

Workshop Link:

<https://attendee.gotowebinar.com/register/2348474801872643416>

<https://tinyurl.com/corejava6pmnotes>

7386095600

durgasoftonlinetraining@gmail.com

Object Orientation:

There are 4 types of Programming Languages.

1. Unstructured Programming Languages
2. Structured Programming Languages
3. Object Oriented Programming Languages
4. Aspect Oriented Programming Languages

Q)What are the differences between Unstructured Programming Languages and Structured Programming Languages?

Ans:

1. Unstructured Programming Languages are outdated programming languages, they were introduced at the starting point of the computers.
EX: BASIC, FORTRAN,.....

Structured Programming Languages are not outdated programming

Languages.

EX: C, PASCAL,....

2. Unstructured Programming Languages are not following any proper structure to prepare applications.

Structured programming languages are following proper structure to prepare Applications.

3. Unstructured Programming Languages are using mnemonic codes, here mnemonic codes are available in a very less number, they will provide less number of features to prepare applications.

Structured programming Languages are using high level syntaxes to prepare applications , where high level syntaxes are available in more numbers and they will provide more features.

4. Unstructured Programming languages are using only “GOTO” statement to define flow of execution in the applications.

Structured programming Languages are using more flow controllers to prepare applications, it will provide very good flow of execution in the applications.

5. Unstructured Programming Languages are not using Functions feature, so there is no chance of providing Code Reusability, it will increase code redundancy, it is not suggestible in application development.

Structured Programming languages are having Functions feature, it is able to provide more Code Reusability, it is suggestible in the application development.

Q)What are the differences between Structured programming languages and Object Oriented Programming languages?

Ans:

1. Structured programming languages are providing a difficult approach to represent real world entities in programming.

Object Oriented Programming languages are providing a simple approach to represent real world entities in programming.

2. Modularization is not good in Structured programming languages.

Modularization is very good in Object Oriented programming Languages.

3. Structured programming languages are not providing very good abstraction in the application.

Object Oriented Programming languages are able to provide very good abstraction in the applications.

4. Security is not good in Structured programming languages.

Security is very good in Object Oriented Programming Languages.

5. Shareability is not good in Structured Programming languages.

Shareability is very good in Object Oriented programming Languages.

6. Reusability is not good in Structured Programming Languages.

Reusability is very good in Object Oriented Programming Languages.

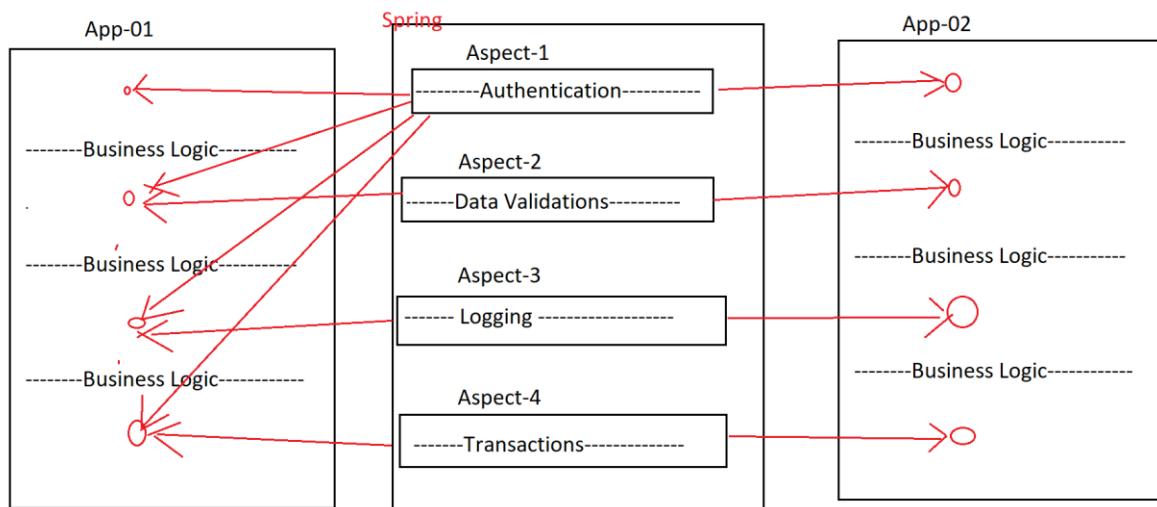
Q)What is the difference between Object Oriented Programming Languages and Aspect Oriented programming Languages?

Ans:

Aspect oriented programming languages is a wrong terminology, because no programming language exists to implement Aspect orientation.

Aspect Orientation is a methodology or a set of conventions or a set of rules and regulations which are applied on the Object Oriented programming in order to improve Shareability and Reusability.

In general, Aspect orientation will be used in the products of Frameworks like Spring Framework, compilers, JVMs, Servers.....



Object Oriented Features:

To describe the nature of Object Orientation , Object Oriented programming languages have provided Object Oriented Features.

1. Class
2. Object
3. Encapsulation

- 4. Abstraction
- 5. Inheritance
- 6. Polymorphism
- 7. Message Passing

On the basis of the above Object oriented features there are two types of programming languages.

- 1. Object oriented Programming Languages
- 2. Object Based Programming Languages

Q)What is the difference between Object Oriented Programming Languages and Object based Programming Languages?

Ans:

Object Oriented programming languages are following almost all the Object Oriented features including Inheritance.

EX: Java

Object Based Programming languages are following almost all the object oriented features excluding Inheritance.

EX: Java Script, SNOBOL

Q)What are the differences between Class and Object?

Ans:

- 1. A group of elements have common properties and common behaviors , here the group of elements is called a Class.

Among the group of elements , an individual element having physical properties and physical behaviors is called an Object.

2. Class is virtual.

Object is real / physical.

3. Class is the virtual encapsulation of the properties and behaviors.

Object is the physical encapsulation of the properties and behaviors.

4. Class is Generalization.

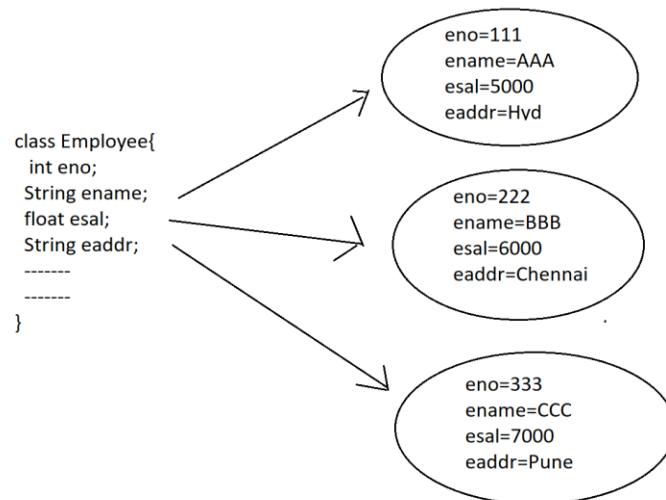
Object is Specialization.

5. Class is a blueprint for the objects.

Object is an instance of the class.

6. Class is a Model for all the objects.

Object is an instance of the class.



Q) What is the difference between Encapsulation and Abstraction?

Ans:

The process of Combining/Binding the data and coding part is called “Encapsulation”.

The process of showing necessary implementation and hiding the unnecessary implementation is called “Abstraction”.

Encapsulation + Abstraction = Security

Inheritance:

It is a relation between classes , it provides variables and methods from one class to another class in order to improve Code Reusability.

The main Advantage of Inheritance is “Code Reusability”.

Polymorphism:

Polymorphism is a Greek word, where Poly means Many and Morphism means forms or Structures.

