Module 3

Classes, Inheritance,Exception Handling:
Classes: Classes fundamentals; Declaring objects; Constructors, this keyword, garbage collection.
Inheritance: inheritance basics, using super, creating multi level hierarchy, method overriding.
Exception handling: Exception handling in Java.

Shortcut key to type **System.out.println**

Type sysout ctrl+space

To add **getters** and **setters** function for each data members

Create the required class

Place the cursor inside class & right click

Choose option “Source”

## Class in Java

"A class is a template for an object, and encapsulates the fields and methods for the object. Class methods provide access to manipulate fields.

" fields" of an object are often called instance variables."

Ex:

```
class Rectangle  {
        private int length;   //Data Member or instance Variables or fields
        private int width;

        public Rectangle()  { } // ZPC

public void getdata(int x,int y)   //Methods or Instance level functions or member functions
        { length=x;width=y;  }

        public int rectArea()    //Method returning a value
        {  return(length*width);  }
    }
```

## Class in Java

```
class RectangleArea       {
   public static void main(String args[])  {
        Rectangle rect1=null;
        Rect1 = new Rectangle();
 // object creation.  rect1 is a reference type variable which can hold a null value;
// rect1 is a referential entity which refers to an instance.

    rect1.getdata(10, 20);   //invoking methods using object with dot(.)
    int area1  =  rect1.rectArea();                System.out.println("Area1="+area1);
  }  //End of main
}//End of class
```

After defining a class, it can be used to create objects by instantiating the class.

Each object acquires memory to hold information for its instance variables **(i.e. its state)**.

**Class in Java** : **Creating instance of a class/Declaring objects**

```
Rectangle rect1=new Rectangle();
                    OR
Rectangle rect1;       // rect1 will be automatically initialized to null value
rect1 = new Rectangle();  // rect1 will be pointing to a memory of type Rectangle
```

Above two statements declares an object rect1 which is of type Rectangle class using **new** operator, this operator dynamically allocates memory for an object and returns a reference to it.

In java all class objects must acquire memory dynamically.

Java's primitive types are not considered as objects, rather they are considered as variables.

```
Rectangle rect2 = rect1;
```

will make both references (rect1 & rect2) to point to the same instance.

There will be no separate instance which will be pointed by rect1 and rect2 (Shallow copy)

**The Constructors**

A constructor initializes an object when it is created.

It has the same name as its class and is syntactically similar to a method.

Constructors have no explicit return type.

Typically, constructors are used to provide initial values to the instance variables defined by the class, or to perform any other startup procedures required to create a fully formed object.

Java automatically provides a default constructor that initializes all member variables to zero if primitive numerical type. If field is a reference type then it will be automatically initialized to null value ( char types to a whitespace).

However, once a constructor is defined, the default constructor is no longer used.

## The Constructors

```
class comp
{
        private int r;                  private char k;
        private comp j;

        public void disp()
        { System.out.println(r+" "+j+" "+k); }
}
public class First {
        public static void main(String[] args) {
                comp a = new comp(); a.disp();
        }
}
```

**this keyword**

this keyword is used to refer to the invoking object.

If there is ambiguity between the instance variable and formal-parameter, this keyword resolves this ambiguity.

Ex:
```
class drd  {
        private int i;
        public void access(int i) {
                i = i;   //LHS 'i' and RHS 'i' are formal parameters
                // this.i = i;  Remedy
                System.out.println(this.i); }
}

drd a = new drd( );      a.access(10);          Output: 0
```

# Program to create a Singly linked list in java

## Garbage Collection

In Java deallocation of memory of objects is done automatically by JVM.

