Object:

A thing in a real world that can be either physical or conceptual

An object in object oriented programming can be physical or conceptual

Bulb is a physical object and college is a conceptual object.

Object has state and behavior.

An object has three characteristics:

- **State:** represents data (value) of an object.
- **Behavior:** represents the behavior (functionality) of an object such as deposit, withdraw etc.
- **Identity:** Object identity is typically implemented via a unique ID. The value of the ID is not visible to the external user. But, it is used internally by the JVM to identify each object uniquely.

Attributes and Operations:

- The object's state is determined by the value of its properties or attributes.
- Properties or attributes → member variables or data members
- The object's behavior is determined by the operations that it provides.
- Operations → member functions or methods

A bulb:

1. It's a real-world thing. ~~Object~~
2. Can be switched on to generate light and switched off. → Methods
3. It has real features like the glass covering, filament and holder. → Member Variables
4. It also has conceptual features like power.
5. A bulb manufacturing factory produces many bulbs based on a basic description / pattern of what a bulb is. Class →

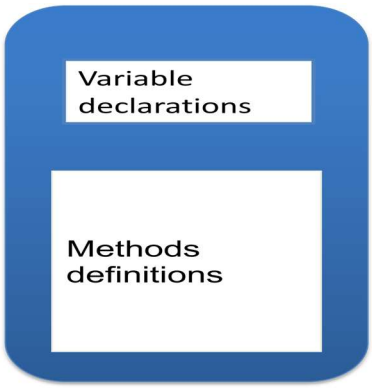
Class

A class is a group of objects that has common properties.

A class is a construct created in object-oriented programming languages that enables creation of objects.

Also sometimes called blueprint or template or prototype from which objects are created.

- It defines members (variables and methods).
- A class is an **abstraction.**

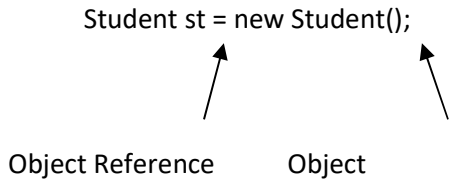



```
class Student
{
    int rollno;
    String name;

    void setStudent(int r, String n)  //method
    {
          rollno=r;
          name=n;
    }
    void displayStudent()  //method
    {
           System.out.println(rollno+" "+name);
    }
  }
```

Student stud1;

Employee emp1;

- The **new** operator is used to create an object of that reference type



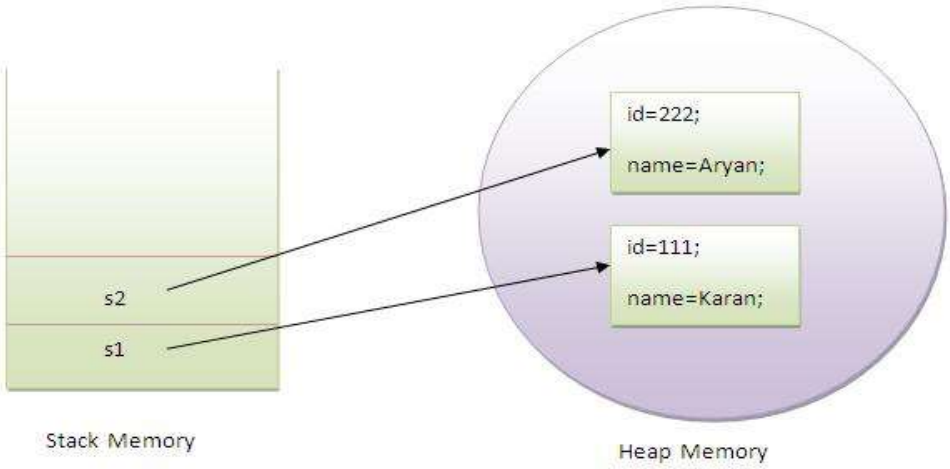
E.g

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

    void setStudent(int r, String n)   //method
    {
        rollno=r;
        name=n;
     }

    void displayStudent()    //method
    {
        System.out.println(rollno+" "+name);
     }

    public static void main(String args[])
    {

        Student s1=new Student();  //object creation
        Student s2=new Student();  //object creation

        s1.insertRecord(111,"Karan");
        s2.insertRecord(222,"Aryan");

        s1.displayInformation();
        s2.displayInformation();