If one exists in multiple forms then it is called “Polymorphism”.

The main advantage of Polymorphism is “Flexibility”.

Message Passing:

The process of transferring data from one class to another class along with flow of execution is called “Message Passing”.

The main advantage of Message passing is to improve communication between classes, and data navigation.

Containers in Java or Top most elements in Java:

Container is a Java element, it contains some other Java programming elements like variables, blocks, methods,.....

In Java , there are three types of containers.

1. Class
2. Abstract Class
3. Interface

Class:

Class is a container representing entities.

EX: Student, Employee, Product, Account,.....

Note: To represent entities data we have to use “Variables”.

EX: accNo, accHolderName, accType,...

Note: To represent entities behaviors / actions we have to use “Methods”.

EX: createAccount(), updateAccount(), searchAccount(), deleteAccount(),...

To declare classes in java applications we have to use the following syntax.

```
[AccessModifiers] class ClassName [extends SuperClassName]  
[implements InterfaceList]{  
    -----Variables-----  
    -----Methods-----  
    -----Constructors-----  
    -----Blocks-----  
    -----Enums-----
```

```
-----Classes-----  
-----Abstract Classes---  
-----Interfaces-----  
}
```

Observations from Class Syntax:

1. Class syntax can allow only one super class.
2. Class Syntax can allow any number of interfaces.
3. In Class syntax, extends and implements keywords are optional, that is,
 - a. We can write a class with extends keyword without implements keyword.
 - b. We can write a class with implements keyword without extends keyword.
 - c. We can write a class without both extends and implements keywords.
 - d. We can write a class with both extends and implements keywords but first we have to write extends then we have to write implements, not possible to interchange order of extends and implements keywords.
4. We can write classes, abstract classes, interfaces inside the classes, inside the abstract classes and inside the interfaces.

Access Modifiers:

Access modifiers are used to provide scopes to the programming elements and to define some extra nature to the programming elements.

There are two types of Access modifiers in java:

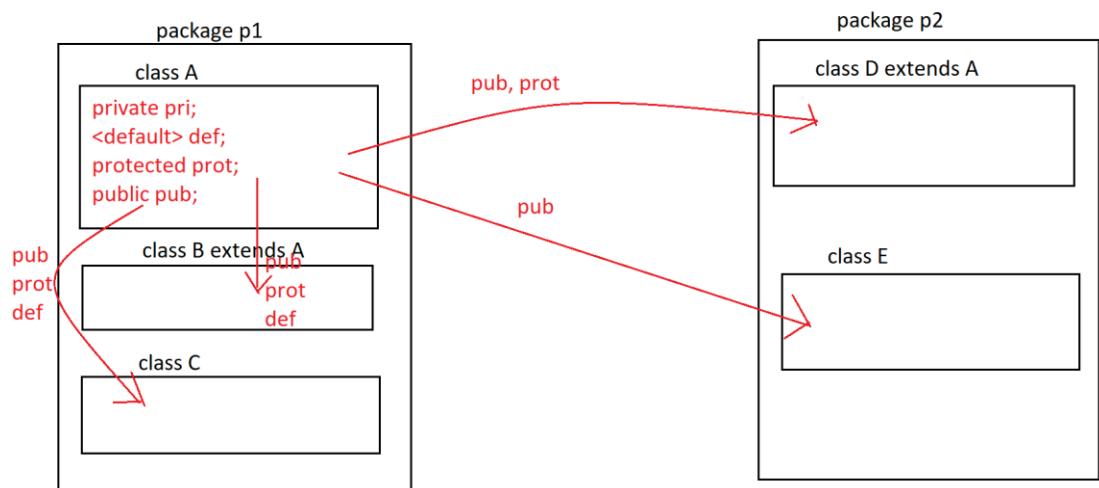
1. To define Scope to the programming elements we will use the Access modifiers like **public** , **protected**, **<default>**, **private**.

Where “private” members will have scope up to the respective class.

Where “<default> members will have scope up to the current package.

Where “protected” members will have scope up to the current package and the child classes existed in the other packages.

Where “public” members will have scope throughout the application.



Out of all the four types of scopes, only public and <default> scopes are applicable for classes, protected and private are not applicable for classes.

Note: Inner classes are able to allow all four types of scopes like public, protected, <default> and private.

Note: If any access modifier is defined on the boundaries of the classes then that access modifier is

not applicable for the classes, it is applicable for the members of the class including inner classes.

EX: The access modifiers like private and protected are defined on the boundaries of the classes, so they are not applicable for the classes , they are applicable for the members of the classes including inner classes.

2. To define some extra nature to the programming elements we will use the access modifiers like static, final, abstract, native, volatile, synchronized, transient, strictfp,.....

From the above access modifiers the access modifiers like final, abstract and strictfp are allowed in the classes declaration, all the remaining access modifiers are not allowed.

Note: From the above list of access modifiers , the access modifiers like static, abstract , final, strictfp are allowed for the inner classes.

Note: ‘static’ keyword is defined on the origin of the classes, so it is not applicable for classes, it is applicable for the members of the classes including inner classes.

Where ‘class’ is a keyword in java, it is able to represent “Class” object oriented feature.

Where ‘className’ is an identifier assigned to the respective class in order to access that class.

Where ‘extends’ is a keyword in Java, it is able to represent “Inheritance” object oriented feature.

Note: In class syntax, ‘extends’ keyword is able to allow only one super class name, not possible to allow more than one superclass, because if we provide more than one superclass to a single subclass then it is Multiple Inheritance , it is not possible in java.

Where ‘implements’ is a java keyword, it is able to represent polymorphism in java applications.

Note: In class Syntax, implements keyword is able to allow more than one interface.

Note: IN class syntax, extends and implements keywords are optional , we can write a class with extends keyword and with out implements keyword, we can write a class without extends keyword and with implements keyword, we can write a class without both extends and implements keyword, we can write a class with both extends and implements keywords but first we have to provide extends then we have to provide implements keyword.

Q)Find the valid syntaxes of the classes from the following syntaxes?

1. public class A{ } -----> Valid
2. protected class A{ } -----> Invalid
3. class A{ } -----> Valid
4. private class A{ } -----> Invalid
5. class A{ private class B{} } -----> Valid

```
6. class A{      class B{}      } ----->
   Valid
7. class A{      protected class B{}     } ---->
   Valid
8. class A{    public class B{}     } -----> Valid
9. class A extends B{}      } -----> Valid
10.    class A extends B, C{}     } -----> Invalid
11.    class A implements I{}     } -----> Valid
12.    class A implements I1, I2{} } ----->
   Valid
13.    class A implements I extends B{} } ---->
   Invalid
14.    class A extends B implements I{} } ----->
   Valid
15.    class A extends B implements I1, I2{} } --
-> Valid
16.    static class A{}      } ----->
   Invalid
17.    abstract class A{}     } -----> Valid
18.    final class A{}      } ----->
   Valid
19.    native class A{}     } ----->
   Invalid
20.    strictfp class A{}    } ----->
   Valid
21.    class A {    static class B{}  } ---->
   Valid
22.    class A{      abstract class B{} } ---->
   Valid
23.    class A{    volatile class B{} } ---->
   Invalid
24.    class A{    strictfp class B{} } ---->
   Valid
```

Procedure to use classes in Java Applications:

1. Declare a class by using the 'class' keyword.

2. Declare variables and methods inside the class as per the requirement.
3. Declare main class and main() method.
4. Inside the main class, inside the main() method , create an object for the class.
5. Access members of the class by using the generated reference variable.