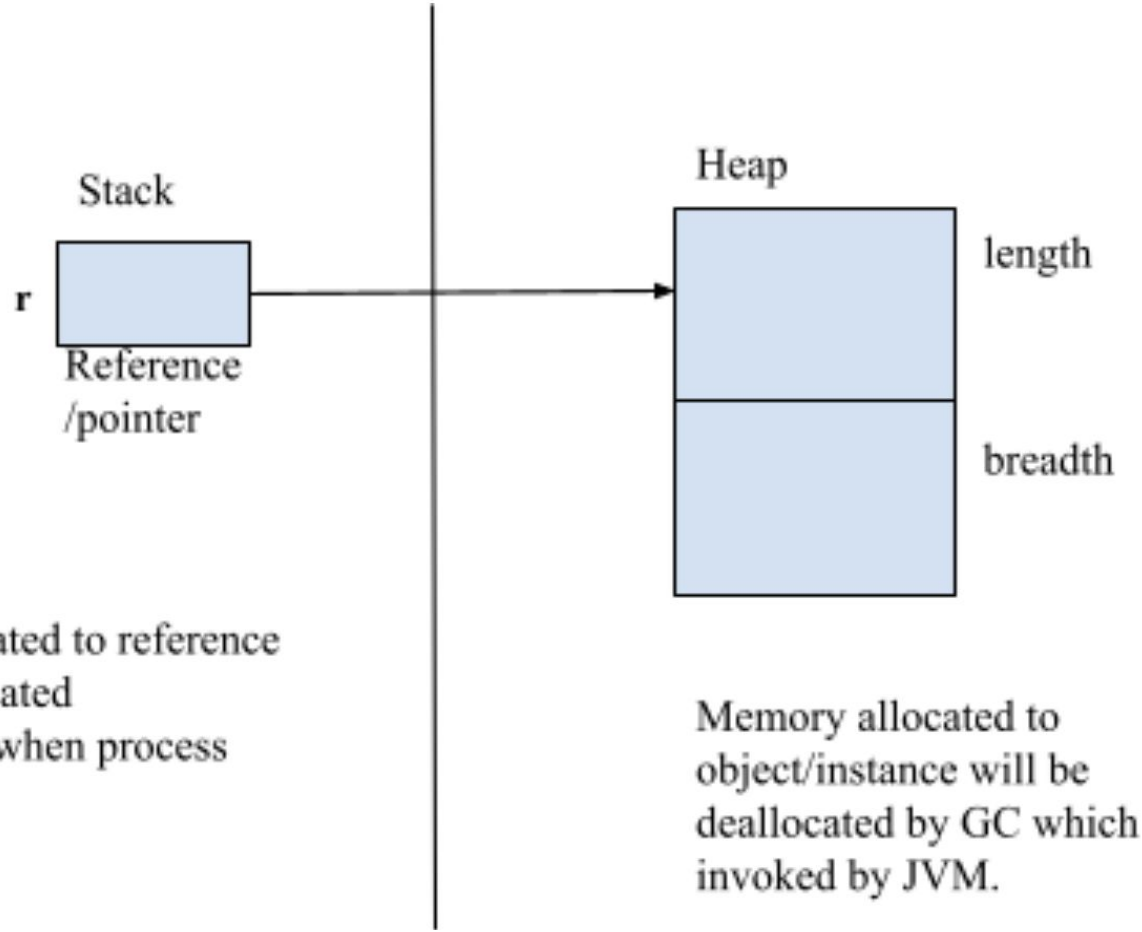
Java contains system programs such as GC, JVM etc.,

JVM helps java source program to get executed on native OS

GC which is also a system program and part of JVM will be invoked by JVM when a process comes to an end, or when it is necessary to reclaim resources such as main memory that are allocated to objects.

```
Rectangle r = new Rectangle( );        r=null;
```

# Garbage Collection



**Garbage Collection**

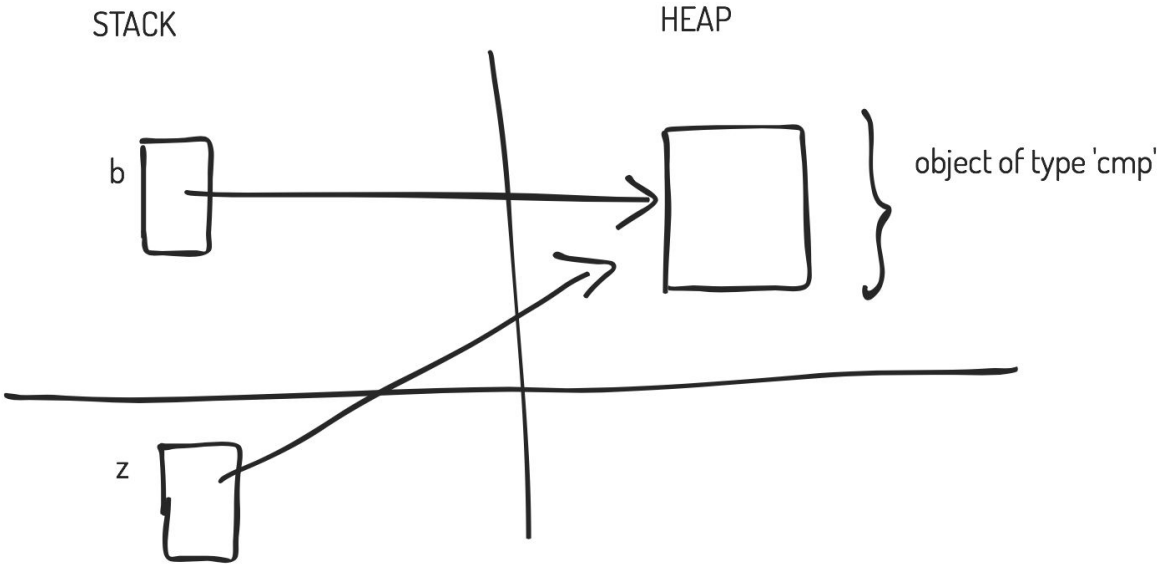
When there is no reference to an object, then that object is assumed to be no longer needed and the memory allocated to the object can be deallocated.

This technique is called Garbage Collection, which is accomplished automatically by the JVM.

Ex:
```
class cmp  {
        public   void   add(cmp z)  {
       // z is an another reference to b & 'this' is an another reference to a invoking object
            cmp t = new cmp();
        }  }

cmp b = new cmp( );    cmp a = new cmp( );      a.add(b);
```

as soon as control is out of add, 'z' reference will be de-allocated as they are local references. Instances pointed by z still remain becz they are being pointed by b.