}
}
```

Memory Allocations:

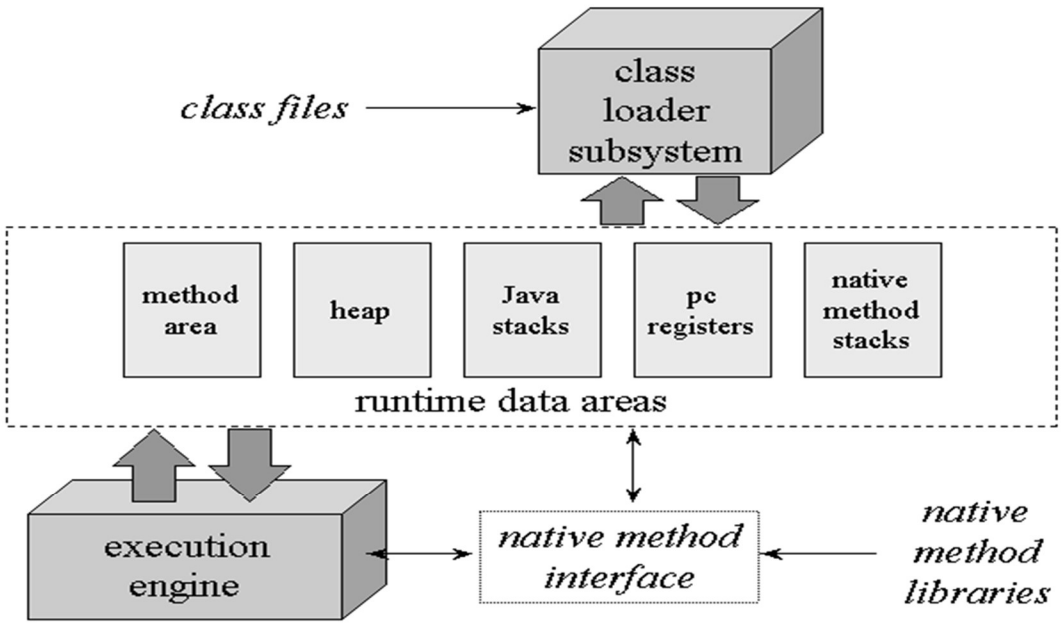
- Stack – for Local variables

- Heap – for references

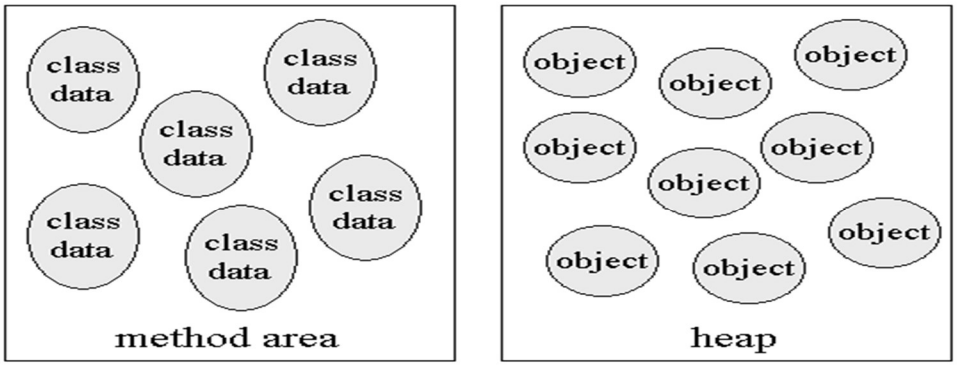


When a Java virtual machine runs a program, it needs memory to store many things, including byte codes and other information it extracts from loaded class files, objects the program instantiates, parameters to methods, return values, local variables, and intermediate results of computations. The Java virtual machine organizes the memory it needs to execute a program into several *runtime data areas*.

The internal architecture of the Java virtual machine.



When the virtual machine loads a class file, it parses information about a type from the binary data contained in the class file. It places this type information into the method area. As the program runs, the virtual machine places all objects the program instantiates onto the heap.



Constructor:

Constructor in java is a *special type of method* that is used to initialize the object.

Java constructor is *invoked automatically at the time of object creation*.

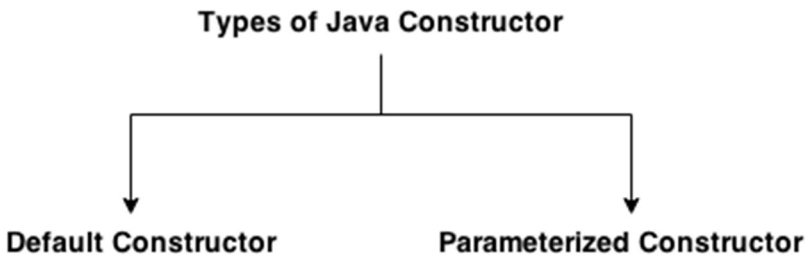
Rules to define a constructor

- A constructor has the same name as the class name
- A constructor should not have a return type, even void.
- A constructor can be defined with any access specifier (like private, public)
- There can be more than one constructor for a class. (constructor overloading)
- Space is allocated for an object only when the constructor is called. Declaring a variable of class and not calling new does not consume any memory!

Type of Constructors:

There are two types of constructors:

1. Default constructor (no-arg constructor)
2. Parameterized constructor



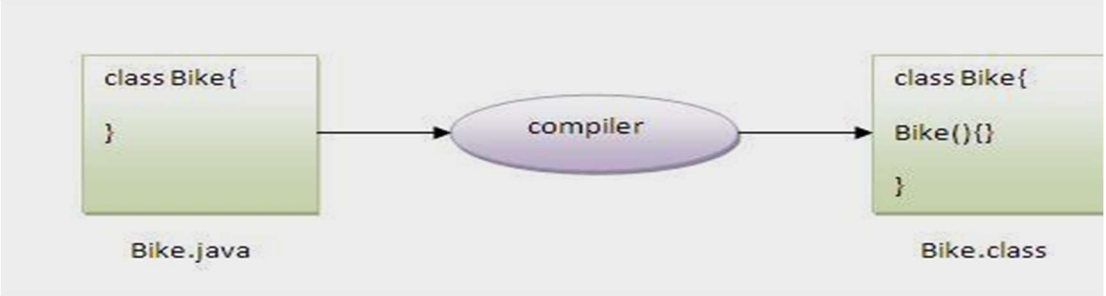
Default Constructor:

A constructor that has no parameter is known as default constructor.

```
class Sample
{
   private int id;
```

```
  Sample()  {

        id = 101;

       System.out.println("No Arg constructor, with ID: "+id);

     }

}
```

Rule: **Rule: If there is no constructor in a class, compiler automatically creates a default constructor.**



Java parameterized constructor

A constructor that has parameters is known as parameterized constructor.

Why use parameterized constructor?

Parameterized constructor is used to provide different values to the distinct objects.

```
class Student
{
   int id;
   String name;

   Student4(int i,String n)
   {
        id = i;
        name = n;
   }
   void display()
   {
```

```
        System.out.println(id+" "+name);
    }

    public static void main(String args[])
    {
        Student4 s1 = new Student4(111,"Raj”);
        Student4 s2 = new Student4(222,”Sam");
        s1.display();
        s2.display();
    }
}
```

Constructor Overloading

Defining more than one constructor inside a class, but parameter should be differ in the following ways:

1. no. of parameters

2. type of parameters

3. order of parameters

```
class Student
{
    int id;
    String name;
    int age;

    Student(int i,String n)
    {
        id = i;
        name = n;
    }
    Student(int i,String n,int a)
    {
    id = i;
    name = n;
    age=a;
    }
    void display()
    {
    System.out.println(id+" "+name+" "+age);
```