EX:

```

class Employee{

    int eno = 111;
    String ename = "Durga";
    float esal = 50000.0f;
    String equal = "MBA";
    String edes = "Manager";
    String emailId = "durga@dss.com";
    String emobile = "91-9988776655";
    String eaddr = "Hyderabad";

    public void displayEmpDetails(){
        System.out.println("Employee Details");
        System.out.println("-----");
        System.out.println("Employee Number      : "+eno);
        System.out.println("Employee Name       : "+ename);
        System.out.println("Employee Salary     : "+esal);
        System.out.println("Employee Qualification: "+equal);
        System.out.println("Employee Designation : "+edes);
        System.out.println("Employee Email Id   : "+emailId);
        System.out.println("Employee Mobile Number: "+emobile);
        System.out.println("Employee Address     : "+eaddr);
    }

}

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

```

```

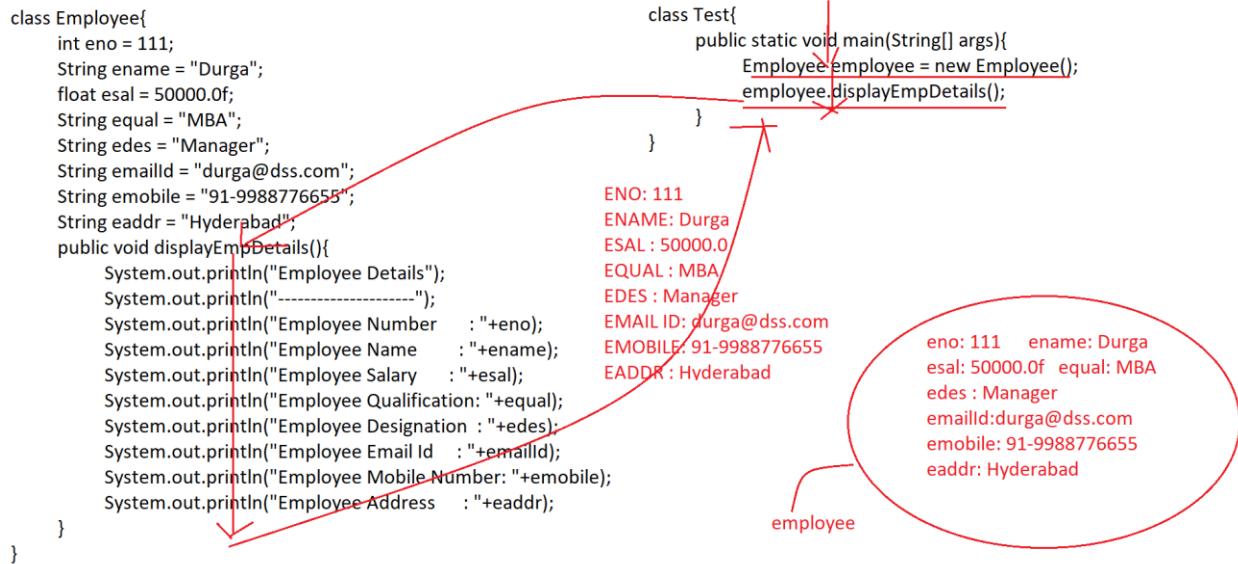
        Employee employee = new Employee();
        employee.displayEmpDetails();
    }
}

```

```

D:\java6>javac Test.java
D:\java6>java Test
Employee Details
-----
Employee Number      : 111
Employee Name       : Durga
Employee Salary     : 50000.0
Employee Qualification: MBA
Employee Designation : Manager
Employee Email Id   : durga@dss.com
Employee Mobile Number: 91-9988776655
Employee Address     : Hyderabad

```



EX:

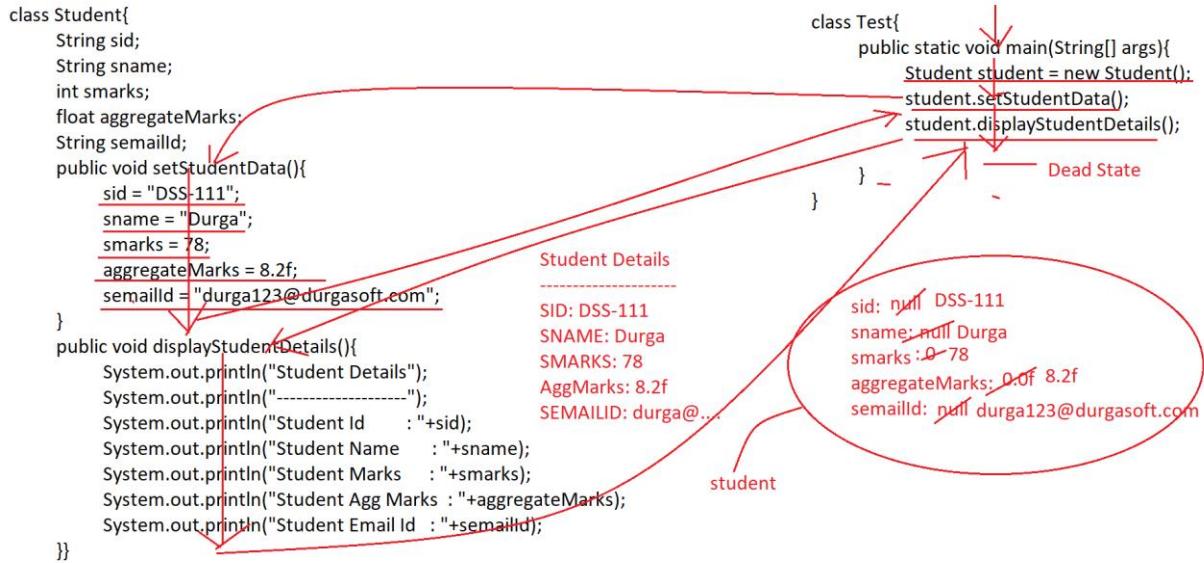
```
-----
class Student{
```

```
String sid;
String sname;
int smarks;
float aggregateMarks;
String semailId;
public void setStudentData(){
    sid = "DSS-111";
    sname = "Durga";
    smarks = 78;
    aggregateMarks = 8.2f;
    semailId = "durga123@durgasoft.com";
}

public void displayStudentDetails(){
    System.out.println("Student Details");
    System.out.println("-----");
    System.out.println("Student Id : "+sid);
    System.out.println("Student Name : "+sname);
    System.out.println("Student Marks : "+smarks);
    System.out.println("Student Agg Marks : "+aggregateMarks);
    System.out.println("Student Email Id : "+semailId);
}
}

class Test{
    public static void main(String[] args){
        Student student = new Student();
        student.setStudentData();
        student.displayStudentDetails();

    }
}
```



```

class Account{

    String accNo;
    String accHolderName;
    String accType;
    double accBalance;

    public void setAccountDetails(String acc_No, String
acc_Holder_Name, String acc_Type, double acc_Balance){
        accNo = acc_No;
        accHolderName = acc_Holder_Name;
        accType= acc_Type;
        accBalance = acc_Balance;
    }
    public void displayAccountDetails(){
        System.out.println("ACcount Details");
        System.out.println("-----");
        System.out.println("Account Number : "+accNo);
    }
}

