final variable behaves as constant throughout your application.

**Final variable**

- **Non-static final variables can be** assigned a value either in constructor or with the declaration.

```
public class Demo {

        final int iVal;

        Demo()
        {
          iVal=10;  //final variable initialised in constructor
        }

        public static void main(String args[]){

            Demo m = new Demo();
            System.out.println(m.iVal);
        }
}
```

- **Final variables** are best to use in multi threading application as its immutable (It can't be changed after initialized it once), Less error prone code.

  - **Static final variables**
    - Static final variables cannot be assigned value in constructor; they must be assigned a value with their declaration.

```
public class Demo {

        static final int iVal = 10;

        Demo()
        {
          iVal=10;  //ERROR: Can't initialised in constructor
        }

        public static void main(String args[]){

            Demo m = new Demo();
            System.out.println(m.iVal);
        }
}
```

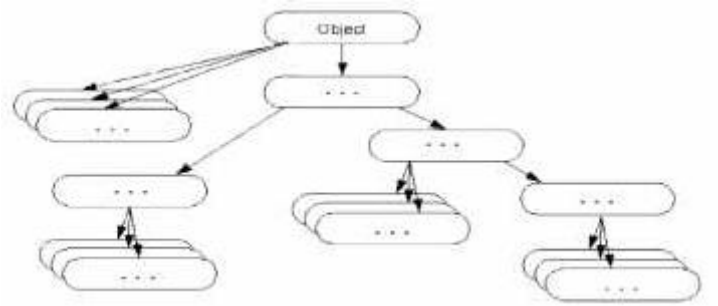
  - **Final Parameters**
    - valuesof the parameters cannot be changed after initialization. Its always best if you make your method arguments as **final which makes your program more readable and less error prone and easy to debug.**

```
class Box{
    int width, height;

    void setVal(final int w, final int h)
    {
        width=w;
        height=h;
    }
}
```

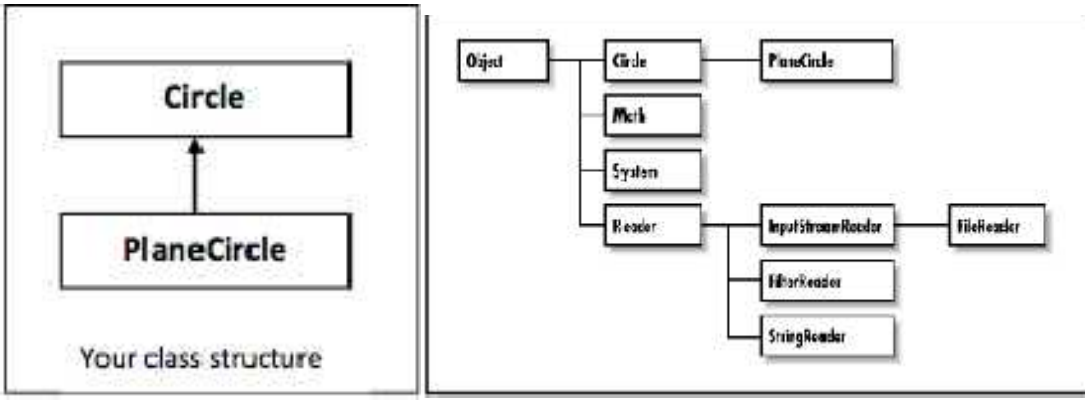
- **Final object reference variable?**
  - **A reference variable marked final can't ever be reassigned to refer to a** different **object.**
  - The data within the object can be modified, but the reference variable cannot be changed.

**The object class**

- The **java.lang.Object**class is the root of the class hierarchy. Every class has **Object** as a **superclass**. Every class you use or write inherits the instance methods of Object.




- Consider you have a super class called ***Circle*** and Subclass called ***PlaneCircle***then it can be viewed as shown below.




**Methods of object class:**

| Method | Description |
|---|---|
| public boolean **equals(Object obj)** | compares the given object to this object. |
| protected Object **clone**() throws CloneNotSupportedException | creates and returns the exact copy (clone) of this object. |
| public String **toString**() | returns the string representation of this object. |
| public final void **notify**() | wakes up single thread, waiting on this object's monitor. |
| public final void **notifyAll**() | wakes up all the threads, waiting on this object's monitor. |
| public final void **wait (long timeout)** throws InterruptedException | causes the current thread to wait for the specified milliseconds, until another thread notifies (invokes notify() or notifyAll() method). |
| public final void **wait**() throws InterruptedException | causes the current thread to wait, until another thread notifies (invokes notify() or notifyAll() method). |
| protected void **finalize**() throws Throwable | is invoked by the garbage collector before object is being garbage collected. |

- The Object class provides some common behavior to all the objects such as object can be compared, object can be cloned, object can be notified etc.

```
class Point {
      int x, y;

      Point(int a, int b)
      {
            x=a; y=b;
      }
      /* public String toString()
      {
            return "x = " + x + " , y = " + y ;
      }*/ }
public class Demo {
      public static void main(String [] a)
      {
            Point P1 = new Point(10, 20);
      System.out.println(" " + P1.getClass() );
      System.out.println(" Values : " + P1 );
      }
 }
```

**Output without overriding toString()**

class Point
Values : Point@e83912

**Output with overriding toString() (after un-commenting)**
class Point
Values : x = 10 , y = 20