```
    }
    public static void main(String args[])
    {
    Student s1 = new Student5(111,"Karan");
    Student s2 = new Student5(222,"Aryan",25);
    s1.display();
    s2.display();
}
}
```

this reference keyword

- Each class member function contains an implicit reference of its class type, named this
- this reference is created automatically by the compiler
- It contains the address of the object through which the function is invoked

Use of this keyword

- this can be used to refer instance variables when there is a clash with local variables or method arguments
- this can be used to call overloaded constructors from another constructor of the same class

1. To overcome shadowing and allow a parameter or local variable to be the same name as an instance field

```
class Student
{
    String name;

    setName(String name)
    {
       this.name=name;
    }
}
```

2.To call a constructor from another constructor of the same class

```
class Sample
{
     Sample()
    {
        this("Java");       // calls overloaded constructor
        System.out.println("No Arg constructor ");
     }
     Sample(String str)
    {
        System.out.println("One argument constructor "+ str);
     }
}
```

Rule: Call to this() must be the first statement in constructor.

Static Memebers:

Static class members are the members of a class that do not belong to an instance of a class
They are shared by objects of same class.

Are created using the **static** modifier.

Are initialized even before the instance variables are initialized and can be accessed even if the objects are not created.

We can apply java static keyword with variables, methods, blocks and nested class.

The static can be:

1. variable (also known as class variable)
2. method (also known as class method)
3. block

We can access static members directly by prefixing the members with the class name

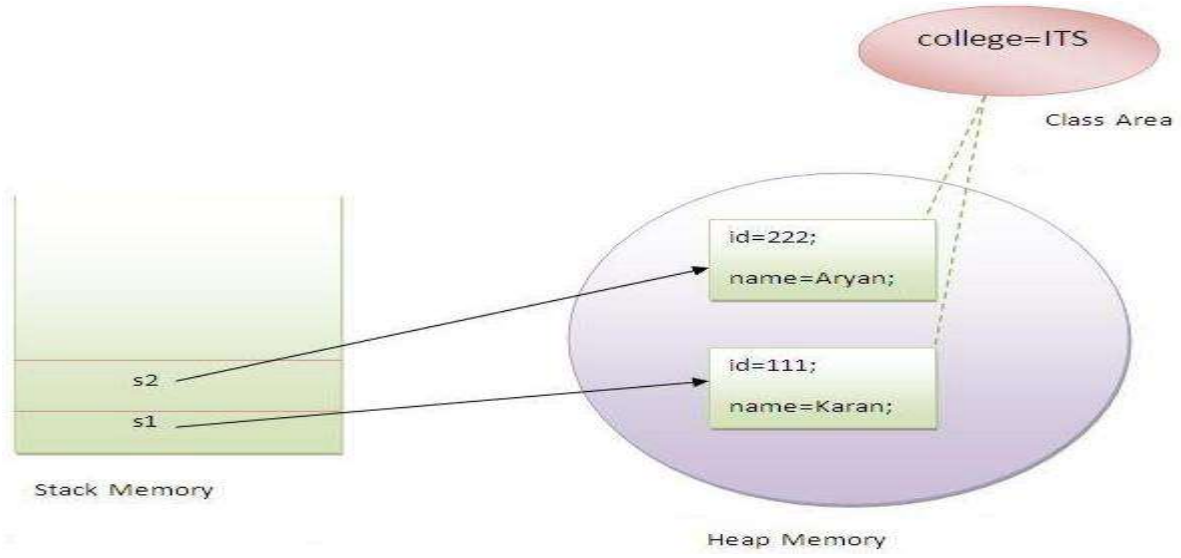
```
ClassName.staticVariable
ClassName.staticMethod(...)
```

Static variables:

- Shared among all objects of the class
- Only one copy exists for the entire class to use
- The static variable gets memory only once in class area at the time of class loading.

E.g

```
class Student
{
  int id;
  String name;
  static String college ="ITS";

 Student(int r,String n)
 {
     id = r;
     name = n;
  }
 void display ()
 {
    System.out.println(id+" "+name+" "+college);
 }

 public static void main(String args[]){
 Student s1 = new Student8(111,"Karan");
 Student s2 = new Student8(222,"Aryan");

 s1.display();
 s2.display();
 }
}
```



Static methods:

- Static methods can only access directly the static members and manipulate a class's static variables
- A static method can be invoked without the need for creating an instance of a class.
- Static methods cannot access non-static members(instance variables or instance methods) of the class
- Static method cant access this and super references

E.g

```
class Student
{
    int rollno;
    String name;
    static String college = "ITS";

    static void change()
    {
        college = "BBDIT";
     }

     Student(int r, String n)
     {
          rollno = r;
          name = n;
      }

    void display ()
    {
           System.out.println(rollno+" "+name+" "+college);
      }

   public static void main(String args[])
  {

   Student s1 = new Student (111,"Karan");
   Student.change();
   Student s2 = new Student (222,"Aryan");
   Student s3 = new Student (333,"Sonoo");

   s1.display();
   s2.display();
   s3.display();
   }
```

```
}
```

E.g 2

```
class Calculate
{
     static int cube(int x)
     {
         return x*x*x;
      }

      public static void main(String args[])
      {
           int result=Calculate.cube(5);
           System.out.println(result);
       }
}
```

Restrictions for static method

1. The static method cannot use non static data member or call non-static method directly.
2. this and super cannot be used in static context.