```

```
        System.out.println("Account Holder Name :  
"+accHolderName);  
        System.out.println("Account Type      : "+accType);  
        System.out.println("Account Balance   :  
"+accBalance);  
    }  
  
}  
class Test{  
    public static void main(String[] args){  
        Account account1 = new Account();  
        account1.setAccountDetails("abc123", "AAA", "Savings",  
10000.0);  
  
        Account account2 = new Account();  
        account2.setAccountDetails("xyz123", "BBB", "Savings",  
200000.0);  
  
        Account account3 = new Account();  
        account3.setAccountDetails("aaa123", "CCC", "Current",  
300000.0);  
  
        account1.displayAccountDetails();  
        account2.displayAccountDetails();  
        account3.displayAccountDetails();  
    }  
}
```

```

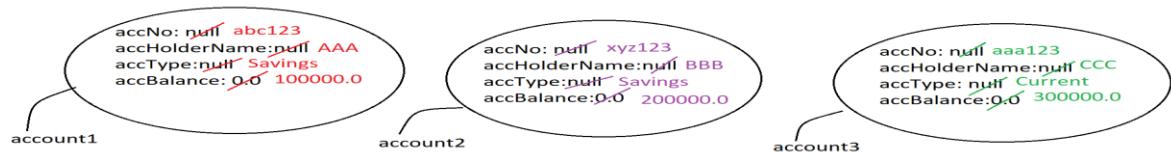
class Account{
    String accNo; abc123 xyz123 aaa123
    String accHolderName; AAA BBB CCC aaa123
    String accType; Savings Savings Current
    double accBalance; 100000.0 200000.0 XYZ123
                                         300000.0 abc123
    public void setAccountDetails(String acc_No, String acc_Holder_Name,
        String acc_Type, double acc_Balance){ Savings 100000.0
                                                Savings 200000.0
                                                Current 300000.0
        accNo = acc_No;
        accHolderName = acc_Holder_Name;
        accType = acc_Type;
        accBalance = acc_Balance;
    }
    public void displayAccountDetails(){
        System.out.println("Account Details");
        System.out.println("-----");
        System.out.println("Account Number : "+accNo);
        System.out.println("Account Holder Name : "+accHolderName);
        System.out.println("Account Type : "+accType);
        System.out.println("Account Balance : "+accBalance);
    }
}

```

```

class Test{
    public static void main(String[] args){
        Account account1 = new Account();
        account1.setAccountDetails("abc123", "AAA",
                                   "Savings", 100000.0);
        Account account2 = new Account();
        account2.setAccountDetails("xyz123", "BBB",
                                   "Savings", 200000.0);
        Account account3 = new Account();
        account3.setAccountDetails("aaa123", "CCC",
                                   "Current", 300000.0);
        account1.displayAccountDetails();
        account2.displayAccountDetails();
        account3.displayAccountDetails();
    }
}
ACCNO:abc123 ACCNO:xyz123
ACCHOLDERNNAME: AAA ACCHOLDERNNAME:BBB ....
ACCTYPE:Savings ACCTYPE:Savings
ACCBALANCE:100000.0 ACCBALANCE:200000.0

```



In Java, there are two types of Methods

1. Concrete Methods
2. Abstract Methods

Q) What are the differences between Concrete Methods and Abstract Methods?

Ans:

1. Concrete Method is a Java method, it will have both method declarative part and method implementation part.

Abstract Method is a Java method, it will have only Method declarative part without the implementation part.

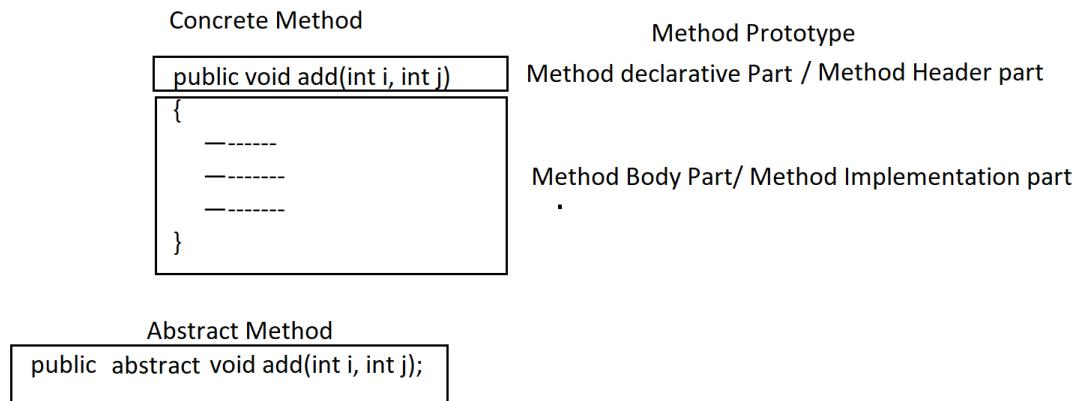
2. To declare concrete methods no need to use a special keyword.

To declare abstract methods we must use a special keyword like “abstract”.

3. Concrete methods are possible in classes and abstract classes.

Abstract methods are possible in abstract classes and interfaces.

4. Concrete Methods are able to provide less shareability.
Abstract methods are able to provide more shareability.



Abstract Classes:

Abstract class is a Java class, it is able to allow both concrete methods[zero or more number] and abstract methods[zero or more number].

- a. We can write an abstract class with only concrete methods without the abstract methods.
- b. We can write an abstract class with only abstract methods without concrete methods.
- c. We can have an abstract class with both concrete methods and abstract methods.
- d. We can write an abstract class without both concrete methods and abstract methods.

In Java applications, for abstract classes we are able to declare reference variables only, we are unable to create objects.

In Java applications, to declare an abstract class we have to use a special keyword that is “abstract” .

Note: If we declare any method or any class with “abstract” keyword then the compiler and JVM will understand that method and that class as an incomplete method and an incomplete class, for them we are unable to create objects.

Procedure to use abstract classes in Java applications:

1. Declare an abstract class with “abstract” keyword.
2. Declare variables and methods inside the abstract class as per the application requirement.
3. Declare a subclass for the abstract class.
4. Provide implementation for all the abstract methods of the abstract class in the sub class.
5. Declare main class, main() method.
6. In the main class, in the main() method create an object for the sub class and declare a reference variable either for abstract class or for sub class.

7. Access abstract class members.

Note: If we declare a reference variable for the abstract class[super class] then it is possible to access only abstract class members, not possible to access subclass own members . If we declare a reference variable for a subclass then it is possible to access both abstract class [super class] members and subclass own members.

EX:

```
abstract class A{
    int x = 10;
    void m1(){
        System.out.println("m1-A");
    }
    abstract void m2();
    abstract void m3();
}
class B extends A{
    void m2(){
        System.out.println("m2-B");
    }
    void m3(){
        System.out.println("m3-B");
    }
    void m4(){
        System.out.println("m4-B");
    }
}
class Test{
    public static void main(String[] args){
        //A a1 = new A(); ---> Error
        A a = new B();
        System.out.println(a.x);
        a.m1();
        a.m2();
```

```

    a.m3();
    //a.m4(); -----> Error
    System.out.println();

    B b = new B();
    System.out.println(b.x);
    b.m1();
    b.m2();
    b.m3();
    b.m4();
}
}

```

EX:

--

```

abstract class Account{
    public int getMinBal(){
        return 2000;
    }
    public abstract float getIR();
}

class LoanAccount extends Account{
    public float getIR(){
        return 7.5f;
    }
}
class SalaryAccount extends Account{
    public float getIR(){
        return 8.2f;
    }
}
class BusinessAccount extends Account{
    public float getIR(){
        return 9.5f;
    }
}

```

```
class Test{
    public static void main(String[] args){
        Account loanAccount = new LoanAccount();
        System.out.println("Loan Account Min BAL : "+loanAccount.getMinBal());
        System.out.println("Loan Account IR : "+loanAccount.getIR());
        System.out.println();

        Account salaryAccount = new SalaryAccount();
        System.out.println("Salary Account Min BAL : "+salaryAccount.getMinBal());
        System.out.println("Salary Account IR : "+salaryAccount.getIR());
        System.out.println();

        Account businessAccount = new BusinessAccount();
        System.out.println("Business Account Min BAL : "+businessAccount.getMinBal());
        System.out.println("Busniess Account IR : "+businessAccount.getIR());
        System.out.println();
    }
}
```

D:\java6>javac Test.java

D:\java6>java Test
Loan Account Min BAL : 20000
Loan Account IR : 7.5

Salary Account Min BAL : 20000
Salary Account IR : 8.2

Business Account Min BAL : 20000
Busniess Account IR : 9.5

Q)What are the differences between Concrete class[Normal Class] and abstract class?