# Garbage Collection




Consider t, whose **reference (not instance)** will be deleted automatically once the control is out of add, the **object's memory** (dynamically acquired memory) pointed by t will be reclaimed automatically when GC system program is invoked by JVM.

**Garbage Collection**

There is no fixed time when GC will be called by JVM, it is left to the discretion of JVM to invoke GC.

If a process is executed continuously, without concluding and there is a lot of memory acquisition activity done by the program, then more amount of main memory will be used.

GC will be called oftenly by JVM to reclaim the required memory, whenever needed.

Ex: cmp a = new cmp( ); cmp b = new cmp( );

When control is about to **exit out of main** local references of main such as 'a' & 'b' will be deleted. As soon as the local references are deleted, the memory they are pointing to will be reclaimed by GC because it is the end of the process.

**Garbage Collection**

If there is an abrupt exit out of a process

```
import java.lang.System;
…….
System.exit(0);  // 0 means no errors and control is exiting out of current process
```

Then also JVM takes suitable measures to call GC before control leaves the process to reclaim main memory that has been acquired previously.

In the above scenario instance memory must be re-claimed even if they are being pointed by references. If not done then those locations will account for GARBAGE COLLECTION.

Native OS will not be responsible to deallocate those acquired memory locations, it will be the responsibility of JVM.

## Garbage Collection

Garbage Collection program cannot be called explicitly.

A request can be placed to JVM for garbage collection by calling **System.gc()** method.
gc( ) is a static method of the System class (as it is called by class name).

GC Increases memory efficiency and nullifies the chances for memory leak.

**finalize() method (similar to destructor)**

Sometimes an object needs to perform some specific cleanup task before it is deallocated, such as releasing any resources held or closing an open connection.

**Garbage Collection**

Releasing resources word will not be considered for releasing memory allocations in java

Because dynamically acquired memory locations will be reclaimed automatically in java by GC.

Releasing resources word is more applicable for those resources, which cannot be reclaimed automatically by GC.

Ex: closing the file handler or closing the connection established to DBMS.

To handle such situations, **finalize()** method is used.

Signature of finalize() method which is part of super class Object.

```
protected void finalize()
{
    //finalize-code
 }
```

## Garbage Collection

Necessity of explicit invocation of GC, can be for the following situation.

Ex: Scanner instance created to read information from keyboard must be closed using sc.close( ). Which can be done in finalize( ) method even if it is a static data member.

finalize() method is called by the garbage collection thread (if defined for a class) before deallocating objects.

It's the last step for any object to perform cleanup activity.

**JVM -> GC -> finalize**

**Garbage Collection**

```
Ex:  class temp{
            public void finalize()
            {
                  System.out.println("in finalize method");
            }
      }
class Test {
      public static void main(String[] args)    {
                  temp r = new temp();
                  r=null;
            // if r is not set to null gc() will not be called, if gc() is not called then
            //finalize() will not be called.
                  System.gc();  // just a request of jvm
      }     }
```

**Garbage Collection**

Ex: To release the standard input device attached to scanner

```
import java.util.Scanner;
class node
{  public int info;      public node next; }

class list {
    private node first; // first is just a reference variable
    public static Scanner sc;

    public list() { first=null; }

    public void i_f( )  {   }

    public void disp( )  {   }
```

**Garbage Collection**

```
    protected void finalize()
    {
        System.out.println("In finalize method");
        sc.close(); // valid statement for finalize.
    }   }
public class test {
    public static void main(String[] args) {
        list.sc = new Scanner(System.in);

        list l = new list();    l.i_f();  l.i_f();  l.i_f();  l.i_f();      l.disp();

        l=null;
        System.gc();
        }           }
```

**Chapter 7 is important from a Java language perspective, but not part of syllabus.**

**Topics to be considered and understood**

- Using Objects as parameters
- A Closer Look at Argument Passing
- Returning Objects
- Introducing Access Control
- Understanding static
- Exploring the String Class
- Using Command-Line Arguments

**Inheritance (code reusability)**

One of the most important features of Object Oriented Programming.

Inheritance means to inherit **fields & methods** that are already present in a different class.

Classes which share fields & methods to other classes are termed as **base/super/parent class**.
Class which inherits fields & methods are termed as **derived/sub/child class**.

Inheritance serves the purpose of **code reusability**.

To inherit a class properties in java the keyword used is **extends**.

# Inheritance (code reusability)

| Access Modifier | Within base class | Within derived class |
|---|---|---|
| default | Yes | Yes |
| private | Yes | No |
| public | Yes | Yes |
| protected | Yes | Yes |

**If “Yes” member is accessible in the scope**
**If “No” member is not accessible in the scope**

**Inheritance (code reusability)**

```
class base  {
        int defi;
        static int sdefi;
        void default_disp()
                { System.out.println("in default MF "+defi); }
        static void default_disp_static()
                { System.out.println("in Static default MF "+sdefi); }

        private int pri;
        private static int spri;
        private void private_disp()
                { System.out.println("in private MF "+pri); }
        private static void private_disp_static()
                { System.out.println("in Static private MF "+spri); }
```

