SDA and OOD LAB

Java Lab Manual # 05

OOP in Java

Instructor: Engr. Khuram Shahzad

Fast National University of Computer and Emerging Sciences, Peshawar
Department of Computer Science & Software Engineering

Table of Contents

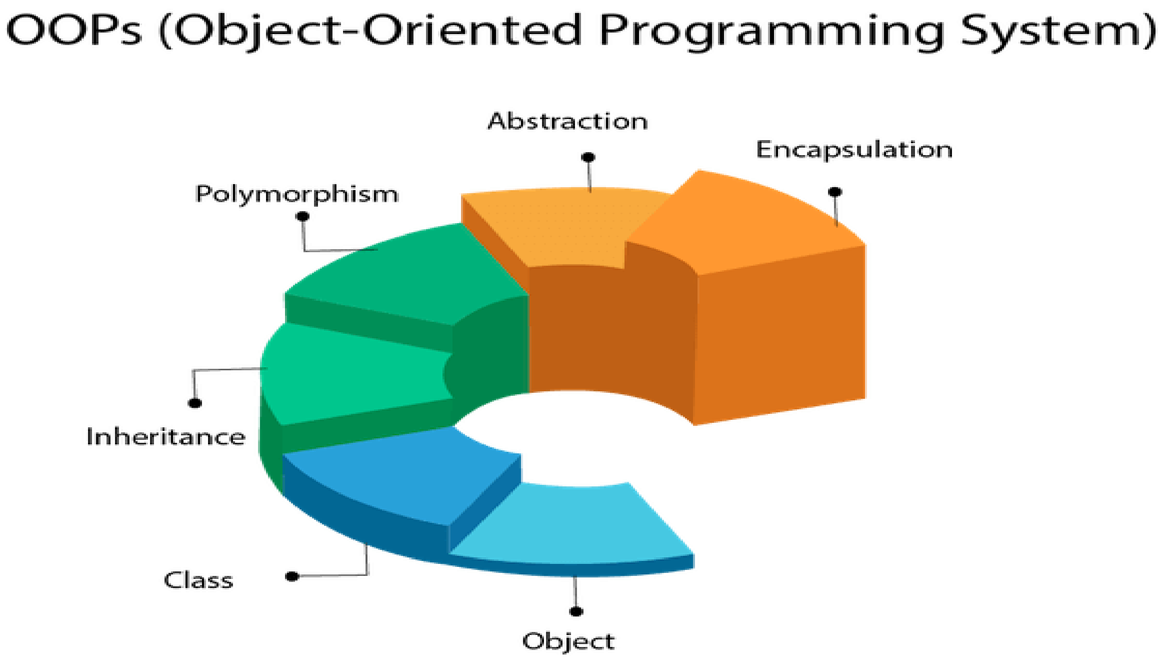
Object Oriented Programming (OOP)

OOP is mythology or paradigm to design a program using class and object.

OOP is paradigm that provides many concepts such as:

- Class and objects
- Inheritance
- Modularity
- Polymorphism
- Encapsulation (binding code and its data) etc.
- Paradigm نمونہ

OOP is used to reduce complexity. To divide large and complex program into chunks and modules.



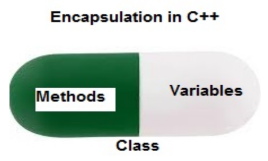
Why OOP?

Real world implementation.

Encapsulation

- Data and behaviour (function) are tightly coupled inside an object.
- Combine data (variables) and functions in a single container.

- The combining of both data and functions into a single unit.
- To combine code functions and data (variables) in a single box or wrapper.
- In java class is the example of encapsulation.
- **Capsule:** It is wrapped with different medicines.



Capsule

Data Hiding

- Means you cannot access data.
- Making data to be accessed from within the class.

Access Specifier

- It specifies that member of a class is accessible outside or not.
- It may be public, protected or private or default.

Class

- Class is blue print or map for object.
- Class is the logical construct of object.
- Class is the description of object.
- Class is a template which contains behaviour (member functions) and attributes/properties (data/variables) of object.
- Means data members and member functions are defined within a class.
- Class is user defined data type because user defined it (non primitive data type).
- **Attribute:** Properties abject has.
- **Methods:** Actions that an object can perform.

Object

- An entity that has state and behaviour.
- An actual existence of a class is called object.
- Object encapsulates data and behaviour.
- When a class template is implemented in real world then it becomes an object.
- Object is the instance of the class.
- Class is the template or blue print from which objects are created, so object is the **instance (result) of the class.**
- The space reserved in memory for class is called an object. Instance of a class.
- (Instance---→ Single occurrence)
- Object is used to perform responsibility of communication between different classes.

- Any entity that has state and behavior is known as an object. For example, a chair, pen, table, keyboard, bike, etc. It can be physical or logical.
- **Example:** A dog is an object because it has states like color, name, breed, etc. as well as behaviors like wagging the tail, barking, eating, etc.

Class VS Object

- **Class:** No data
- **Object:** Having Data

Member data and member function

Member data or data members: The data or the attributes defined within a class is called member data.

Member Function: The functions that are used to work on the data items are called member functions.

Member functions are used to process and access data members of an object.

Member functions are functions that are included within the class.

Inheritance

❖ The mechanism in which one class acquires all the properties and behavior of another class is called inheritance.

❖ Including features of one class into another class is called inheritance.

❖ *When one object acquires all the properties and behaviors of a parent object*, it is known as inheritance.

❖ It provides code reusability.

❖ It is used to achieve runtime polymorphism.

Polymorphism

- When one task is performed by different ways.
- Poly means many and morphs means shapes.
- **E.g.** to convince customer differently, to draw something e.g. shape or rectangle.
- Another example can be to sound/speak something i.e. cat speaks meaw, dog barks woof etc.
- **Note:** In java we use method overloading and method overriding to achieve polymorphism.
- It may be compile time polymorphism and run time polymorphism.

"Now Speak!"
"Quack"
"Woof"
"Meow"

Abstraction

- Hiding internal details and showing functionality is known as abstraction.
- **For example,** phone call, we do not know the internal processing.
- **Note:** In java we use abstract class and interface to achieve abstraction.

Instance variable in java

- A variable that is created inside a class but outside a method is called instance variable.
- Instance variable does not get memory at compile time.
- It gets memory at runtime when an object (instance) is created. That is why it called instance variable.