```
class A
{
      int a=40;//non static

      public static void main(String args[])
      {
       System.out.println(a);   //compiler error,bcoz non static data accessing from static
       }
}
```

Arrays:

Arrays are objects in Java that store multiple variables of the same type.
Arrays can hold either primitives or object references, but the array itself will always be an object on the heap

```
int[] testScores; // Declares the array of ints

testScores = new int[4]; // constructs an array

int[] testScores = new int[4];
```

OOPs:

Abstraction:

Abstraction denotes essential characteristics of an object that distinguish it from all other kinds of objects and thus provide crisply defined conceptual boundaries, relative to the perspective of the viewer.

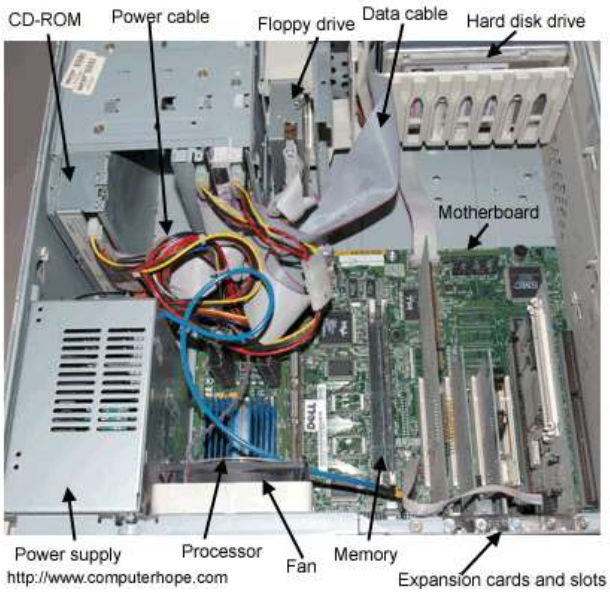
-Grady Booch

- Abstraction is the process of taking only a set of essential characteristics from something.
- Example
    - For a Doctor→ you are a Patient
        - Name, Age, Old medical records
    - For a Teacher→ you are a Student
        - Name, Roll Number/RegNo, Education background
    - For HR Staff→ you are _____

Encapsulation:

Would you like it if your CPU is given to you like this?

What are the problems if it were given to you like this?



Encapsulation is the process of compartmentalizing the elements of abstraction that constitute its structure and behavior; encapsulation serves to separate the contractual interface of an abstraction and its implementation.

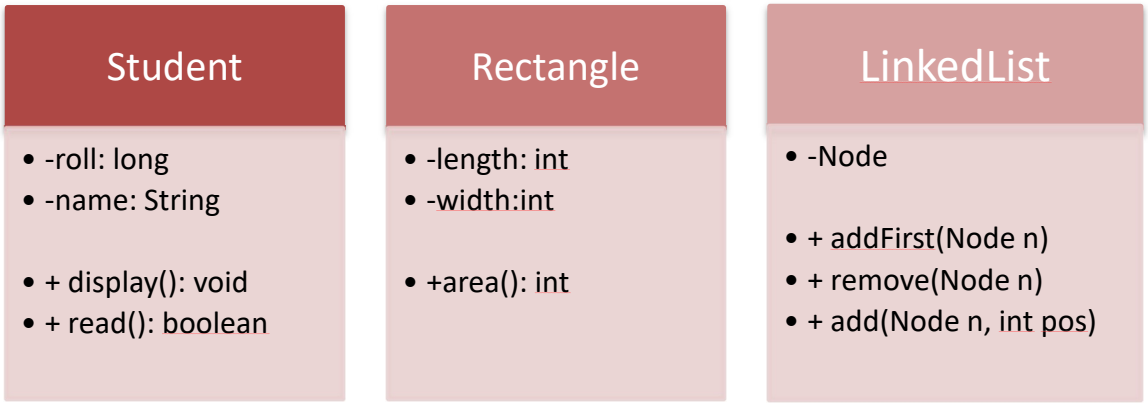
- Grady Booch

- Encapsulation is binding data and operations that work on data together in a construct.
- Encapsulation involves Data and Implementation Hiding.

For example capsule i.e. mixed of several medicines.

Capsule

We can create a fully encapsulated class in java by making all the data members of the class private. Now we can use setter and getter methods to set and get the data in it.



-: private

+: public

Encapsulation is:

- Binding the data with the code that manipulates it.
- It keeps the data and the code safe from external interference

Advantage of Encapsulation in java

By providing only setter or getter method, you can make the class **read-only or write-only**.

It provides you the **control over the data**. Suppose you want to set the value of id i.e. greater than 100 only, you can write the logic inside the setter method.

```
class Point {
	private int x;
	private int y;
	public void setX( int x){
		this.x= (x > 79 ? 79 : (x < 0 ? 0 :x));
	}
	public void setY (int y){
		this.y= (y > 24 ? 24 : (y < 0 ? 0 : y));
	}
	public int getX( ){
		return x;
	}
public int getY( ){
		return y;
	}
}
```

```
class PointDemo {

        public static void main(String args[ ] ){

                int a, b;

                Point p1 = new Point( );

                p1.setX(22);

                p1.setY(44);

                a = p1.getX( );

                System.out.println("The value of a is "+a);

                b = p1.getY( );

                System.out.println("The value of b is "+b);

        }

}
```

Output :

```
The value of a is 22
The value of b is 24
```

Reuse in Object Oriented Language

- Object Oriented Languages also implements reuse in the same way that we do in real life.
- Using
    - has-a
    - is-a
- Has-a or composition relationship is implemented by having a class having another class as its member, or rather an object having another object as its member.
    - Car has a Stereo
    - College has Teachers and Students
- Is-a is implemented through what we call **inheritance** relationship

E.g of Has-A:

```
class Employee
{
int id;
String name;
Address address;//Address is a class
...
}


public class Address {
String city,
String state,
String country;
…
}
```

Inheritence:

Inheritance defines relationship among classes, wherein one class shares structure or behavior defined in one or more classes.

- **Grady Booch**

- Defines IS-A relationship between classes
- Cat IS-A Animal
- Car IS-A Vehicle
- Rose IS-A Flower

- Approach:
    - Common properties of related classes can be defined in a generic class.
    - This generic class can then be used by more specific classes through inheritance.
    - Each of these specific classes can add things that are unique to it.
- As a result of inheritance, a hierarchy of classes is formed.
- The class that is inherited is called super class and the class that is inheriting is called a subclass.

Example scenarios for inheritance

- **Student** and **Teacher** are **Person**. **Person** can be a super class and **Student** and **Teacher** can be subclass of **Person** class. **Student** is-a **Person**, **Teacher** is-a **Person**
- **HOD** is-a **Teacher. Since Teacher** is-a **Person, HOD** is also a **Person**
- **Written_Test** and **Viva** are **Test**
- **Theory and Lab** are **ClassRoomSession**
- **SeminarHall** is a **ClassRoom**

- **extends** keyword is used to indicate inheritance relationship.
- Syntax:

    Class-name2 extends Class-name1

E.g

```
public class HOD extends Teacher {
/*Features of  Teacher class automatically available.
add additional members */
}
```

```
Class Teacher
{
  Int factId;
  String name;
  setFactId();
  setName();
  getFactId();
  getName();
}

Class HOD  extends Teacher
{
  //6 properties from teacher class
   getDateOfJoining();
   viewGrade();
}
```

E.g.