**Inheritance (code reusability)**

```
protected int proi;
protected static int sproi;
protected void protected_disp()
        { System.out.println("in protected MF "+proi); }
protected static void protected_disp_static()
        { System.out.println("in Static protected MF "+sproi); }


public int pubi;
public static int spubi;
public void public_disp()
        { System.out.println("in protected MF "+pubi); }
public static void public_disp_static()
        { System.out.println("in Static public MF "+spubi); }
```

## Inheritance (code reusability)

```
        public void base_private_access()
        {
                pri=89;  private_disp();
                spri=89;  private_disp_static();
                System.out.println(this.getClass());
        }
} // end of class base
```

## Inheritance (code reusability)

```
class drd extends base {
        public void access( ) {
            defi = 90;        default_disp();
            sdefi = 90;       default_disp_static();

            base_private_access();

            proi=87;     protected_disp();
            sproi=87;    protected_disp_static();
        }
    }
```

## Inheritance (code reusability)

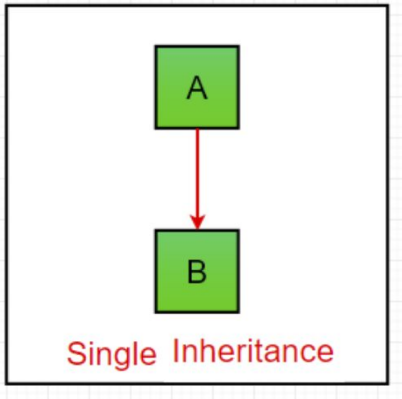
```
public class First {
        public static void main(String[] args) {
                drd d = new drd();
                d.access();

                d.pubi=86;      d.public_disp();
                d.spubi=86;     d.public_disp_static();
        }
}
```

**Types of Inheritance**

1. Single/Simple Inheritance
2. Multilevel Inheritance
3. Multiple Inheritance

Java doesn't support Multiple inheritance but we can achieve this through the concept of **interface**.

**1. Single/Simple Inheritance**



## Single/Simple Inheritance

When a subclass is derived from its parent class then this mechanism is known as simple inheritance.

In case of simple inheritance there is only a sub class and its parent class.

It is also called single inheritance or one level inheritance.
Ex:

## Object Superclass, Representing Single Inheritance

The Object class, in the java.lang package, is the **supreme implicit base class** in java.

Each and every class in java, whether built-in or user-defined inherits the properties of supreme base class Object.

Several methods are available in Object class, which can be called by derived classes without overriding them or by overriding them (final methods cannot be overridden).

Some of the methods in Object class are as follows.

```
public final Class getClass()
public int hashCode()
public boolean equals(Object obj)
public String toString()
public final void notify()
```

## Object Superclass

“final” member functions of Object class can be inherited and used in derived class.

Non-final member functions of Object class can be inherited and overridden in derived class.

Overridden functions provides the facility to experience Specialized functions in derived classes, in class hierarchy.

Functions defined in Object class scope provides Generalized task.

Functions that are inherited and if extended further in derived class provides Specialized task.

It is not compulsory to override the non-final member function that are inherited from base class.

## Object Superclass

```
class comp
{
    private int r,i;

    public comp(int r, int i)
    { this.r = r;  this.i = i;  }

    public String toString( ) // overridden function
    {
        System.out.println("In generalized toString( ) \n"+super.toString() );
        System.out.println("In Overridden Specailized toString() ");
        String a = r +"+i"+i;
        return a;
    }
```

## Object Superclass

```
        public boolean equals(Object obj) //overridden function
        {
            // a is referred by this reference
            // b is referred by base class reference
            System.out.println("In generalized equals( ) ");
            System.out.println(super.equals(obj));
            System.out.println("In Overridden Specialized equals( ) ");

            comp t = (comp)obj;

            if (this.r == t.r && this.i == t.i)
                return true;
            return false;
        }
    }
```

## Object Superclass

```
public class First {
    public static void main(String[] args) {
        comp a = new comp(1,2);      comp b = new comp(1,2);

        System.out.println("Calling equals( ) ");
        System.out.println("******* ********* ");
        if (a.equals(b))
            System.out.println("a is equal to b");
        else
            System.out.println("a is not euqal to b");

        System.out.println("\n\nCalling toString() ");
        System.out.println("******* ********** ");
        System.out.println(a);
    }        }
```

**An example of Single inheritance**

```
class Bicycle {   //base class
    public int gear, speed;
    public Bicycle(int gear, int speed)
    {       this.gear = gear;      this.speed = speed;    }
    public void applyBrake(int decrement)
    {     speed -= decrement;     }
    public void speedUp(int increment)
    {    speed += increment;     }
    @Override
    public String toString()
    {
        String a = "No of gears are "+ gear +"\n";
        a = a + "speed of bicycle is "+ speed;
        return a;
    }     }
```

**An example of Single inheritance**

```
class MountainBike extends Bicycle     {
        public int seatHeight;
        public MountainBike(int gear, int speed,  int startHeight)   {
                // invoking base-class(Bicycle) parameterized constructor
                super(gear, speed);
                seatHeight = startHeight;       }
        public void setHeight(int newValue)
        {    seatHeight = newValue;      }
        @Override
        public String toString()
        {
                String a =  super.toString()+ "\n"+"seat height is ";
                a = a + seatHeight;
                return a;
        }      }
```

**An example of Single inheritance**

```
public class First {
        public static void main(String args[]) {

                MountainBike mb = new MountainBike(3, 100, 25);
                System.out.println("Mountain bike\n"+mb);



                Object bi = new Bicycle(1,2);
    //base class reference,  referencing an instance of derived class type
                System.out.println("\n\nBicycle\n"+bi);
        }
}
```

**"super" keyword**

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

**super(*arg-list*);**

*arg-list* specifies any arguments needed by the constructor in the superclass.