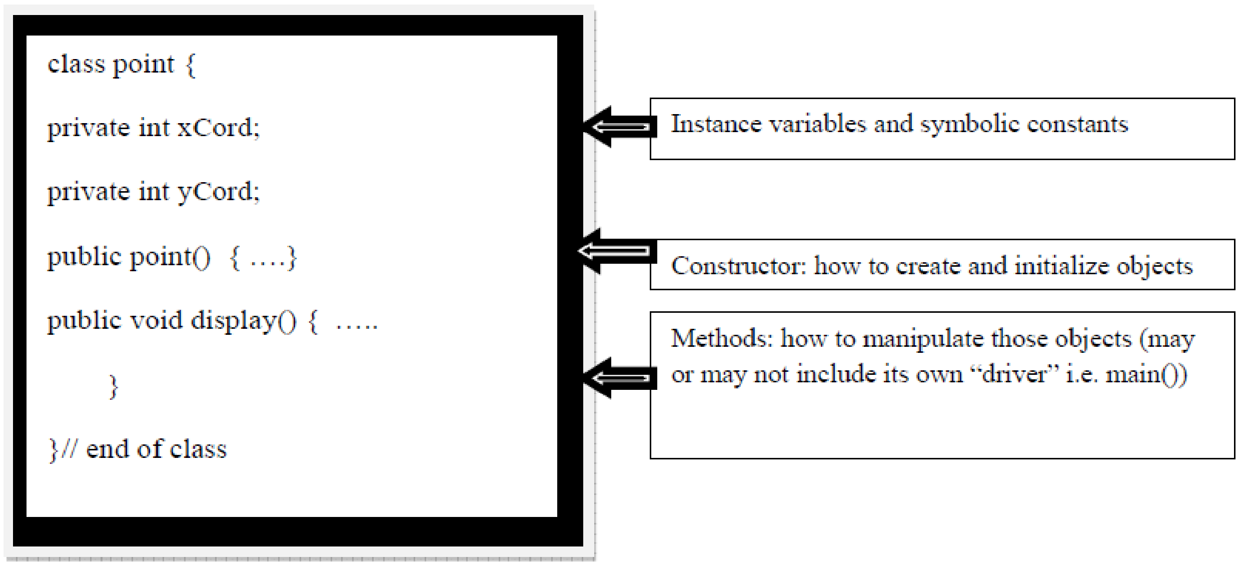
Static Data Members

Is shared by all objects of a class.

Data Members

- ❖ Are variables of any primitive data types.
- ❖ It can be any type of array.

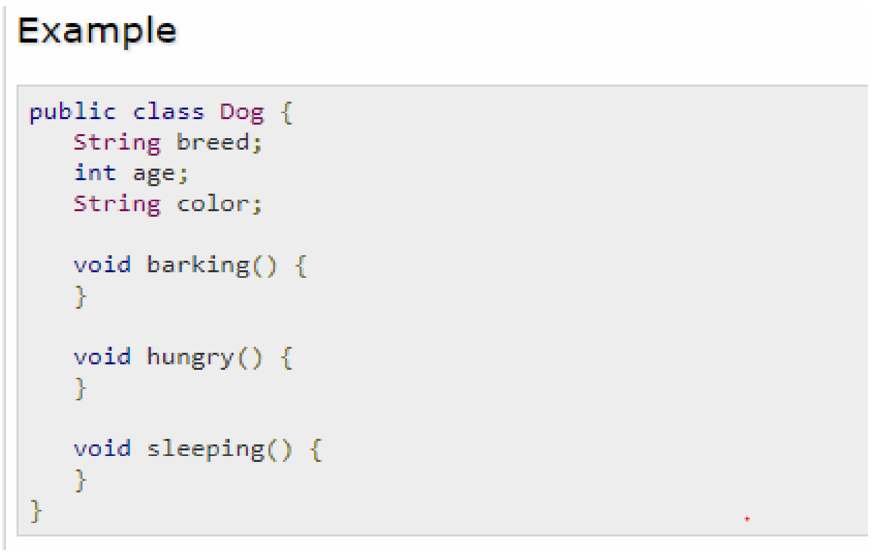
Defining a Class



```
class point {
private int xCord;
private int yCord;
public point()  { ….}
public void display() {  …..
        }
}// end of class
```

Instance variables and symbolic constants
Constructor: how to create and initialize objects
Methods: how to manipulate those objects (may or may not include its own "driver" i.e. main())

Classes and Objects



Example

```
public class Dog {
   String breed;
   int age;
   String color;

   void barking() {
   }

   void hungry() {
   }

   void sleeping() {
   }
}
```

Object declaration

Student obj ; // obj is reference variable

Object instantiation

obj = new Student();

OR

Student obj = new Student();

How to Access member of a class

1) obj.variableName = value; // to access data member

2) obj.methodName(parameter list); // to access method

Examples

obj.rollNo = 123;

obj.getRollNo();

```
//Java Program to illustrate how to define a class and fields
//Defining a Student class.
class Student{
 //defining fields
int id;       //field or data member or instance variable
String name;
 //creating main method inside the Student class
 public static void main(String args[]){
  //Creating an object or instance
  Student s1=new Student();  //creating an object of Student
  //Printing values of the object
  System.out.println(s1.id);
//accessing member through reference variable
 System.out.println(s1.name);
 }
}
```

Output:

```
0
null
```

Ways to initialize object

There are 3 ways to initialize object in Java.

1. By reference variable
2. By method
3. By constructor

1) Initialization through reference

Initializing an object means storing data into the object. Let's see a simple example where we are going to initialize the object through a reference variable.

Driver Class: Class which contains main method is called main Class or driver class.

```
class Student{
 int id;
 String name;
}
// driver class
class TestStudent2 {
public static void main(String args[]){
  Student s1=new Student();
  s1.id=101;
  s1.name="Sonoo";
  System.out.println(s1.id+" "+s1.name); //printing members with a white space
 } }
```

Output

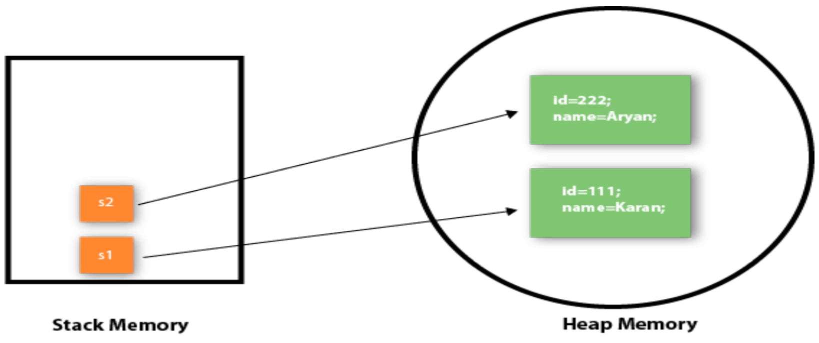
```
101 Sonoo
```

2) Initialization through method

In this example, we are creating the two objects of Student class and initializing the value to these objects by invoking the insertRecord method. Here, we are displaying the state (data) of the objects by invoking the displayInformation() method.

```
 class Student{
        int rollno;
        String name;
        void insertRecord(int r, String n){
         rollno=r;
         name=n;
        }
        void displayInformation(){System.out.println(rollno+" "+name);}
    }
class TestStudent4{
        public static void main(String args[]){
         Student s1=new Student();
         Student s2=new Student();
         s1.insertRecord(111,"Karan");
         s2.insertRecord(222,"Aryan");
         s1.displayInformation();
         s2.displayInformation();
        }
       }
```

As you can see in the below figure, object gets the memory in heap memory area. The reference variable refers to the object allocated in the heap memory area. Here, s1 and s2 both are reference variables that refer to the objects allocated in memory.