```
class A {
        int m, n;
        void display1( ){
                System.out.println("m and n are:"+m+" "+n);
        }
}
class B extends A {
        int c;
        void display2( ){
                System.out.println("c :" + c);
        }
        void sum(){
                System.out.println("m+n+c = " + (m+n+c));
        }
}

class InheritanceDemo {
        public static void main(String args[ ]) {
                A s1 = new A();
                B s2 = new B();
                s1.m = 10; s1.n = 20;
                System.out.println("State of object A:");
                s1.display1();
                s2.m = 7; s2.n = 8; s2.c = 9;
                System.out.println("State of object B:");
                s2.display1();
                s2.display2();
                System.out.println("sum of m, n and c in object B is:");
```

```
                s2.sum();
        }
}
```

Accessing Super class Members from a Subclass Object

- A subclass includes all of the members of its super class
- But, it cannot directly access those members of the super class that have been declared as private.

```
class A {
int money;
private int pocketMoney;
void fill(int money, int pocketMoney)
{
        this.money = money;
        this.pocketMoney = pocketMoney;
}
public int getPocketMoney()
{
        return pocketMoney;
}
class B extends A {
        private int total;
        void sum( ) {
              total = money + getPocketMoney();
        }
   public int getTotal() {
      return total;
  }
}
class AccessDemo
{
        public static void main(String args[ ])
        {
                B subob = new B();
                subob.fill(10,12);
                subob.sum();
                System.out.println("Total: " + subob.getTotal());
        }
}
```

Super keyword:

- The creation and initialization of the super class object is a prerequisite to the creation of the subclass object.
- When a subclass object is created,
  - It creates the superclass object
  - Invokes the relevant superclass constructor.
    - The initialized superclass attributes are then inherited by the subclass object
  - finally followed by the creation of the subclass object
    - initialization of its own attributes through a relevant constructor subclass
- The constructors of the superclass are never inherited by the subclass
- This is the only exception to the rule that a subclass inherits all the properties of its superclass
- **super** has two uses.
  - The first is the call to the superclass' constructor from the subclass constructor
  - The second usage is to access a member of the superclass that has been overridden by a subclass
- A subclass constructor can call a immediate superclass constructor by the following syntax**:**
  - super(parameter-list);

e.g.

```
class Person {
        private String name;
        Person(String name) {
                this.name=name;
        }
}
class Employee extends Person {
        private int empId;
```

```
        private double salary;

        Person(String name,int empId,double salary) {

                super(name);

                this.empId=empId;

                this.salary=salary;

        }

}
```

The order of invocation of constructors in a class hierarchy

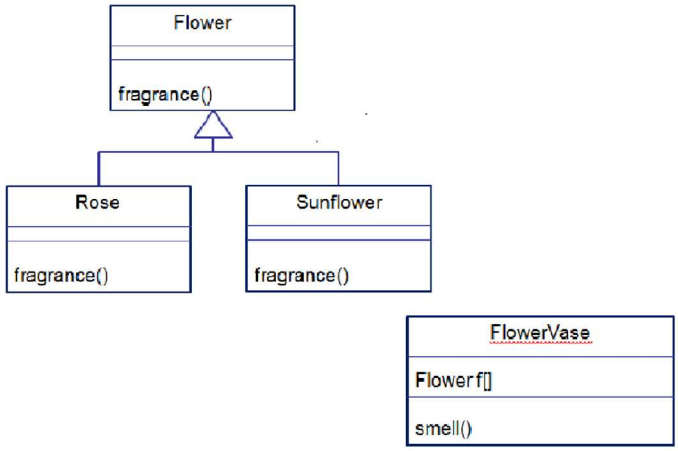
```
class X  {
        X( ) {
                System.out.println("Inside X's Constructor");
        }
}
class Y  extends X {
        Y( ) {
                System.out.println("Inside Y's Constructor");
         }
}
class Z extends Y {
        Z() {
                System.out.println("Inside Z's Constructor");
         }
}
class Demo {
        public static void main(String args[]) {
                Z z = new Z();
        }
}
```

- first line of a constructor must EITHER be a super *(call on the super class constructor)* OR a this *(call on the constructor of same class)*
- If the first statement within a constructor is NEITHER super() NOR this(), then the compiler will automatically insert a super(). (That is, invocation to the super class' no argument constructor)

Polymorphism

A concept in type theory, according to which a name (such as a variable declaration) may denote objects of many different classes that are related by some common super class; thus, any object denoted by this name is able to respond to some common set of operations in different ways.

- Grady Booch



Polymorphism is the ability by which, we **can create functions or reference variables which behaves differently in different programmatic context**.

In java language, polymorphism is essentially considered into two versions.

1. Compile time polymorphism (static binding or method overloading)
2. Runtime polymorphism (dynamic binding or method overriding)

Compile time polymorphism (static binding or method overloading)

This is used to write the program in such a way, that flow of control is decided in compile time itself. It is achieved using method overloading.

In method **overloading**, an object can have two or more methods with same name, BUT, with their method parameters different.

These parameters may be different on the following bases:

1. No of parameters

2. Type of parameters

3. Order of parameters

Exercises:

Create a class called Calculator which has 4 different methods add, diff, mul and div that accept two numbers as parameters. Overload the methods such that the parameters can be of the following pattern.

a) *Both are of int data type.*

b) *Both are of double data type.*

c) *First parameter is of int data type and second parameter is of double data type.*

d) *First parameter is of double data type and second parameter is of int data type*

Create an object to access these methods and invoke these methods with different type of numbers and display the result in the corresponding methods.

Overriding:

- Redefinition of an inherited method declared in the super class by the subclass is called Overriding.
- Rules
    1. The signature of the method (method name + argument list) must exactly match.
    2. The return type must be same or a subtype of the return type of super class method (covariant returns)
    3. The access can be same or be increased.
    4. (List of access specifiers in order of their increasing accessibility: **private→default→protected→public**)
    5. Instance methods can be overridden only if they are inherited/visible by the subclass.
    6. Exception thrown cannot be new exceptions or parent class exception.

e.g.