Ans:

1. Concrete classes are able to allow only concrete methods.

Abstract classes are able to allow both Concrete methods and Abstract methods.

2. To declare concrete classes we have to use the “class” keyword.

To declare abstract classes we have to use the “abstract” keyword along with “class” keyword.

3. For the concrete classes we are able to provide both reference variables and objects.

For the abstract classes we are able to provide only reference variables, we are unable to create objects.

4. Concrete classes are able to provide less shareability.

Abstract classes are able to provide more shareability.

Interfaces:

Interface is a Java feature, it is able to allow only abstract methods, not possible to have concrete methods.

Note: From JAVA8 version onwards interfaces are able to have default methods and static methods with the implementation part.

Note: From JAVA 9 version onwards interfaces are able to have private methods with the implementation.

To declare interfaces we will use a separate keyword that is “interface”.

For the interfaces we are able to create reference variables only, we are unable to create objects.

Inside the interfaces, by default all variables are “public static final”, not required to declare explicitly.

Inside the interfaces, by default all methods are “public and abstract”, not required to declare explicitly.

Procedure to use Interfaces in Java applications:

1. Declare an interface with the “interface” keyword.
2. Declare variables and methods inside the interface as per the application requirement.
3. Declare an implementation class for the interface.
4. Provide implementation for all the abstract methods of the interface inside the implementation class.
5. Declare Main class and main() method.
6. Create objects for the implementation class and declare reference variables either for the interface or for the implementation class.
7. Access interface members.

Note: If we declare reference variables for the interface then we are able to access only interface members, we are unable to access implementation class own members, If we declare reference variables to the implementation class then we are able to access both interface members and implementation class own members.

EX:

```
interface I{
    int x = 10;// public static final
    void m1();// public abstract
    void m2();// public abstract
    void m3();// public abstract
}
class A implements I{
    public void m1(){
        System.out.println("m1-A");
    }
    public void m2(){
        System.out.println("m2-A");
    }
    public void m3(){
        System.out.println("m3-A");
    }
    public void m4(){
        System.out.println("m4-A");
    }
}
class Test{
    public static void main(String[] args){
        //I i1 = new I(); ---> Error
        I i = new A();
        i.m1();
        i.m2();
        i.m3();
        //i.m4(); ---> Error
        System.out.println();

        A a = new A();
        a.m1();
        a.m2();
        a.m3();
        a.m4();
        System.out.println();
    }
}
```

```
        System.out.println(I.x);
        System.out.println(i.x);
        System.out.println(A.x);
        System.out.println(a.x);

    }

}
```

EX:

```
interface Loan{
    long getLoanAmount();
    float getIR();
}

class GoldLoan implements Loan{
    public long getLoanAmount(){
        return 500000L;
    }
    public float getIR(){
        return 5.5f;
    }
}

class HomeLoan implements Loan{
    public long getLoanAmount(){
        return 1000000L;
    }
    public float getIR(){
        return 7.2f;
    }
}

class CraftLoan implements Loan{
    public long getLoanAmount(){
        return 300000L;
    }
    public float getIR(){
```

```
        return 2.3f;
    }
}

class StudyLoan implements Loan{
    public long getLoanAmount(){
        return 200000L;
    }
    public float getIR(){
        return 12.5f;
    }
}

class Test{
    public static void main(String[] args){
        Loan goldLoan = new GoldLoan();
        System.out.println("Gold Loan Amount : "+goldLoan.getLoanAmount());
        System.out.println("Gold Loan IR : "+goldLoan.getIR());
        System.out.println();

        Loan homeLoan = new HomeLoan();
        System.out.println("Home Loan Amount : "+homeLoan.getLoanAmount());
        System.out.println("Home Loan IR : "+homeLoan.getIR());
        System.out.println();

        Loan craftLoan = new CraftLoan();
        System.out.println("Craft Loan Amount : "+craftLoan.getLoanAmount());
        System.out.println("Craft Loan IR : "+craftLoan.getIR());
        System.out.println();

        Loan studyLoan = new StudyLoan();
        System.out.println("Study Loan Amount : "+studyLoan.getLoanAmount());
    }
}
```

```
        System.out.println("Study Loan IR      :  
"+studyLoan.getIR());  
  
    }  
}  
  
}
```

D:\java6>javac Test.java

```
D:\java6>java Test  
Gold Loan Amount   : 500000  
Gold Loan IR       : 5.5  
  
Home Loan Amount   : 1000000  
Home Loan IR       : 7.2  
  
Craft Loan Amount  : 300000  
Craft Loan IR      : 2.3  
  
Study Loan Amount  : 200000  
Study Loan IR      : 12.5
```

Q)What are the differences between classes, abstract classes and interfaces?

Ans:

1. To declare a class we have to use only the “class” keyword.

To declare an abstract class we have to use the “abstract” keyword along with “class” keyword.

To declare an interface we have to use the “interface” keyword.

2. Classes are able to allow only concrete methods.

Abstract classes are able to allow both concrete methods and abstract methods.

Interfaces are able to allow only abstract methods.

3. For classes , we are able to create both reference variables and objects.

For the abstract classes and interfaces , we are able to create only reference variables , we are unable to create objects.

4. In the case of interfaces, by default all variables are “public static final”.

No default cases exist for the variables in the classes and abstract classes.

5. In the case of interfaces, by default all methods are “public and abstract”.

No default cases exist for the methods in the classes and abstract classes.

6. Classes are able to provide less shareability.

Abstract classes are able to provide middle level shareability.

Interfaces are able to provide more shareability.

Method in Java:

To represent entities' behaviors or actions we have to use methods in the classes.

Method is a set of instructions , which are representing a particular action of an entity.

EX: Transaction is an entity , where in Transactions we are able to define the behaviors or actions like “deposit”, “withdraw”, “transferFunds”,....

EX: Account is an entity, where in Account entity we are able to define the behaviors or actions like “createAccount”, “searchAccount”, “updateAccount”, “deleteAccount”.

To prepare Methods in Java applications we have to use the following syntax.

```
[AccessModifiers] ReturnType methodName([ParamList])[throws  
ExceptionList]{  
-----  
-----  
-----  
[return Value;]  
}
```

All Java methods are able to allow the access modifiers like public, protected, <default> and private.

From the above list of access modifiers only one access modifier we are able to provide in the methods, it is not possible to provide more than one access modifier.

All Java methods are able to allow the access modifiers like static, final, abstract, native, synchronized and strictfp.

From the above list of access modifiers we are able to use more than one access modifier in the methods, but they must be in valid combination.

EX1: static final : valid

```
static final void m1(){
    -----
}

EX2: static abstract : Invalid
static abstract void m1(){
-----
}
Status: Invalid.

EX3: final abstract :
final abstract void m1(){
-----
}
Status: Invalid
```

Where “ReturnType” is mandatory in Java methods, it represents the type of the data which the Java method wants to return as an output from the action which is represented in the form of the method.