3) Initialization through a constructor

```
    class Student4{
    int id;
    String name;
    //creating a parameterized constructor
    Student4(int i,String n){
    id = i;
    name = n;
    }
    //method to display the values
    void display(){System.out.println(id+" "+name);}

    public static void main(String args[]){
    //creating objects and passing values
    Student4 s1 = new Student4(111,"Karan");
    Student4 s2 = new Student4(222,"Aryan");
    //calling method to display the values of object
    s1.display();
    s2.display();
   }
}
```

Output:

```
111 Karan
 222 Aryan
```

Anonymous object

- Anonymous simply means nameless. An object which has no reference is known as an anonymous object. It can be used at the time of object creation only.
- If you have to use an object only once, an anonymous object is a good approach.
- For example:

```
new Calculation();   //anonymous object
```

Calling method through a reference:

```
Calculation c=new Calculation();

c.fact(5);
```

Calling method through an anonymous object

```
new Calculation().fact(5);
```

```
//Let's see the full example of an anonymous object in Java.

class Calculation{
 void fact(int  n){
  int fact=1;
  for(int i=1;i<=n;i++){
   fact=fact*i;
  }
 System.out.println("factorial is "+fact);
}
public static void main(String args[]){
 new Calculation().fact(5);
//calling method with anonymous object
}
}
```

Access Specifier

- It specifies that member of a class is accessible outside or not. It may be public, protected or private or default.
- A **private** member is *only* accessible within the same class as it is declared.
- A member with **no access modifier (Default)** is only accessible within classes in the same package.
- A **protected** member is accessible within all classes in the same package *and* within subclasses in other packages.
- A **public** member is accessible to all classes (unless it resides in a module that does not export the package it is declared in).

Private: Limited access to class only

Default (no modifier): Limited access to class and package

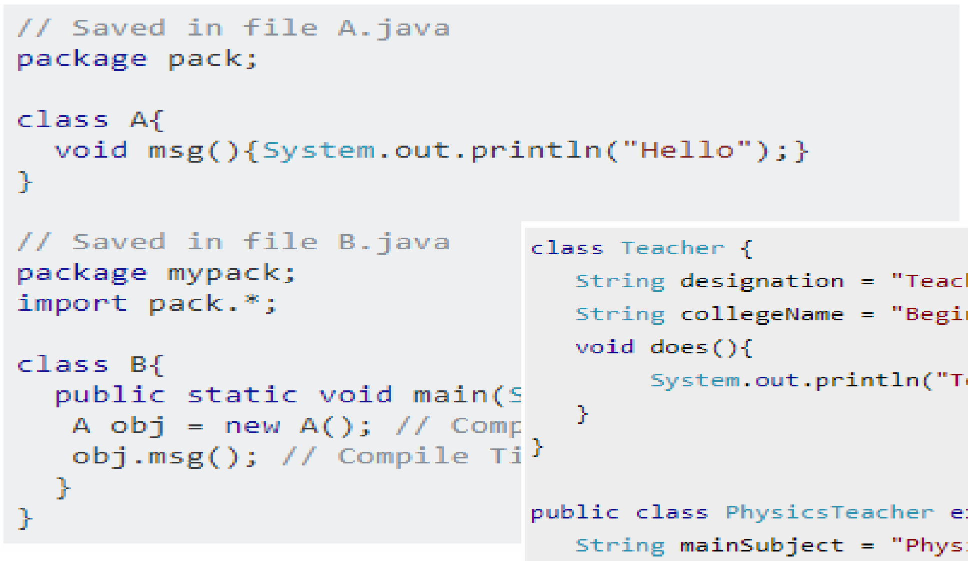
Protected: Limited access to class, package and subclasses (both inside and outside package)

Public: Accessible to class, package (all), and subclasses

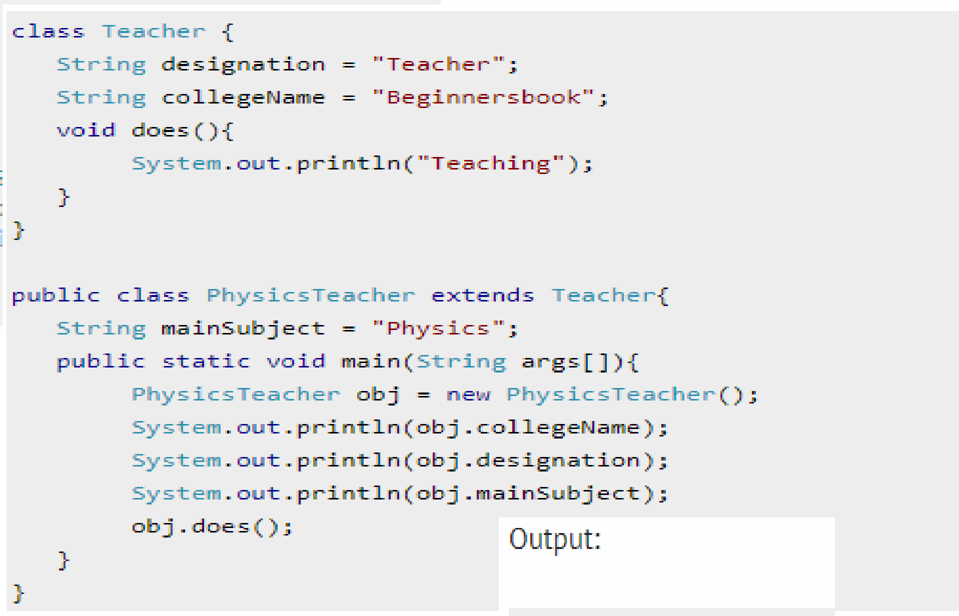
Access Specifier Table

Access Modifier	within class	within package	outside package by subclass only	outside package
Private	Y	N	N	N
Default	Y	Y	N	N
Protected	Y	Y	Y	N
Public	Y	Y	Y	Y