**super( )** (base class constructor call) must always be the first statement executed inside a subclass' constructor.

**A Second Use for super**

G.F: super.member

Here, *member* can be either a member function or a datamember.

This second form of super is most applicable to situations in which member names of a subclass hide members by the same name in the superclass (overridden functions).

This type of invocation can be coded in any part of the function.

## “super” keyword

```
class base{  protected int i;  }

class drd extends base  {
        private int i;
        public void access() {
                super.i=10;
                i=20;
                System.out.println("i in base " + super.i);
                System.out.println("i in drd " + i);
        }
}
public class First {
        public static void main(String args[]) {
                drd  d = new drd();          d.access();
        } }
```

## A Superclass Reference can refer to a Subclass Object

A reference variable of type superclass can be assigned to an instance of subclass.

By making a base class reference, refer to instance of derived class,
**Members that are inherited from base class to derived class can be accessed by a base class reference. Members of the derived class cannot be accessed by base class reference.**

**Most important use of base class reference is "one interface, multiple access".**

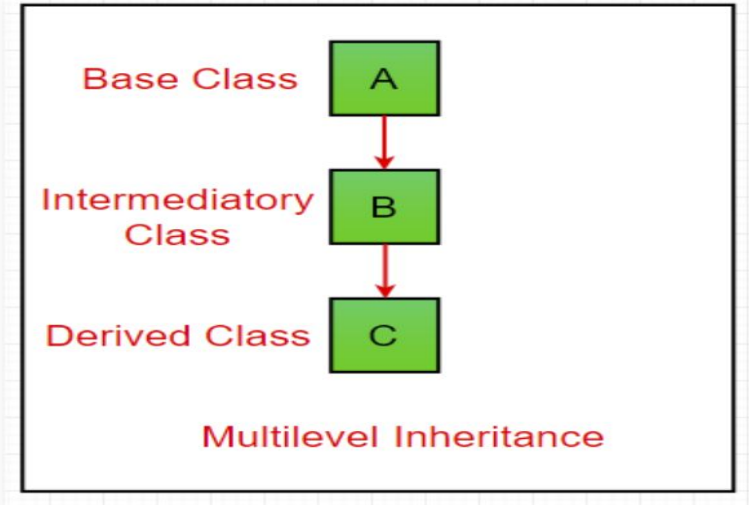
By using a single base class reference, members that are passed to derived classes can be accessed at any level of inheritance.

**(Considering, overridden functions, since they are inherited and overridden in derived class, they can also be accessed by base class references)**

"One interface, multiple access" is possible only when **mutually exclusive** classes are linked together with inheritance hierarchy.

**Multilevel Inheritance**

When a class is inheriting from a derived class then this mechanism is known as **"multilevel inheritance".**




Ex: B is inheriting properties from class A and B is also base class for derived class C

There is no restriction imposed on multilevel inheritance.

**Multilevel Inheritance**

```
class person    {
    private String name;
    public person(String s)
    {    setName(s); }

     public void setName(String s)
    {  name = s;    }

    public String getName()
    { return name; }

    @Override
    public String toString()
    {   return "Name = " + name;    }   }
```

**Multilevel Inheritance**

```
class Employee extends person     {
        private int empid;
        public Employee(String sname, int id)
        {    super(sname);       setEmpid(id);    }

        public void setEmpid(int id)   { empid = id; }


        public int getEmpid()      { return empid; }

        @Override
        public String toString()  {
          String a = super.toString()+" ";
          a = a + "Empid = " + empid;
          return a;
        }
}
```

**Multilevel Inheritance**

```
class HourlyEmployee extends Employee    {
    private double hourlyRate;          private int hoursWorked;

    public HourlyEmployee(String sname, int id, double hr, int hw)  {
      super(sname,id);          hourlyRate = hr;     hoursWorked = hw;       }

    public double GetGrosspay()
    {   return (hourlyRate * hoursWorked);  }
    @Override
    public String toString()  {
            String a = super.toString()+" ";
            a = a + " Hourly Rate = " + hourlyRate;
            a = a + " Hours Worked = " + hoursWorked;
            a = a + "Gross pay = " + GetGrosspay();
     return a;                }    }
```

**Multilevel Inheritance**

```
class First
{
    public static void main(String[] args)
    {
        HourlyEmployee emp = new HourlyEmployee("AB",1,15,1800);
        emp.GetGrosspay();
        System.out.println(emp);
    }
}
```

**Constructors in inheritance hierarchy**

Constructor in the inheritance hierarchy will be called in the order of derivation.