In Java applications, we are able to use all primitive data types[byte, short, int, long, float, double, boolean and char] , all user defined data types [classes, abstract classes, interfaces, arrays, enums,...] and “void” as return types.
Note: ‘void’ return type is representing “no Value to return” from the method.

Where “methodName” is a name assigned to the method in order to access that method.

Where “paramList” is a list of parameters to the method which are used to provide input data to the methods in order to perform the respective action.

In Java methods, parameterList is able to allow all primitive data types and all user defined data types.

In Java methods, throws keyword can be used to bypass the generated exception from the present method to the caller method in order to handle that exception.

EX:

```
void m1()throws ExceptionName{  
    void m2();  
}  
void m2()throws ExceptionName{  
    void m3();  
}  
void m3()throws ExceptionName{  
    ----Exception----  
}
```

Where “return” keyword can be used to return a value from the method to the caller of the method as per the return type.

Q) Find the valid syntaxes of the methods from the following list?

Ans:

1. public void m1(){ } -----> Valid
2. public void m1(){ }; -----> Valid
3. protected void m1(){ } -----> Valid
4. default void m1(){ } -----> Invalid
5. void m1(){ } -----> Valid
6. private void m1(){ } -----> Valid
7. public protected void m1(){ } ----->
 Invalid
8. static void m1(){ } -----> Valid
9. final void m1(){ }-----> Valid
10. native void m1(); -----> Valid
11. abstract void m1();-----> Valid

12. volatile void m1(){ } ----->
 invalid

13. abstract void m1(){ }; ----->
 Invalid

14. transient void m1(){ } ----->
 Invalid

15. synchronized void m1(){ } ----->
 Valid

16. strictfp void m1(){ } ----->
 Valid

17. int m1(){ }; -----> Invalid

18. float m1(){ return 0.0f; } ----->
 --> Valid

19. long m1(){ return 0.0f; } ----->
 -> Invalid

20. double m1(){ return 0.0f; } ----->
 ---> Valid

21. double m1(){ return 10; } ----->
 Valid

22. String m1(){ return "abc"; } ----->
 ----> Valid

23. int[] m1(){ return new int[]{10,20,30}; } ----->
 -----> Valid

24. void m1(int i, int j){ } ----->
 Valid

25. void m1(void){ } -----> Invalid

26. void 9m1(){ } -----> Invalid

27. void m1(String str){ } ----->
 valid

28. void m1(String str){ return str; } ----->
 -----> Invalid

29. void m1(){ void m2(){ } } ----->
 -----> Invalid

30. int[] m1(int[] intArray){ return intArray; } ----->
 -----> Valid

31. void m1(){ class A{ }; } ----->
 Valid

In Java , there are two ways to provide method description.

1. Method Signature
2. Method Prototype

Q)What is the difference between Method Signature and Method Description?

Ans:

Method Signature is the method description which includes method name and parameter List.

EX:

```
forName(String className)
```

Method Prototype is the method description which includes the method details like Access Modifiers, Return type, Method Name, Parameter List and throws Exception List.

EX:

```
public static Class forName(String className) throws  
ClassNotFoundException
```

In Java applications, there are two types of methods on the basis of the Object state manipulations.

1. Mutator Methods
2. Accessor Methods

Q)What is the difference between the Mutator Method and Accessor Method?

Ans:

If any Java method is able to perform modifications on the data inside an object then that Java method is called the “Mutator Method”.

EX: All setXXX() methods in the Java bean classes are Mutator methods.

If any Java method is able to get/access data from the Object then that method is called “Accessor Method”.

EX: All getXxx() methods in the Java Bean classes are Accessor Methods.

Note: Java Bean is a normal java class containing private variables and the respective setXXX() methods to set data to the variables and getXXX() methods to get data from the variables.

EX:

```
class Employee{  
    private int eno;  
    private String ename;  
    private float esal;  
  
    public void setEno(int emp_No){// Mutator Method  
        eno = emp_No;  
    }  
    public int getEno(){// Accessor Method  
        return eno;  
    }  
  
    public void setEname(String emp_Name){// Mutator Method  
        ename = emp_Name;  
    }  
    public String getEname(){// Accessor Method  
        return ename;  
    }  
}
```

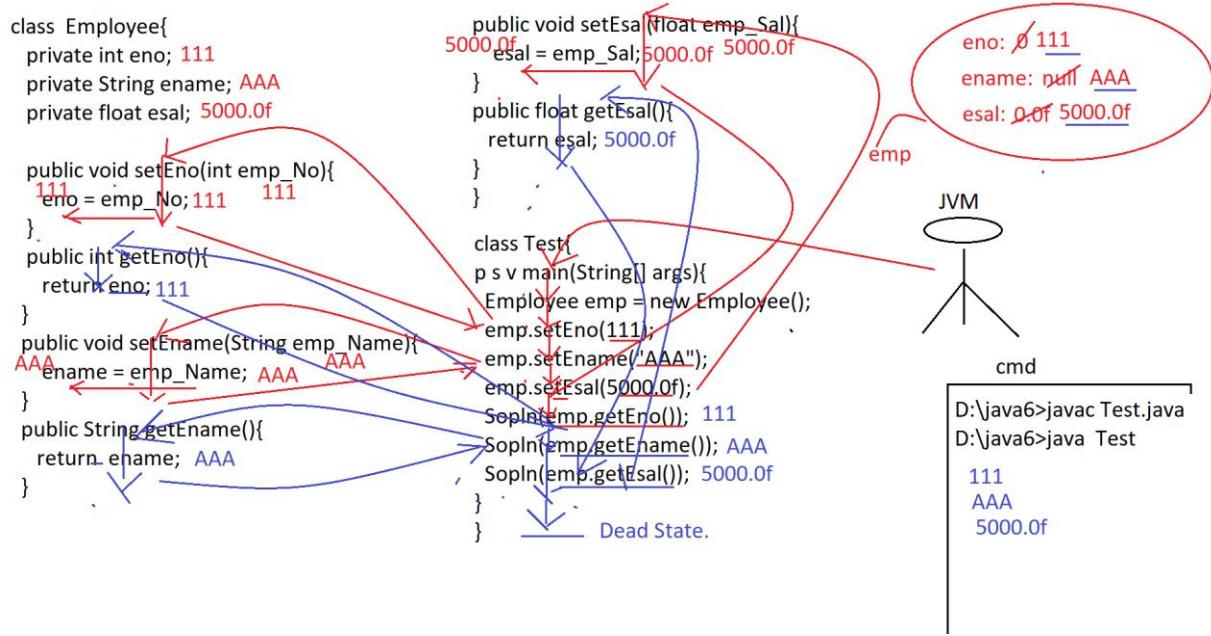
```
public void setEsal(float emp_Sal){// Mutator Method
    esal = emp_Sal;
}
public float getEsal(){// Accessor Method
    return esal;
}
}
class Test{
    public static void main(String[] args){

        Employee emp = new Employee();

        emp.setEno(111);
        emp.setEname("AAA");
        emp.setEsal(50000.0f);

        System.out.println("Employee Details");
        System.out.println("-----");
        System.out.println("Employee Number : "+emp.getEno());
        System.out.println("Employee Name : "+emp.getEname());
        System.out.println("Employee Salary : "+emp.getEsal());

    }
}
```



Variable-Argument Method:

In Java applications, if we declare a method with 'n' number of parameters then we are able to access that method by passing the same 'n' number of parameter values only, it is not possible to access that method by passing n+1 number of parameter values and n-1 number of parameter values.