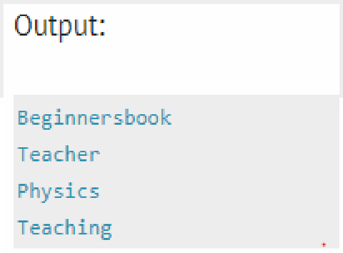
Access Specifier… (default)



```
// Saved in file A.java
package pack;

class A{
  void msg(){System.out.println("Hello");}
}

// Saved in file B.java
package mypack;
import pack.*;

class B{
  public static void main(S
   A obj = new A(); // Comp
   obj.msg(); // Compile Ti
  }
}
```



```
class Teacher {
   String designation = "Teacher";
   String collegeName = "Beginnersbook";
   void does(){
        System.out.println("Teaching");
   }
}

public class PhysicsTeacher extends Teacher{
   String mainSubject = "Physics";
   public static void main(String args[]){
        PhysicsTeacher obj = new PhysicsTeacher();
        System.out.println(obj.collegeName);
        System.out.println(obj.designation);
        System.out.println(obj.mainSubject);
        obj.does();
   }
}
```



Output:

```
Beginnersbook
Teacher
Physics
Teaching
```

Program 01: Data member and member functions

```
	package classes_objects_constructors;	// pakage name
	public class Student {
	int rollNo;
	String name;
	public void setRollNo(int rno)
	{
```

```
rollNo = rno;
}
public int getRollNo()
{
return rollNo;
}
public void setName(String n)
{
name = n;
}
public String getName()
{
return name;
}
public String toString()
{
return "Roll No is:" +rollNo + "\n" +"Name is:" +name;
}
}         // Student Class body closed
```

Main Class: Class which contains main method is called main class or driver class.

```
package classes_objects_constructors;
public class StudentTest {
public static void main(String[] args) {
Student s= new Student();
System.out.println(s.getRollNo());      // This will show "0"
because no argument is passed
System.out.println(s.getName());      // This will show "null"
because no argument is passed
System.out.println("\nAfter Passing Arguments to Functions\n");
s.setRollNo(150535);
s.setName("Rizwan Ullah");

System.out.println("Roll Number is: "+s.getRollNo());     // This
will show "150535" because argument is passed
System.out.println("Name is: " +s.getName());   // This will show
"Rizwan Ullah" because argument is passed
/*System.out.println(s);   /*this will just print object
reference because "tostring()" function is not defined in student
class  */
System.out.println("\nAfter defininig toString() Function in
Student Class\n");
System.out.println(s);                  //this will call toString
function
//System.out.println(s.toString());    // This will also call
toString() Function
System.out.println("");
Student s2 = new Student();
s2.setRollNo(150509);
s2.setName("Sami Ullah");
System.out.println(s2);
}
}    // StudentTest class body closed
```

Note

- ❖ When we write it in main class “ System.out.print(s); ”

  This will print reference of an object “s” on screen because toString() function is not defined in student class, means in that class whose object has been created.

- ❖ After defining function :

  public String toString()

  {

  return (which thing do you want to return);

  }

System.out.println(s);

It will call toString() function and will display RollNo and name of object s.

Means it will display member data of object.

System.out.print(s.toString()); This will also call toString function.

toString() function

- Doing object representation.
- Used for combo boxes.

Constructor

- Special method that is implicitly invoked.
- Used to create an object (an instance of the class) and initialize it.
- Every time an object is created using the new() keyword, at least one constructor is called.
- It is special member function having same name as class name and is used to initialize object.
- It is invoked/called at the time of object creation.
- It constructs value i.e. provide data for the object that is why it called constructor.
- Can have parameter list or argument list.
- Can never return any value (no even void).
- Normally declared as public.
- At the time of calling constructor, memory for the object is allocated in the memory.
- It calls a default constructor if there is no constructor available in the class. In such case, Java compiler provides a default constructor by default.
- **Note:** It is called constructor because it constructs the values at the time of object creation. It is not necessary to write a constructor for a class. It is because java compiler creates a default constructor if your class doesn't have any.

Rules for creating Java constructor

There are some rules defined for the constructor.

- Constructor name must be the same as its class name
- A Constructor must have no explicit return type.
- A Java constructor cannot be abstract, static, final, and synchronized.

Type of Java constructor

There are two types of constructors in Java:

1. Default constructor (no-arg constructor)

2. Parameterized constructor

1) Java Default Constructor

- A constructor is called "Default Constructor" when it doesn't have any parameter.
- It is also called non-parameterized constructor.

Syntax of default constructor:

Access Specifier className()

{

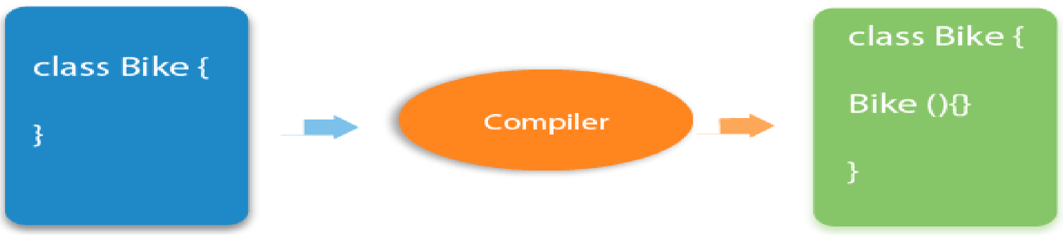
}

Java Default Constructor Example

In this example, we are creating the no-arg constructor in the Bike class. It will be invoked at the time of object creation.

```
//Java Program to create and call a default constructor
class Bike1{
Bike1()                  //creating a default constructor
{
System.out.println("Bike class object created");
}  //constructor body
public static void main(String args[]){        //main method
//calling a default constructor
Bike1 b=new Bike1();
}

}
```

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



What is the purpose of a default constructor?

The default constructor is used to provide the default values to the object like 0, null, etc., depending on the type.

Example of default constructor that displays the default values

```
//Let us see another example of default constructor  which displays th
e default values
class Student3{
int id;
String name;
//method to display the value of id and name
void display(){
System.out.println(id+" "+name);
}

public static void main(String args[]){
//creating objects
Student3 s1=new Student3();
Student3 s2=new Student3();
//displaying values of the object
s1.display();
s2.display();
}
}  // Student3 class body closed
```

Output:

```
0 null

0 null
```

Explanation:

- In the above class, you are not creating any constructor so compiler provides you a default constructor.
- Here 0 and null values are provided by default constructor.

2) Java Parameterized Constructor

A constructor which has a specific number of parameters is called a parameterized constructor.

Why use the parameterized constructor?

- The parameterized constructor is used to provide different values to distinct objects. However, you can provide the same values also.
- Used to initialize objects with different values.

Example of parameterized constructor

In this example, we have created the constructor of Student class that have two parameters. We can have any number of parameters in the constructor.

```
class Student4{
    int id;
    String name;
    //creating a parameterized constructor
    Student4(int i,String n){
    id = i;
    name = n;
    }
    //method to display the values
    void display(){System.out.println(id+" "+name);}

    public static void main(String args[]){
    //creating objects and passing values
    Student4 s1 = new Student4(111,"Karan");
    Student4 s2 = new Student4(222,"Aryan");
    //calling method to display the values of object
    s1.display();
    s2.display();
   }
}  // Student4 class body closed

/*
Output
111 Karan
222 Aryan
*/
```

Constructor Overloading

❖ In Java, a constructor is just like a method but without return type. It can also be overloaded like Java methods.