That is first constructor of base class will be called, followed by the constructor in the derived class. **Constructors are executed in the order of hierarchy.**

Ex:
```
class A {
    public A()  {   System.out.println("Inside A's constructor."+this.getClass( ));   }
}

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

class C extends B   {
    public C()  {    System.out.println("Inside C's constructor."+this.getClass( )); }
}
```

**Constructors in inheritance hierarchy**

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

}
```

Output: Inside A's constructor.
Inside B's constructor.
Inside C's constructor.

**Method with different signatures acting as overloaded instance methods**

```
Ex: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; }
       // overload show()
        void show(String msg)   {  System.out.println(msg + k);    }
      }

            B subOb = new B(1, 2, 3);
            subOb.show("This is k: "); // this calls show() in B
            subOb.show(); // this calls show() in A
```

**Dynamic method dispatch (DMD)**

Dynamic method dispatch is the mechanism by which a call to an overridden method is resolved at run time, rather than at compile time.

Main requirements for DMD are

1. Inheritance hierarchy (at least one parent child relationship).
2. Overridden function in derived class.
3. Base class reference, pointing to derived class instance.
4. Calling overridden function by base class reference.

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 (DMD)**

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.

**Generalization and Specialization**

Function in base class exhibit generalized property.

Overridden functions in derived classes exhibit specialized property.

Ex: toString( ) method in Object base class and overridden toString( ) in user-defined classes.

## Generalization and Specialization

Overridden methods allow Java to support run-time polymorphism.

**Polymorphism is essential to object-oriented programming for one reason: it allows a general class to specify methods that will be common to all of its derivatives, 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" aspect of polymorphism.

Part of the key to successfully applying polymorphism is understanding that the superclasses and subclasses form a hierarchy which moves from lesser to greater specialization. Used correctly, the superclass provides all elements that a subclass can use directly. It also defines those methods that the derived class must implement on its own. This allows the subclass the flexibility to define its own methods, yet still enforces a consistent interface.Thus, by combining inheritance with overridden methods, a superclass can define the general form of the methods that will be used by all of its subclasses.

**Difference between Method overloading and overriding**

**Method Overloading**
Occurs at compile time.
Different types and number of parameters must be passed to different function
Return type can be different for different functions.
Scope of the functions must be same

**Method Overriding**
Occurs at execution time.
Parameters must be the same in type and number.
Return type cannot be different, it has to be the same.
Scopes of the overridden functions will be in different classes; scopes that share hierarchical relationships between them.

**Abstract class**
Abstract class is a collection of one or more undefined member functions.

**Abstract classes are used to provide some common functionality across a set of related classes.**
**Word “functionality” does not signify generalized property, but signifies the compulsion on derived classes to provide the functionality.**

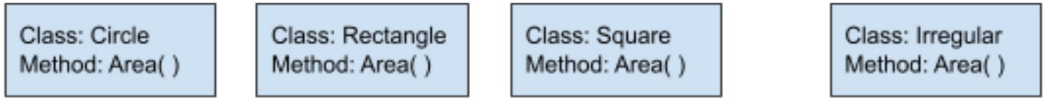
Definition (of undefined methods) must be done compulsorily in derived class.

Some base classes will be unable to provide a generalized functionality for the hierarchy, these methods are termed as **"abstract methods"**

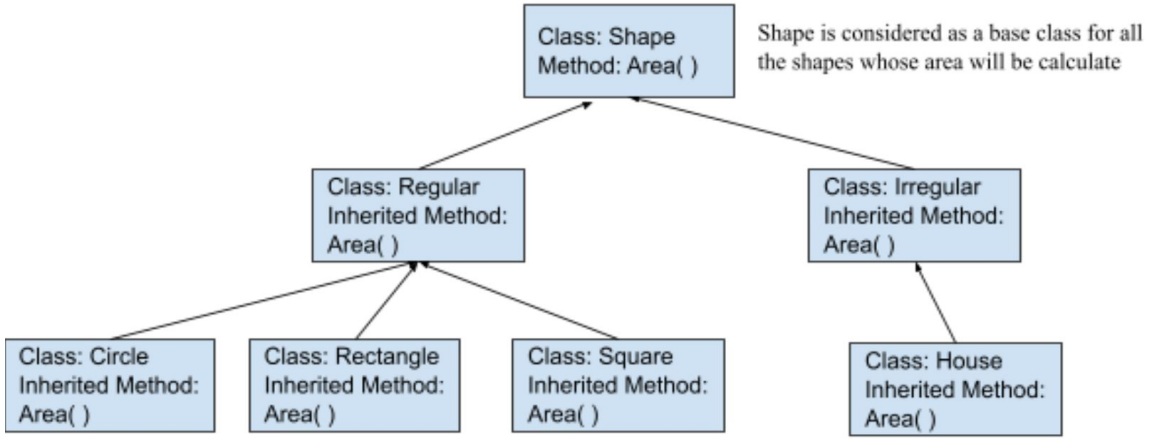
Class which contains at least one abstract method is termed as **"abstract class"**.

*(Abstract means nonrealistic)*

# Abstract class : Another scenario, where abstract class is required




**Any modification needed on all these classes (like to add method to find volume( ) ), requires changes in each class, since they are all different classes**




**Now, if we want all the shapes to calculate volume of it, then only one modification is enough to base class Shape.**
**Adding abstract method volume() to class Shape, will pass the properties to all subclasses.**

**ONE OF THE USAGES OF INHERITANCE AND ABSTRACT CLASSES.**

**Why abstract class ???**

Consider an application program to process **employee** and **student type** information.


**Why abstract class ???**

Further, it can be observed that 2 classes have some fields in common, like name, gender and address.

Consider, methods such as accept( ) and display( ) to be present in both classes.

These common information can be accumulated in a base class, as shown below.

Ex:
```
class person {
        private String name,gender
        private String addr;