```
class Animal {
  void makeNoise()
   {
      System.out.println("Some sound");
   }
}
```

```
class Dog extends Animal{
   void makeNoise()
   {
      System.out.println("Bark");
   }
}

class Cat extends Animal{
   void makeNoise()
   {
      System.out.println("Meawoo");
   }
}
public class Demo
{
   public static void main(String[] args) {
      Cat c1 = new Cat();
      c1.makeNoise();

      Dog d1 = new Dog();
      D1.makeNoise();
   }
}
```

Super class Reference Variable

A reference variable of type super class can be assigned a reference to any subclass object derived from that super class.

```
class  A1 {

}

class A2 extends A1 {

}

class A3 {

public static void main(String[] args) {

A1 x;

A2 z = new A2();
```

x = new A2();//valid

z = new A1();//invalid

}

Dynamic Method Dispatch or Runtime Polymorphism

- Method overriding forms the basis of one of Java's most powerful concepts: **dynamic method dispatch**
- Dynamic method dispatch occurs when the Java language resolves a call to an overridden method at runtime, and, in turn, implements runtime polymorphism
- Java makes runtime polymorphism possible in a class hierarchy with the help of two of its features:
  - superclass reference variables
  - overridden methods

Overridden Methods & Runtime Polymorphism - An Example

```
class Figure {
  double dimension1;
  double dimension2;

  Figure(double x, double y){
    dimension1 = x;
    dimension2 = y;
  }

  double area() {
    System.out.println("Area of Figure is
  undefined");
    return 0;
  }
}
```

Overridden Methods & Runtime Polymorphism - An Example (Contd.).

```
class Rectangle extends Figure {
   Rectangle(double x, double y) {
      super(x,y); }
    double area() //method overriding
    {
     System.out.print("Area of rectangle is :");
     return dimension1 * dimension2;
    }
}
class Triangle extends Figure {
  Triangle(double x, double y) {  super(x,y); }
double area()  //method overriding {
   System.out.print("Area for triangle is :");
   return dimension1 * dimension2 / 2;
  }
}
```

Overridden Methods & Runtime Polymorphism - An Example (Contd.).

```
class FindArea {
 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 fig;    //reference variable

   fig = r;
   System.out.println("Area  of  rectangle  is  :"  +
fig.area());
   fig = t;
   System.out.println("Area  of  triangle  is  :"  +
fig.area());
   fig = f;
   System.out.println(fig.area());
 }
}
```

Exercises:

Program 1:
Write a program to create a class named Shape. In this class we have three sub classes Circle, Triangle and Square each class has one member function named draw(). Create these using polymorphism concepts.

Program 2:
Create a super class called Figure that stores the dimensions of various two dimensional objects. Figure defines a method called area( ) that computes the area of an object. The program derives two subclasses from Figure. The two subclasses are Rectangle and Triangle. Each of these subclasses overrides area( ) so that it returns the area of a Rectangle and a Triangle respectively.

Abstract classes & methods

- **Abstract** methods are the methods that don't have the method body. They are just declarations
- The whole class must be declared **abstract**, even if a single method is **abstract.**
- The subclass which extends **abstract** class must provide the definition to the abstract methods.
- Declaring a class as an **abstract** class prevents you from creating instances of that class.
- While an **abstract** class can have abstract methods, it could also NOT have any **abstract** methods.
- A class can inherit from abstract class either by complete or partial Implementation. In the case of partial implementation, the class should be marked **abstract**.

e.g:

```
abstract class Figure {
        double dimension1;
        double dimension2;
        Figure(double x, double y){
                dimension1 = x;
                dimension2 = y;
        }
        abstract double area();
}
class Rectangle extends Figure{
        Rectangle(double x, double y){
                super(x,y);             }
        double area(){
                System.out.print("Area of rectangle is :");
                return dimension1 * dimension2;
        }
}
```

Improved Version of the Figure Class Hierarchy

```
abstract class Figure{
  double dimension1;
  double dimension2;
  Figure(double x, double y){
    dimension1 = x;
    dimension2 = y;
  }
  abstract double area();
}
```

Improved Version of the Figure Class Hierarchy (Contd.)

```
 class Rectangle extends Figure{
   Rectangle(double x, double y){
     super(x,y);     }

   double area(){
     System.out.print("Area of rectangle is :");
     return dimension1 * dimension2;
   }
 }

 class Triangle extends Figure{
   Triangle(double x, double y){   super(x,y); }
   double area(){
     System.out.print("Area for triangle is :");
     return dimension1 * dimension2 / 2;
   }
 }
```

Improved Version of the Figure Class Hierarchy (Contd.).

```
class FindArea{
  public static void main(String args[]){
    Figure fig;
    Rectangle r = new Rectangle(9,5);
    Triangle t  = new Triangle(10,8);
    fig = r;
    System.out.println("Area  of  rectangle  is  :"  +
  fig.area());
    fig = t;
    System.out.println("Area  of  triangle  is  :"  +
  fig.area());
  }
}
```

Use of instance of operator

The instanceof operator in java allows you determine the type of an object

The instanceof operator compares an object with a specified type

It takes an object and a type and returns true if object belongs to that type. It returns false otherwise.

We use instanceof operator to test if an object is an instance of a class, an instance of a subclass, or an instance of a class that implements an interface

instanceof operator example

```
class A {
    int i, j;
}
class B extends A {
    int a, b;
}
class C extends A {
    int m, n;
}
```

instanceof operator example 1