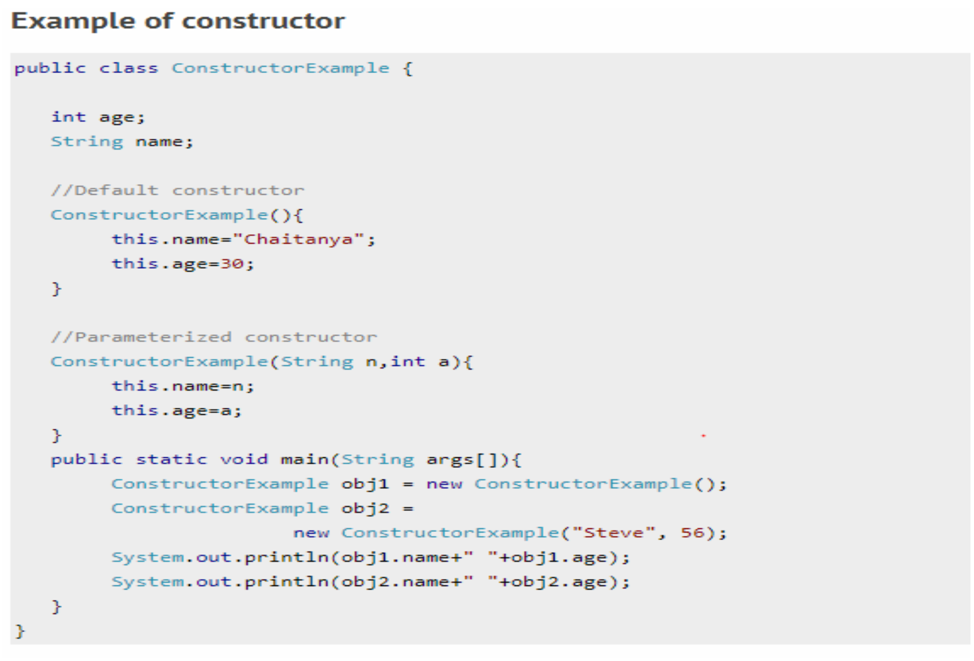
❖ Constructor overloading in Java is a technique of having more than one constructor with different parameter lists.

❖ They are arranged in a way that each constructor performs a different task. They are differentiated by the compiler by the number of parameters in the list and their types.

❖ When we have more than one constructors in a class with different set of parameters i.e. (type, number, order).

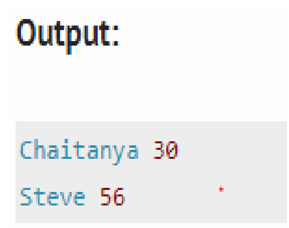
Example of Constructor Overloading

```
//Java program to overload constructors
class Student5{
    int id;
    String name;
    int age;
    //creating two argument constructor
    Student5(int i,String n){
    id = i;
    name = n;
    }
//creating three arg constructor
    Student5(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[]){
    Student5 s1 = new Student5(111,"Karan");
    Student5 s2 = new Student5(222,"Aryan",25);
    s1.display();
    s2.display();
   }
}

/*
Output:
111 Karan 0
222 Aryan 25
*/
```



Example of constructor

```
public class ConstructorExample {

   int age;
   String name;

   //Default constructor
   ConstructorExample(){
        this.name="Chaitanya";
        this.age=30;
   }

   //Parameterized constructor
   ConstructorExample(String n,int a){
        this.name=n;
        this.age=a;
   }
   public static void main(String args[]){
        ConstructorExample obj1 = new ConstructorExample();
        ConstructorExample obj2 =
                       new ConstructorExample("Steve", 56);
        System.out.println(obj1.name+" "+obj1.age);
        System.out.println(obj2.name+" "+obj2.age);
   }
}
```



Output:

```
Chaitanya 30
Steve 56
```

Destructor

Destructors are not required in java class because memory management is the responsibility of JVM.

Copy Constructor

❖ A **copy constructor** in a **Java** class is a **constructor** that creates an object using another object of the same **Java** class.

❖ That's helpful when we want to **copy** a complex object that has several fields, or when we want to make a deep **copy** of an existing object.

❖ There is no copy constructor in Java. However, we can copy the values from one object to another like copy constructor in C++.

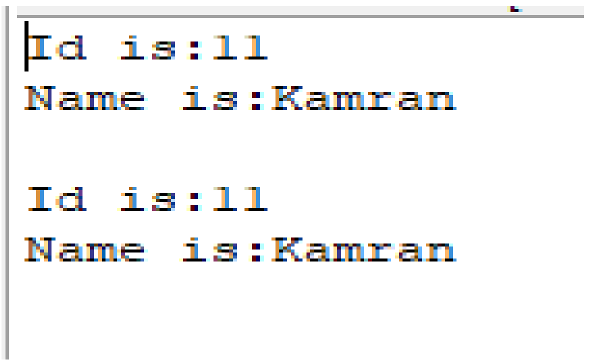
❖ There are many ways to copy the values of one object into another in Java. They are:

1. By constructor
2. By assigning the values of one object into another
3. By clone() method of Object class

❖ In this example, we are going to copy the values of one object into another using Java constructor.

```
// This program is about copy constructor and will copy values of
one object into another
package classes_objects_constructors;
```

```
public class Student6
{
 int id;
 String name;
 Student6(int i,String n)
 {
 id = i;
 name = n;
 }
Student6(Student6 s)         // copy constructor
 {
    id = s.id;
    name =s.name;
 }
 void display()
 {
 System.out.println("Id is:" + id +"\nName is:" + name);
 }
public static void main(String args[])
{
Student6 s1 = new Student6(11,"Kamran");
Student6 s2 = new Student6(s1);  // Values of s1 will be copied
to s2
    s1.display();
    System.out.println("");
    s2.display();
}
}   // Student6 class body closed
```



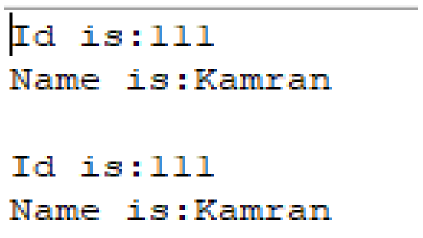
```
Id is:11
Name is:Kamran

Id is:11
Name is:Kamran
```

Copy Values without constructor

- We can copy the values of one object into another by assigning the objects values to another object.
- In this case, there is no need to create the constructor.