        public void  accept( );
        public void display( );  }
```

The member functions can be defined in class person, but the object of type student and employee has significance rather than the generic object type person.

Hence, class person will be converted as abstract and the method definition will be delegated to the derived class, which are going to inherit the properties of class person.

**Abstract class**

An abstract class is defined in java using the keyword “abstract”.

If a single member function in a class is defined as abstract, then the class must be compulsorily defined as abstract class.

Considering, the previous example, methods accept( ) and display( ) will be defined as abstract as follows.

Ex:

```
abstract class person
{
        private String name, gender;
        private String addr;

        public abstract void  accept( );
        public abstract void display( );
}
```

**Abstract class**

The class person can be used as a base class for student and employee class.

The derived class student and employee has to **compulsorily** override the accept() and display() member functions that are inherited and not defined in abstract class person.

Ex:

```
abstract class person  {
        protected String name;

        public abstract void accept( );

        public String toString() {
            String a = name;
            return a;
        }
}
```

**Abstract class**

```
class input
{  public static Scanner sc = new Scanner(System.in);  }

class student extends person {
        String usn;

        public void accept()
        { name=input.sc.next();    usn=input.sc.next(); }

        public String toString()
        {
                String a = super.toString();
                a = a + " " + usn ;
                return a;
        }
}
```

**Abstract class**

```
class employee extends person {
        String eid;

        public void accept()
        { name=input.sc.next();    eid=input.sc.next(); }

        public String toString()
        {
                String a = super.toString();
                a = a + " " + eid ;
                return a;
        }
}
```

**Abstract class**

```
class First
{
    public static void main(String [] args)
    {
        student s = new student();
        s.accept();
        System.out.println(s);

        employee e = new employee();
        e.accept();
        System.out.println(e);
    }
}
```

**Abstract base class reference**

```
person p = new student();    p.accept();
System.out.println(p);

p = new employee();              p.accept();
System.out.println(p);
```

If there is a member defined in class student or employee which is not inherited from abstract base class, then those member cannot be invoked by, abstract base class reference p.

```
p.name = "abc";  works because it is inherited from
p.usn = "123" ; //CTE
```

Assume class student contains a member function access( ) of its own, then

```
p.access( );  //CTE
```

**Abstract base class with constructors**

If there is instance level fields in abstract class, **it can be** initialized with the constructors of abstract class.

(NOTE: an instance of abstract base is not created here, but an instance of abstract base is created in derived class)

After creating an instance of type derived class, the abstract base class constructor is called on the instance of it, which is present in derived class.

Ex:
```
abstract class person  {
        protected String name;

        public person(String n)  {
            System.out.println("in person constr "+this.getClass());
            name=n;  }
        public abstract void accept( );
```

**Abstract base class with constructors**

```
        public String toString() {
                String a = name;
                return a;
        } }
        class student extends person {
                String usn;
                public student(String n, String u)   {
                        super(n);
                        System.out.println("in student constr");
                        usn=u;     }

        public void accept() { }
```

**Abstract base class with constructors**

```
        public String toString()  {
                String a = super.toString();
                a = a + " " + usn ;
                return a;
        }  }
        class First   {
                public static void main(String [] args)  {
                        person p = new student("a","1");
                        System.out.println(p);
                }      }
```

Output:

```
in person constr  class first.student
in student constr
a 1
```

**Properties of abstract class**

An object of abstract class cannot be instantiated, if done it raises CTE.
Ex: person p = new person(); ///CTE

Reference of an abstract class type can always be created.

Multiple abstract classes/Multiple base classes cannot be inherited by a single derived class.
Ex: class drd extends base, extends base1 // CTE

Any derived class of an abstract class **must define all of the abstract methods** in the superclass, or be itself declared abstract (must be another abstract class)

Abstract class can have abstract and non-abstract methods.

Abstract class can have final methods, which must be defined in abstract class itself.

Abstract class can have constructors and static methods also.

**Properties of abstract class**

Abstract class can have data members and static DM also.

Data members of abstract classes will acquire memory only in derived classes.

Member function of abstract class will be invoked by the object of derived class.

## Abstract class