EX:

```

class A{
    void add(int i, int j){
        System.out.println("add()-A");
    }
}
class Test{
    public static void main(String[] args){
        A a = new A();
        //a.add(); ---> Error
        //a.add(10); --> Error
        a.add(10,20);
        //a.add(10,20,30); --> Error
    }
}

```

```
}
```

In Java applications, we want to define a method and we want to access that method by passing a variable number of argument values. To achieve this requirement JAVA has provided a separate method convention in its JDK1.5 version that is “**Variable-Argument Method**” , in short “**Var-Arg Method**”.

“Var-Arg” method is a Java method , it must have a Var-Arg parameter.

Syntax:

```
void m1(DataType ... VarName){ // DataType[]  
-----  
}
```

Where “**DataType ... VarName**” is Var-Arg Parameter.

Where Var-Arg parameter is internally an array of the same data type.

If we access the Var-Arg method by passing some parameter values then all the parameter values will be stored in the Var-Arg parameter represented array internally.

EX:

```
class A{  
    void add(int ... a){  
        System.out.println("No Of Arguments : "+a.length);  
        int result = 0;  
        System.out.print("Argument List : ");  
        for(int index = 0; index < a.length; index++){  
            System.out.print(a[index]+" ");  
            result = result + a[index];  
        }  
        System.out.println();  
        System.out.println("Arguments SUM : "+result);  
    }  
}
```

```
        System.out.println("-----");
    });
}
}
class Test{
    public static void main(String[] args){
        A a = new A();
        a.add();
        a.add(10);
        a.add(10,20);
        a.add(10,20,30);
        a.add(10,20,30,40);
    }
}
```

D:\java6>javac Test.java

D:\java6>java Test

No Of Arguments : 0

Argument List :

Arguments SUM : 0

No Of Arguments : 1

Argument List : 10

Arguments SUM : 10

No Of Arguments : 2

Argument List : 10 20

Arguments SUM : 30

No Of Arguments : 3

Argument List : 10 20 30

Arguments SUM : 60

No Of Arguments : 4

Argument List : 10 20 30 40

Arguments SUM : 100

Q) Is it possible to provide other parameters along with the Variable-Argument parameter in the Variable-Argument method?

Ans:

Yes, it is possible to provide other parameters along with variable-argument parameter in variable-argument method, but the other parameters must be provided before the Variable-Argument parameter, because in Var-ARg methods , Var-ArG parameter must be the last parameter.

EX:

1. void m1(int ... a, float f){ } -----> Invalid
2. void m1(float f, int ... a){ } -----> Valid

Q) Is it possible to provide more than one Var-Arg parameter in a single Var-Arg method?

Ans:

No, it is not possible to provide more than one Var-Arg parameter in a single Var-Arg method, because In the Var-Arg method, Var-Arg parameter must be the last parameter.

EX: void m1(int ... a, float ... b){ } -----> Invalid.

Object Creation Process in Java:

Q) What is the requirement to create objects in Java applications?

Ans:

In Java applications , we must create objects for the classes due to the following reasons.

1. Java is an Object oriented programming language, in java applications every operation is associated with objects only, so it is a minimum convention to create objects for the classes in java applications.
2. In Java applications, we will create objects to store entities data temporarily, because in java applications , Objects are stored in Heap memory that is in RAM memory, it is temporary memory.
3. In Java applications, if we want to access the members of any particular class then we have to create an object for the respective class and we have to use the generated reference variable.

In Java applications , to create an object for a particular class we have to use the following syntax.

```
ClassName refVar = new ClassName([ParamValues]);
```

Where ‘ClassName([ParamValues])’ is a constructor of the respective class.

EX:

```
class Employee{  
}
```

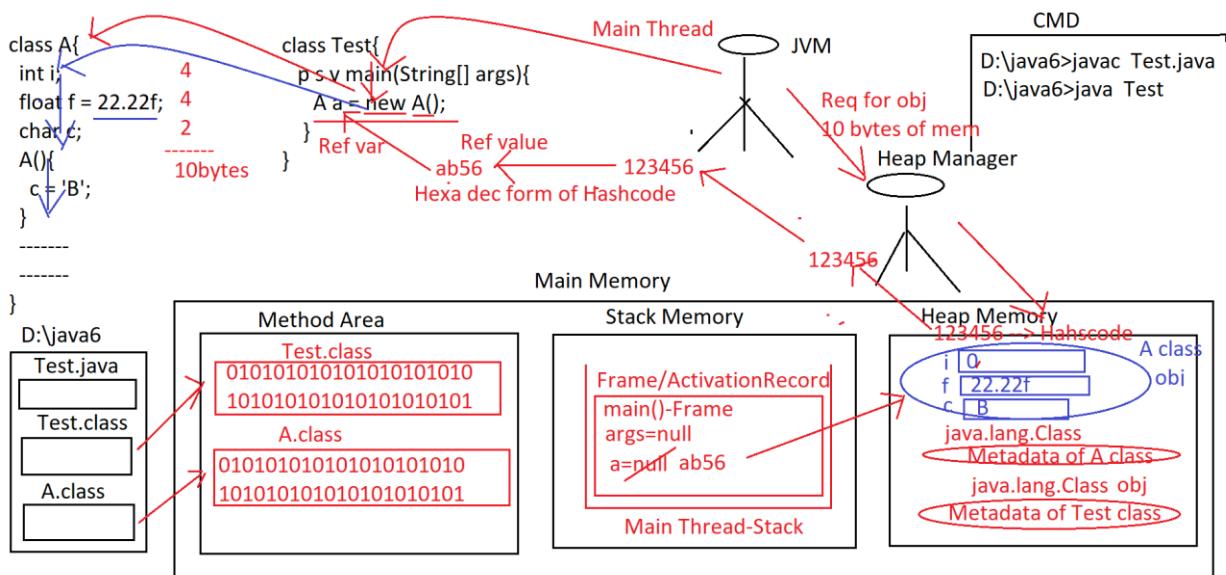
```
Employee employee = new Employee();
```

Where ‘Employee()’ is a constructor of the Employee class.

When a JVM executes object creation statement , that is ‘new’ keyword then JVM will perform the following actions.

1. JVM will know which class object is creating on the basis of the constructor name.

2. JVM will search for the respective class .class file, if it is identified then JVM will load its bytecode to the memory.
3. JVM will create java.lang.Class object in heap memory w.r.t the class bytecode loading.
4. In java.lang.Class object, JVM is able to provide the complete metadata of the loaded class which includes class name, super class details, implemented interfaces details, variables data, methods data,.....
5. JVM will go to the loaded class and recognize all the instance variables and their data types.
6. JVM will calculate minimal memory size for the object on the basis of the instance variables and their data types.
7. JVM will request the Heap Manager to create an object with the generated minimal memory size.
8. Heap Manager will create an object and Heap manager will assign an unique identity in the form an integer value called “HashCode”.
9. Heap Manager will send Object hashCode value to JVM, where JVM will convert Hashcode value into its Hexa-Decimal value called Object reference value.
10. JVM will assign object reference value to a variable called Object reference variable.
11. JVM will allocate memory for all the instance variables inside the object and JVM will provide initial values for the instance variables by checking initializations at class level declaration and at the constructor.
12. If any instance variable is not having initialization at both class level declaration and constructor then JVM will store default value on the basis of the instance variable data type.



To get the hashCode value of an object we have to use the following method.

```
public native int hashCode()
```

Note: Native method is a method declared in Java and implemented in non java programming languages like C, C++, Pascal, H/W languages,.....