Example

```
package classes_objects_constructors;
public class Student7
{
int id;
String name;
Student7(int i,String n)
 {
      id = i;
      name = n;
 }
Student7()      // default constructor
{      }
 void display()
 {
 System.out.println("Id is:" + id +"\nName is:" + name);
 }
public static void main(String[] args)
{
Student7 s1 = new Student7(111,"Kamran");
Student7 s2 = new Student7();
s2.id=s1.id;
s2.name=s1.name;

s1.display();
 System.out.println("");
s2.display();
  }
}
```



```
Id is:111
Name is:Kamran

Id is:111
Name is:Kamran
```

Defining a Student class

The following example will illustrate how to write a class. We want to write a “Student” class that

• Should be able to store the following characteristics of student

Roll No

Name

• Provide default, parameterized and copy constructors

• Provide standard getters/setters (discuss shortly) for instance variables

• Make sure, roll no has never assigned a negative value i.e. ensuring the correct state of the object

• Provide print method capable of printing student object on console

Getters / Setters

- The attributes of a class are generally taken as private or protected. So to access them outside of a class, a convention is followed knows as getters & setters.
- These are generally public methods.
- The words *set* and *get* are used prior to the name of an attribute.
- Another important purpose for writing getter & setters to control the values assigned to an attribute.

Student Class Code

```
// File Student.java
public class Student {
private String name;
private int rollNo;
// Standard Setters
public void setName (String name) {
this.name = name;
}
// Note the masking of class level variable rollNo
public void setRollNo (int rollNo)
{
if (rollNo > 0)
{
   this.rollNo = rollNo;
}else {
```

```
            this.rollNo = 100;  }
      }
      // Standard Getters
      public String getName ( ) {
      return name;
      }
      public int getRollNo ( ) {
      return rollNo;
      }
      // Default Constructor
      public Student() {
      name = “not set”;
      rollNo = 100;
      }
      // parameterized Constructor for a new student
      public Student(String name, int rollNo) {
      setName(name);        //call to setter of name
      setRollNo(rollNo);     //call to setter of rollNo
      }
      // Copy Constructor for a new student
      public Student(Student s) {
      name = s.name;
      rollNo = s.rollNo;
      }
      // method used to display method on console
      public void print () {
      System.out.print("Student name: " +name);
      System.out.println(", roll no: " +rollNo); }
      } // end of Student class
```

Using a Class

Objects of a class are always created on heap using the “new” operator followed by constructor

• Student s = new Student ( ); // no pointer operator “*” between Student and s

• Only String constant is an exception

String greet = “Hello” ; // No new operator

However you can also use

• String greet2 = new String(“Hello”);

Members of a class (member variables and methods also known as instance

variables/methods) are accessed using "." operator. There is no "–>" operator in java

• s.setName("Ali");

• s–>setName("Ali") // is incorrect and will not compile in java

Note: Objects are always passed by reference and primitives are always passed by value in java.

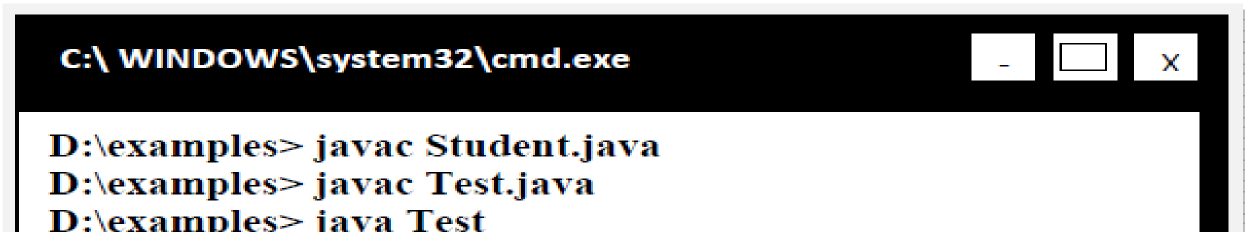
• Create objects of student class by calling default parameterize and copy constructor.

• Call student class various methods on these objects.

Student client code

```
// File Test.java
/* This class create Student class objects and demonstrates how
to call various methods on objects
*/
public class Test{
public static void main (String args[]) {
// Make two student objects
Student s1 = new Student("ali", 15);
Student s2 = new Student();     //call to default constructor
s1.print();     // display ali and 15
s2.print();     // display not set and 100
s2.setName("usman");
s2.setRollNo(20);
System.out.print("Student name:" + s2.getName());
System.out.println(" rollNo:" + s2.getRollNo());
System.out.println("calling copy constructor");
Student s3 = new Student(s2);   //call to copy constructor
s2.print();
s3.print();
s3.setRollNo(-10);    //Roll No of s3 would be set to 100
s3.print();
}  //end of main
}  //end of class


/*NOTE: public vs. private
A statement like "b.rollNo = 10;" will not compile in a client of
the Student class when rollNo is declared protected or private */
```

In



```
D:\examples> javac Student.java
D:\examples> javac Test.java
D:\examples> java Test
```

Inheritance in Java

1. Inheritance
2. Types of Inheritance
3. Why multiple inheritance is not possible in Java in case of class?

Inheritance in Java is a mechanism in which one object acquires all the properties and behaviors of a parent object. It is an important part of OOPs (Object Oriented programming system).

The idea behind inheritance in Java is that you can create new classes that are built upon existing classes. When you inherit from an existing class, you can reuse methods and fields of the parent class. Moreover, you can add new methods and fields in your current class also.

Inheritance represents the **IS-A relationship** which is also known as a *parent-child* relationship.

Why use inheritance in java

- For Method Overriding (so runtime polymorphism can be achieved).
- For Code Reusability.
- Abstraction achieved.

Terms used in Inheritance

- **Class:** A class is a group of objects which have common properties. It is a template or blueprint from which objects are created.
- **Sub Class/Child Class:** Subclass is a class which inherits the other class. It is also called a derived class, extended class, or child class.
- **Super Class/Parent Class:** Superclass is the class from where a subclass inherits the features. It is also called a base class or a parent class.
- **Reusability:** As the name specifies, reusability is a mechanism which facilitates you to reuse the fields and methods of the existing class when you create a new class. You can use the same fields and methods already defined in the previous class.

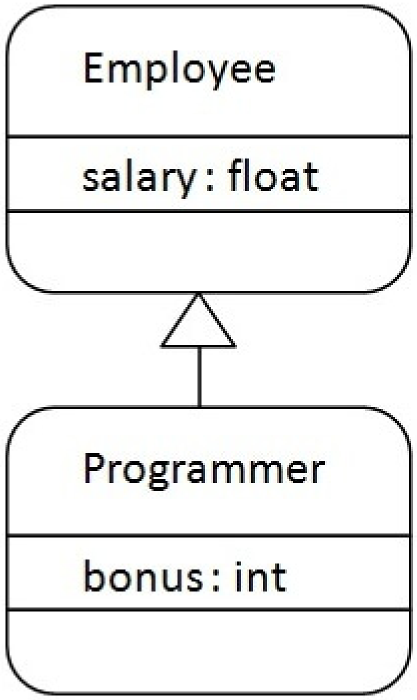
The syntax of Java Inheritance

```
class Subclass-name extends Superclass-name
{
   //methods and fields
}
```

The **extends keyword** indicates that you are making a new class that derives from an existing class. The meaning of "extends" is to increase the functionality.