```
/*Abstract class can have abstract and non-abstract methods,
  Abstract class cannot be instantiated.
  Abstract class can have final methods.
  Abstract class can have constructors and static methods also. */
abstract class base {
        public  int i;  // data member of abstract class   field
        abstract void abm(); // declared method not defined
        public void nonabm()   // defined member function
        {  System.out.println("in non abstract method"); }
        final public void finonabm() // final method
        { System.out.println("final non-abstract method"); }
        public static void snonabm() // static member function
        { System.out.println("Static method of abstract class"); }
        public base() // constructor
        { System.out.println("ZPC abstract base class");              i=90;        }
}
```

## Abstract class

```
class der extends base {
        public der()
        { System.out.println("ZPC der"); }

        public void abm()
       {  System.out.println("Defined function - Abstract type");  }
}
class test {
 public static void main(String[] args)  {
                base d = new der();
                d.i = 900; // base class data member
                d.abm(); // invoking defined function - abstract function
                d.snonabm(); // invoking static non abstract member function
                d.finonabm(); // invoking final non abstract member function
                d.nonabm(); // invoking non-final non abstract member function
         }          }
```

**“final” keyword in Java**

“final” keyword in java has 3 uses

1. can be used to create a constant
2. can be used on method to avoid overriding
3. can be used to prevent inheritance

**Constant creation**

final members must be initialized at the point of declaration.

**final fields can be initialized inside the constructor. If more than one constructor is there for the class, then it must be initialized in all of them, otherwise compile time error will be thrown.**

```
Ex:  class test {
            public static void main(String args[]) {
                 final int i = 10;
                      i=i+1; //CTE
      } }
```

**“final” keyword in Java**

**final to Prevent Overriding**

If a method in the base class is defined using the final keyword, then these members cannot be overridden in derived classes.

Ex:
```
class A {
        final void meth() {  System.out.println("This is a final method.");  }
}
class B extends A {
        void meth( ) { // ERROR! Can't override.
                System.out.println("Illegal!");
        } }
```

Because meth( ) is declared as final, it cannot be overridden in B. If an attempt is made, a compile-time error will result.

Overridden method call decisions are usually made during run-time, since final methods are not overridden, these method invocation is decided during run-time itself.

**“final” keyword in Java**

**final to Prevent Inheritance**

In order to prevent a class from inheriting, it can be preceded by the keyword final.

Declaring a class as final implicitly declares all of its methods as final.

A class in java cannot be created as both abstract and final.

Ex:

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

**Exception in java**

Exception word is synonym for runtime errors.

Exceptions are events that occur during the execution of programs that stop execution.

Ex: divide by zero, array access out of bound, File not found, NullPointer Exception etc

```
class test {
        public static void main(String[] args) {
                int i=10;
                i=i/0;
                System.out.println("After the statement");
 } }
```

Exception generated is as follows

Exception in thread "main" java.lang.ArithmeticException: / by zero

Since, exceptions are runtime errors, information regarding the same will be encapsulated within an object in java.

**Exception in java**

Exception information contains..

Reason why exception has been generated ( / by zero )
package name, class name, method name
line number "hello.test.main(test.java:15)"
etc., where runtime error was generated

will be encapsulated or bundled in an object by JVM and will be handled either by JVM or programmers.

If JVM catches an exception the only remedy is that process comes to an halt.

If a program catches the exception by using built-in keywords in java, a remedy can be provided for the same in the program and program execution can continue (Robust).

(Robust : "Java programs are Robust"

**Software Engineering definition for Robust**

Resistant or impervious to failure regardless of user input or unexpected conditions.)

**Exception in java**

```
class cmp {
        public void access()
        {  }
}

class test {
    public static void main(String[] args)   {
        cmp i=null;
        i.access();
        System.out.println("After the statement");
     }
 }
```

```
Exception in thread "main" java.lang.NullPointerException
        at hello.test.main(test.java:20)
```

**Exception in java**

Exceptions that are generated usually indicate different types of error conditions.

Some common built-in exception classes available in Java are.

FileNotFoundException *to handle file not found condition*
IOException *to handle input output error condition*
SocketTimeoutException
NullPointerException *Dereferencing a null reference*
ArrayIndexOutOfBoundsException *Trying to read outside the bounds of an array*
ArithmeticException *Dividing an integer value by zero*
*etc.,*

Five keywords are used in java to handle exceptions in program itself

**try, catch, throw, throws** and **finally**.

**Exception in java**

**G.F:**

```
try
 {
      Statements that may generate exception or run-time error
      Since, each exception has a name associated with it, catch block will be coded to
      catch that type of exception.
 }
 catch(Type_of_exception e)
 {
    Remedy for exception, will be provided here.
    "e" is a reference variable which is ready to point to the exception object that will be
     generated by JVM
 }
```

**Exception** is a built-in class in java which is the supreme base class for all types of exception classes that are available in java.

**Exception in java**

```
class cmp {
      public void access() {  }
}
class test {
     public static void main(String[] args)   {
       cmp i=null; // this will not generate exception
       try {
             i.access();  // this will generate exception; i.e NullPointerException
          }
        catch(Exception e)  //Generic catch block
       {
              System.out.println(e);
/* the above statement prints the name of the exception generated, which is not a remedy
    for the generated exception.  */
       } }   }
```