```
class InstanceOfImpl 1{
    public static void main(String args[ ]) {
        A a = new A( );
        B b = new B( );
        C c = new C( );
        A ob=b;
        if (ob instanceof B)
            System.out.println("ob now refers to B");
        else
            System.out.println("Ob is not instance of B");
        if (ob instanceof A)
            System.out.println("ob is also instance of A");
        else
            System.out.println("Ob is not instance of A");
        if (ob instanceof C)
            System.out.println("ob now refers to C");
        else
            System.out.println("Ob is not instance of C");
    }
}
```

ob now refers to B
ob is also instance of A
Ob is not instance of C

instanceof operator example 2

```
class InstanceOfImpl 1{
    public static void main(String args[ ]) {
        A a = new A( );
        B b = new B( );
        C c = new C( );
        A ob=c;
        if (ob instanceof B)
            System.out.println("ob now refers to B");
        else
            System.out.println("Ob is not instance of B");
        if (ob instanceof A)
            System.out.println("ob is also instance of A");
        else
            System.out.println("Ob is not instance of A");
        if (ob instanceof C)
            System.out.println("ob now refers to C");
        else
            System.out.println("Ob is not instance of C");
    }
}
```

Ob is not instance of B
ob is also instance of A
ob now refers to C

Final keyword:

The **final keyword** in java is used to restrict the user.

Final can be:

1. variable
2. method
3. class

final variables:

You cannot change the value of final variable (It will be constant).

```
final int i=10;

i=20; //error
```

Note: In java constant is public static final field.

e.g

```
public static final double INTEREST_RATE=0.25;
```

final methods:

You cannot override final methods.

e.g:

```
class A
{
    final void m1() {
          System.out.println("m1 inside A");
    }
}
class B extends A
{
     void m1() {                  // error, why bcoz m1() is final in side A. So we cannot override.
          System.out.println("m1 inside B");
     }
}
```

final classes:

You cannot extend final classes.

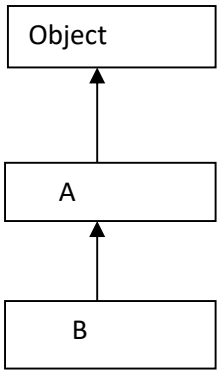
e.g.

```
final class A{

}

class B extends A {   //error

}
```

Object class:

- All classes in java, by default, inherit from a predefined java class called Object.
- Object class is defined in **java.lang** package.
- This class is the root of the class hierarchy.
- **Object** is a super class of all other classes; i.e., Java's own classes(predefined classes), as well as user-defined classes
- This means that a reference variable of type **Object** can refer to an object of any other class
- Object defines the following methods, which means that they are available in every object

```
Class A
{

}

Class B extends A {

}
```



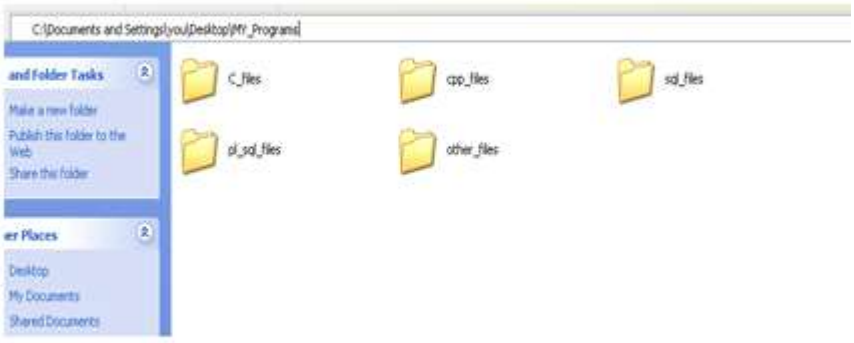
Method	Description
public final Class getClass()	returns the Class class object of this object. The Class class can further be used to get the metadata of this class.
public int hashCode()	returns the hashcode number for this object.
public boolean equals(Object obj)	compares the given object to this 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(long timeout,int nanos)throws InterruptedException	causes the current thread to wait for the specified milliseconds and nanoseconds, 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.

Packages

Package is similar to folders in your Disk

- People normally group their related data in various folders.
  - For example, a programmer may group his programs into the following folders.
  - C programs, CPP programs, SQL queries, PLSQL programs, etc.
- We also use sub-folders for organizing our data more conveniently.
  - The advantage is we can easily locate the files if they are organized.

Package is similar to folders in your Disk (Contd.)



Just relate package concept with directories concept in your file system

- Package is a grouping mechanism that holds a set of classes (or interfaces)
- Packages are implemented using file system directories.
- **Benefits:**
  1. It is easier to work with a set of classes that are grouped in accordance with shared / similar responsibilities.
  2. Easy to locate the classes.
  3. Eliminates name collision
  4. You can restrict the access to your classes.

Creating own package:

- Syntax for creating packages:

  package package_name[.package_name];

  Packages are named like variables except that they are all in lower case and can contain dots (**.**)

  Package must be the first statement in the java source file.

e.g

```
package com;

class A {

public static void main(String[] args)  {

System.out.println("hello");

}

}
```

Where to save?

Create a folder with same name as package name (here create a folder called **com**)

Save this file inside the folder.

e.g

Assume that **E:\myprograms** is location where I want to save all my java programs

Create a folder **com** inside **E:\myprograms** and save the file **A.java** inside **com**.

Compilation:

E:\>myprograms>javac com\A.java

Execution:

E:\>myprograms>java com.A

Need of Packages:

- Packages are containers used to store the classes and interfaces into manageable units of code.
- Packages also help control the accessibility of your classes. This is also called as visibility control.

Access Protection using Packages

- Packages facilitate access-control
- Once a class is packaged, its accessibility is controlled by its package
- That is, whether other classes can access the class in the package depends on the access specifiers used in its class declaration
- There are four visibility control mechanisms packages offer:
  - private
  - no-specifier (default access)
  - protected
  - public

Specifier	**Accessibility**
Private	Accessible in the same class only
Protected	Subclasses and non-subclasses in the same package, and subclasses in other packages

No-specifier (default access)	Subclasses and non-subclasses in the same package
Public	Subclasses and non-subclasses in the same package, as well as subclasses and non-subclasses in other packages. In other words, total visibility

Specifier	Accessibility
private	same class only
protected	same package and subclasses
No-specifier (default access)	same package only
public	Anywhere in the program

How to access class defined inside a package?

- Whenever you need to access a class, you access it through its package by prefixing the class with the package name

  package com.abc;

  public Class A {

```
public Void m1() {

}

}

package com.main;

class Test

{

public static void main(String[] args) {

 // accessing class A

com.abc.A a1=new com.abc.A();

a1.m1();

}

}
```

- Java provides an easy way to access classes in other packages instead of using long names. This is done by using **import** statement.