In the terminology of Java, a class which is inherited is called a parent or superclass, and the new class is called child or subclass.

Java Inheritance Example



As displayed in the above figure, Programmer is the subclass and Employee is the superclass. The relationship between the two classes is **Programmer IS-A Employee**. It means that Programmer is a type of Employee.

```
class Employee{
 float salary=40000;
}
class Programmer extends Employee{
 int bonus=10000;
 public static void main(String args[]){
   Programmer p=new Programmer();
   System.out.println("Programmer salary is:"+p.salary);
   System.out.println("Bonus of Programmer is:"+p.bonus);
}
}
```

Test it Now

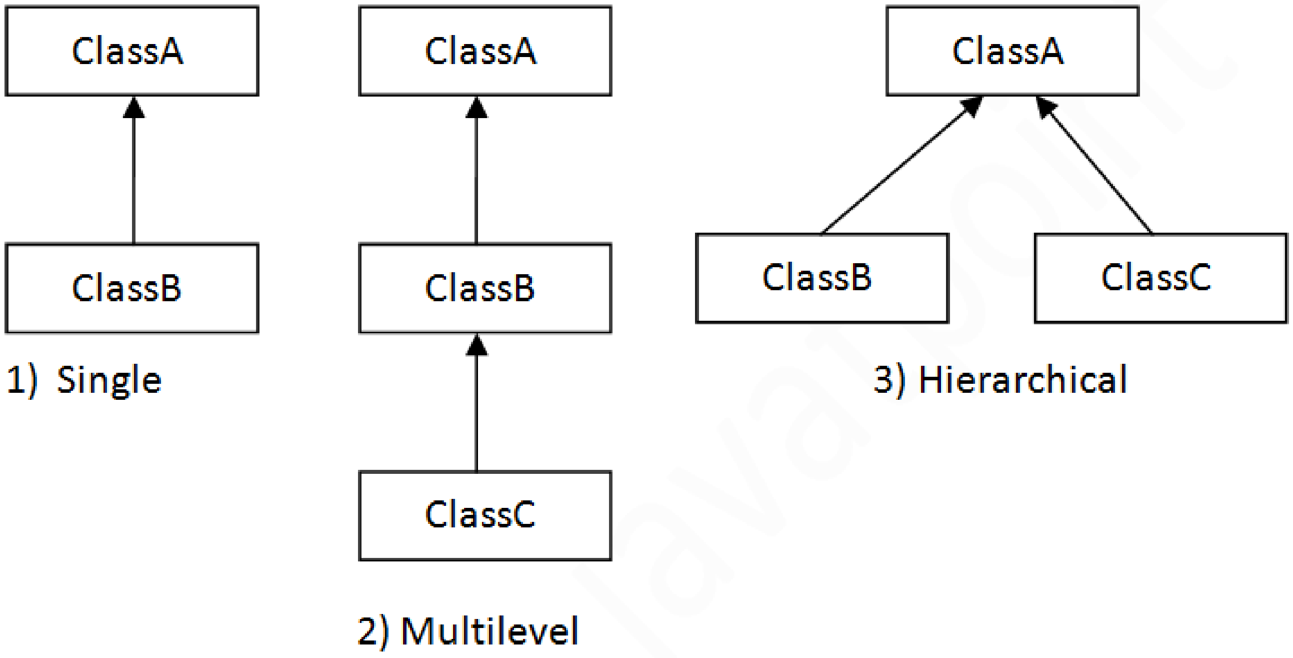
```
Programmer salary is:40000.0
Bonus of programmer is:10000
```

In the above example, Programmer object can access the field of own class as well as of Employee class i.e. code reusability.

Types of inheritance in java

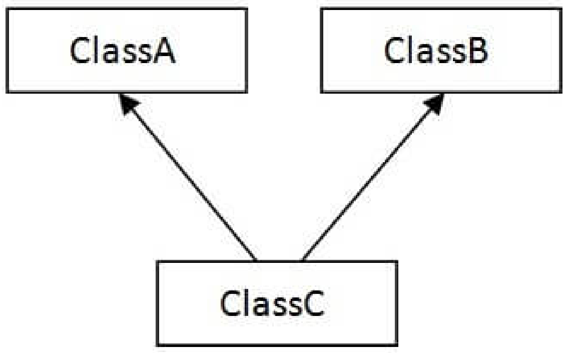
On the basis of class, there can be three types of inheritance in java: single, multilevel and hierarchical.

In java programming, multiple and hybrid inheritance is supported through interface only. We will learn about interfaces later.

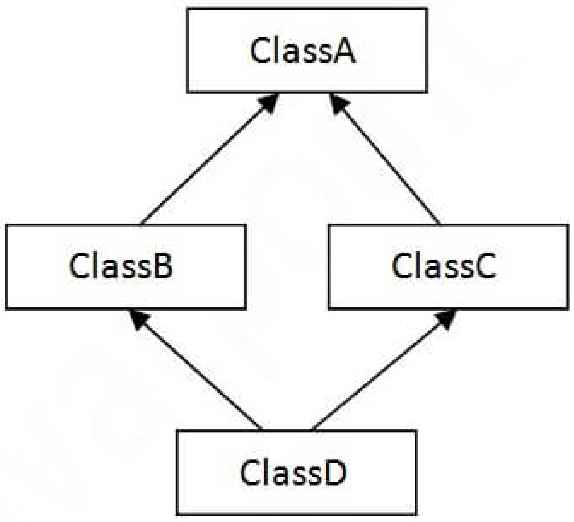


Note: Multiple inheritance is not supported in Java through class.

When one class inherits multiple classes, it is known as multiple inheritance. For Example:



4) Multiple



5) Hybrid

Single Inheritance Example

When a class inherits another class, it is known as a *single inheritance*. In the example given below, Dog class inherits the Animal class, so there is the single inheritance.

File: TestInheritance.java

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

Output:

```
barking...
eating...
```

Multilevel Inheritance Example

When there is a chain of inheritance, it is known as *multilevel inheritance*. As you can see in the example given below, BabyDog class inherits the Dog class which again inherits the Animal class, so there is a multilevel inheritance.

File: TestInheritance2.java

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

Output:

```
weeping...
barking...
eating...
```

Hierarchical Inheritance Example

When two or more classes inherits a single class, it is known as *hierarchical inheritance*. In the example given below, Dog and Cat classes inherits the Animal class, so there is hierarchical inheritance.

File: TestInheritance3.java

```java
class Animal{
void eat(){System.out.println("eating...");}
}
class Dog extends Animal{
void bark(){System.out.println("barking...");}
}
class Cat extends Animal{
void meow(){System.out.println("meowing...");}
}
class TestInheritance3{
public static void main(String args[]){
Cat c=new Cat();
c.meow();
c.eat();
//c.bark();//C.T.Error
}}
```

Output:

```
meowing...
eating...
```

Q) Why multiple inheritance is not supported in java?

To reduce the complexity and simplify the language, multiple inheritance is not supported in java.

Consider a scenario where A, B, and C are three classes. The C class inherits A and B classes. If A and B classes have the same method and you call it from child class object, there will be ambiguity to call the method of A or B class.

Since compile-time errors are better than runtime errors, Java renders compile-time error if you inherit 2 classes. So whether you have same method or different, there will be compile time error.

```java
class A{
void msg(){System.out.println("Hello");}
}
class B{
void msg(){System.out.println("Welcome");}
}
class C extends A,B{//suppose if it were

 public static void main(String args[]){
  C obj=new C();
  obj.msg();//Now which msg() method would be invoked?
}
}
```

Java Polymorphism

Polymorphism means "many forms", and it occurs when we have many classes that are related to each other by inheritance.

Like we specified in the previous chapter; **Inheritance** lets us inherit attributes and methods from another class. **Polymorphism** uses those methods to perform different tasks. This allows us to perform a single action in different ways.

For example, think of a superclass called `Animal` that has a method called `animalSound()`. Subclasses of Animals could be Pigs, Cats, Dogs, Birds - And they also have their own implementation of an animal sound (the pig oinks, and the cat meows, etc.):

Why @override is used in Java?

If a class inherits a method from its superclass, then there is a chance to override the method provided that it is not marked final. The benefit of overriding is: **ability to define a behavior that's specific to the subclass type**, which means a subclass can implement a parent class method based on its requirement.

Is @override necessary?

It is not necessary, but it is highly recommended. It keeps you from shooting yourself in the foot. It helps prevent the case when you write a function that you think overrides another one but you misspelled something and you get completely unexpected behavior.

Compile-Time Polymorphism vs. Run-Time Polymorphism

Compile-Time Polymorphism	Run-Time Polymorphism
• The method call is handled by the compiler	• The compiler cannot control the method call in run-time
• Compile-Time Polymorphism is less flexible, as it needs to handle all method calls in compile-time	• Run-Time Polymorphism exhibits higher flexibility as the method calls get handled at run-time
• Integrating the right method call with the proper method is done in compile-time	• Combining the correct method call with the right method is done in run-time

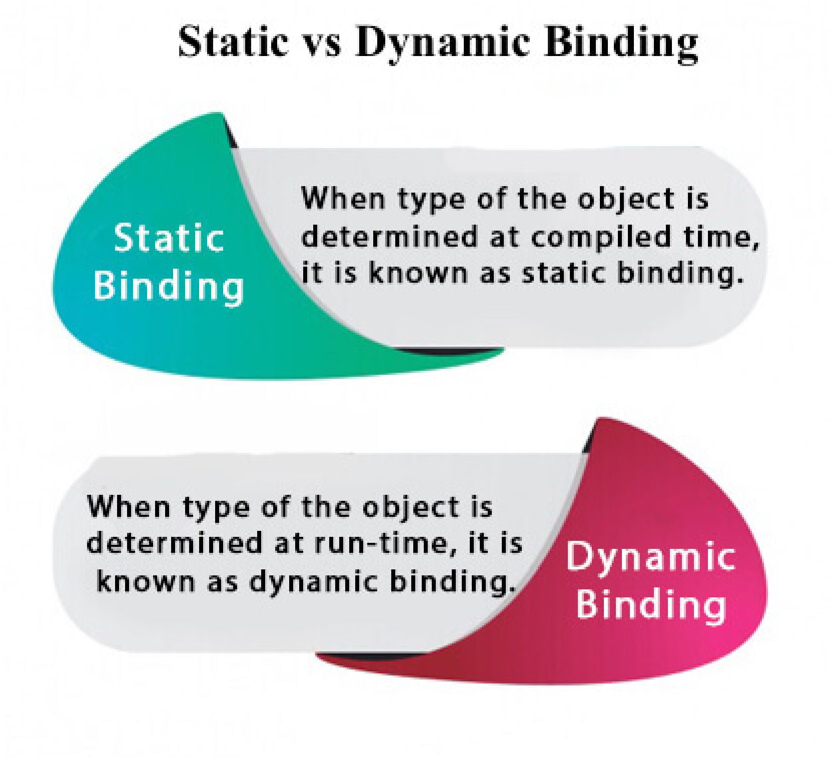
• Occurs during Method Overloading and Operator Overloading	• Occurs during Method Overriding

Static Binding and Dynamic Binding

Connecting a method call to the method body is known as binding.

There are two types of binding

1. Static Binding (also known as Early Binding).
2. Dynamic Binding (also known as Late Binding).


Static vs Dynamic Binding
Static Binding
When type of the object is determined at compiled time, it is known as static binding.
When type of the object is determined at run-time, it is known as dynamic binding.
Dynamic Binding

Understanding Type

Let's understand the type of instance.

1) variables have a type

Each variable has a type, it may be primitive and non-primitive.

```java
int data=30;
```

Here data variable is a type of int.

2) References have a type

```java
 class Dog{
 public static void main(String args[]){
  Dog d1;//Here d1 is a type of Dog
 }
}
```

3) Objects have a type

An object is an instance of particular java class,but it is also an instance of its superclass.

```java
class Animal{}

class Dog extends Animal{
 public static void main(String args[]){
  Dog d1=new Dog();
 }
}
```

Here d1 is an instance of Dog class, but it is also an instance of Animal.

static binding

When type of the object is determined at compiled time(by the compiler), it is known as static binding.

If there is any private, final or static method in a class, there is static binding.

Example of static binding

```java
class Dog{
 private void eat(){System.out.println("dog is eating...");}

 public static void main(String args[]){
  Dog d1=new Dog();
  d1.eat();
 }
}
```

Dynamic binding

When type of the object is determined at run-time, it is known as dynamic binding.

Example of dynamic binding

```
class Animal{
 void eat(){System.out.println("animal is eating...");}
}

class Dog extends Animal{
 void eat(){System.out.println("dog is eating...");}

 public static void main(String args[]){
  Animal a=new Dog();
  a.eat();
 }
}
```

Test it Now

```
Output:dog is eating...
```

In the above example object type cannot be determined by the compiler, because the instance of Dog is also
instance of Animal.So compiler doesn't know its type, only its base type.