```
package main;

import com.abc.A;

class Test

{

public static void main(String[] args) {

 // accessing class A

A a1=new A();
```

```
a1.m1();
}
}
```

- If you want to use several classes from a package, it would be cumbersome to type so many classes qualified by their packages.
- It can be made easy by giving a star(*) at the end of the import statement. For example:

```
import  package1.*;
```

e.g.

A.java

```
package com.abc
public class A{
}
```

B.java

```
package com.abc
public Class B {
}
```

Test.java

```
package com.main;
import com.abc.*;
class Test {
public static void main(String[] args)
```

```
{
        A a1=new A();
        B b1=new B();
}
```

Inbuilt Packages

Java API is a collection of predefined packages

Some of the in-built packages

- java.lang
- java.io
- java.util
- java.awt

The default package is **java.lang. (String, System, Integer etc. classes are inside java.lang package)**

No need to import java.lang package. Its automatically added to source file.

Access specifies:

- Specifies the accessibility or visibility of class and class members i.e. specifies where to access and where we cannot accesses class and class members.
- Four access levels:
    1. Public
    2. Private

3. Protected
4. (default)

- Class can have only two access levels
  - Public
  - (default)

In front of class only public or default is allowed.

```
public class A {    // with public access

}
class  A {       // with default access

}
private class A {   // error, only public and default allowed

}
```

- Class members (variables & methods) can have all access levels.

Class Access:

How many ways we can access a class?

1. Using object creation

```
class A  {
  void m1() {

   }
}

class B
{
    void test() {
```

```
    A a=new A();   // class B accessing class A
   }
}
```

2. Using inheritance

```
class A  {

}

class B extends A {   // also class B accessing A

}
```

Class member access:

1. obj.{class_member}
2. through inheritance

Exercises:

Program 1:

Create a package called test package;

Define a class called foundation inside the test package;

Inside the class, you need to define 4 integer variables;

Var1 as private;

Var2 as default;

Var3 as protected;

Var4 as public;

Import this class and packages in another class.

Try to access all 4 variables of the foundation class and see what variables are accessible and what are not accessible.

Program 2:

Create a new project in which create a package named org.animals. In that create various classes like Lion, Monkey, and Elephant. In each class create data members like color, weight and age. Create methods like isVegetarian, canClimb, getSound.

Create a package called zoo and create a class called VandalurZoo and create objects for the animals that are existing in zoo and print the characteristic of each animal.

Interface:

What is interface?

- Like class, interface is also used to define a new type.
- An interface is a special type of construct that may have
    - some constants : **static and final**
    - some methods listed but with no implementations: **abstract**
- All the methods that are declared within an interface are always, by default, **public** and **abstract**

- Any variable declared within an interface is always, by default, **public static** and **final**
- A class implements an interface using the **implements** clause

e. g.

```
 interface MyInterface {
void sayHello();
}
Class MyClass implements MyInterface {
Public void sayHello() {
System.out.println("Hello Interface …..");
}
```

e.g:

```
public interface Shape {
public static final double PI=3.14;
void area();
}
class  Circle implements Shape {
private double radius;
Circle(double r){radius=r;}
public void area(){
        System.out.println(PI* radius* radius);
}
```

```
public static void main(String a[]){
Shape s= new Circle(10); //or Circle c= new Circle (10);
s.area();
System.out.println(Shape.PI+ " "+ Circle.PI);
}
}
```

- Interface is also used to define the contract for the classes to implement.
- An interface defines a protocol of behavior
- An interface lays the specification of what a class is supposed to do
- How the behavior is implemented is the responsibility of each implementing class

Why interfaces required?

- Interfaces allow you to implement common behaviors in different classes that are not related to each other
- Interfaces are used to describe behaviors that are not specific to any particular kind of object, but common to several kind of objects

Rules:

- A class can implement more than one interface at a time.

```
Interface I1
{
Void m1();
}
Interface I2 {
```

```
Void m2();
}
Class MyClass implements I1,I2 {
Public void m1() {
}
Public void m2() {
}
}
```

- A class can extend only one class, but implement many interfaces.

```
Interface I1 {
}
Interface I2 {
}
Class A {
}
Class B extends A implements I1, I2 {

}
```

- An interface can extend another interface.

```
. Interface I1 {
}
Interface I2 extends I1 {
}
```

Marker interface:

- An Interface with no method declared in it, is known as Marker Interface

- Marker Interface is provided as a handle by java interpreter to mark a class, so that it can provide special behavior to it at runtime
- Examples of Marker Interfaces :
  - java.lang.Cloneable
  - java.io.Serializable
  - java.rmi.Remote

Exercises:

Program 1:

Write an interface called Playable, with a method void play ().

Define two classes Guitar and Piano that implements Playable interface and implements play () method.

Define a class TestInstrument with main () method and instantiate both the classes and test the code.

Expected output:

Guitar is playing tin tin tin . . . . .

Piano is playing tan tan tan . . . . .

Program 2:

A scientific application needs to use several constants throughout the application. It was decided to create an interface called PhysicalConstants. The constants that needs to be set are

- Speed of light in vacuum (C):299 792 458 m/s
- Gravitational constant (G): 6.674 28×10^{-11} m^3 kg^{-1} s^{-2}
- Standard Gravitational Acceleration(g) : 9.806 65 m/s2

Use these constants for a class that has following functions.

- E= MC2
- F=G (m1 m2)/r2
- d=0.5 gt

Make sure those constants names are used without repeating interface names with constants (Hint: use static imports).