To get an Object reference value we have to use the following method.

```
public String toString()
```

EX:

```
class Employee{  
}  
class Test{  
    public static void main(String[] args){  
        Employee emp = new Employee();  
        int hashCode = emp.hashCode();  
        System.out.println("HashCode : "+hashCode);  
}
```

```
        String refVal = emp.toString();
        System.out.println("Ref Value : "+refVal);

    }

}
```

D:\java6>javac Test.java

```
D:\java6>java Test
HashCode : 918221580
Ref Value : Employee@36baf30c
```

In Java, there is a common and default superclass for every java class that is “java.lang.Object”.

java.lang.Object class has the following 11 number of methods in order to provide these methods to every java class.

1. hashCode()
2. toString()
3. equals()
4. clone()
5. finalize()
6. getClass()
7. wait()
8. wait(long time)
9. wait(long time, long time)
10. notify()
11. notifyAll()

```
D:\java6>javap java.lang.Object
Compiled from "Object.java"
public class java.lang.Object {
    public java.lang.Object();
```

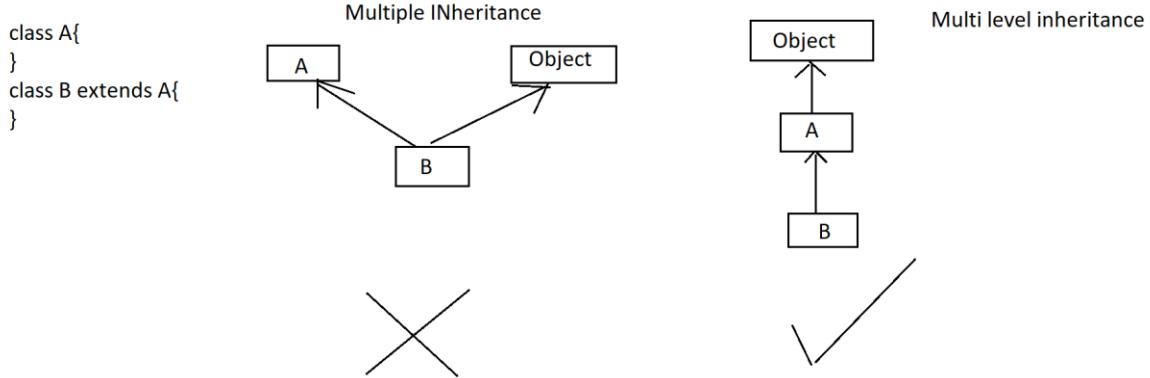
```
public final native java.lang.Class<?> getClass();
public native int hashCode();
public boolean equals(java.lang.Object);
protected native java.lang.Object clone() throws
java.lang.CloneNotSupportedException;
public java.lang.String toString();
public final native void notify();
public final native void notifyAll();
public final void wait() throws java.lang.InterruptedException;
public final native void wait(long) throws
java.lang.InterruptedException;
public final void wait(long, int) throws
java.lang.InterruptedException;
protected void finalize() throws java.lang.Throwable;
}
```

D:\java6>

Q) In Java, `java.lang.Object` class is a common and default superclass for every class, if we provide a superclass for a subclass explicitly then the respective subclass will have two super classes , one is `java.lang.Object` class and other one is the provided explicit superclass , it represents multiple inheritance , how can we say multiple inheritance is not possible in Java?

Ans:

In Java applications, `java.lang.Object` class is a common and default superclass for every java class when the class is not extending from any other class explicitly, if the present class is extending from a superclass explicitly then `java.lang.Object` class is superclass to the sub class indirectly like through Multilevel inheritance not through multiple inheritance.



In Object class, the `toString()` method was implemented in such a way that to return a string containing “`ClassName@RefVal`”.

In Java applications, if we pass a particular object reference variable as parameter to `System.out.println()` method then JVM will access `toString()` method over the provided object reference variable, in this case, JVM will search for `toString()` method in the respective class first, if no `toString()` method is available explicitly in the respective class then JVM will search for the `toString()` method in the respective class's super class, if no super class available explicitly then JVM will search for `toString()` method in the common and default superclass that is Object class, Object class provided `toString()` method will return a string contains “`ClassName@RefValue`”.

EX:

```
class Employee{
```

```

}

class Test{
    public static void main(String[] args){
        Employee emp = new Employee();
        String ref = emp.toString();
        System.out.println(ref);

        System.out.println(emp.toString());

        System.out.println(emp);// Sopln(emp.toString());
    }
}

```

In the above context, if we want to display our own data when we pass an object reference variable as parameter to `System.out.println()` method then we have to define our own `toString()` method in the respective class.

EX:

```

class Employee{
    int eno = 111;
    String ename = "Durga";
    float esal = 50000.0f;
    String eaddr = "Hyd";

    public String toString(){
        System.out.println("Employee Details");
        System.out.println("-----");
        System.out.println("Employee Number : "+eno);
        System.out.println("Employee Name : "+ename);
        System.out.println("Employee Salary : "+esal);
        System.out.println("Employee Address : "+eaddr);
        return "";
    }
}

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

```

```
        Employee emp = new Employee();
        System.out.println(emp);
    }
}
```

OP:

```
D:\java6>javac Test.java
```

```
D:\java6>java Test
Employee Details
-----
Employee Number      : 111
Employee Name        : Durga
Employee Salary       : 50000.0
Employee Address     : Hyd
```

In Java , some predefined classes like String, Exception, Thread, Collection classes, wrapper classes,..... are not depending on the Object class provided `toString()` method, they have their own `toString()` methods to display their own data when we pass the respective class object reference variable as parameter to `System.out.println()` method.

EX:

--

```
import java.util.*;
class Test{
    public static void main(String[] args){
        String str = new String("abc");
        System.out.println(str);
        Exception e = new Exception();
        System.out.println(e);
        Thread t = new Thread();
        System.out.println(t);
        List list = List.of(10,20,30,40,50);
        System.out.println(list);
```

```
    }  
}
```

OP:

```
D:\java6>javac Test.java
```

```
D:\java6>java Test
```

```
abc
```

```
java.lang.Exception
```

```
Thread[Thread-0,5,main]
```

```
[10, 20, 30, 40, 50]
```

In Java, there are two types of Objects.

1. Immutable Objects
2. Mutable Objects

Q)What is the difference between Immutable Objects and Mutable Objects?

Or

What is the difference between String and StringBuffer?

Ans:

Immutable objects are java objects, they will not allow modifications on their content, if we are trying to perform modifications on immutable objects data then immutable objects are allowing modifications but the resultant modified data will not be stored back in the original objects, where the resultant modified data will be stored by creating a new object for the same class.

EX: String class objects are immutable.

All Wrapper classes objects are immutable.

Mutable objects are the java objects, they are able to allow modifications on their content directly.

EX: All Java objects are by default mutable objects.

EX: StringBuffer

EX:

```
import java.util.*;
class Test{
    public static void main(String[] args){
        String str1 = new String("Durga ");
        String str2 = str1.concat("Software ");
        String str3 = str2.concat("Solutions");
        System.out.println(str1);// Durga
        System.out.println(str2);// Durga Software
        System.out.println(str3);// Durga Software Solutions
        System.out.println();

        StringBuffer sb1 = new StringBuffer("Durga ");
        StringBuffer sb2 = sb1.append("Software ");
        StringBuffer sb3 = sb2.append("Solutions");
        System.out.println(sb1);// Durga Software Solutions
        System.out.println(sb2);// Durga Software Solutions
        System.out.println(sb3);// Durga Software Solutions
    }
}
```

OP:

Durga
Durga Software
Durga Software Solutions

Durga Software Solutions
Durga Software Solutions
Durga Software Solutions

```

String str1 = new String("Durga ");
String str2 = str1.concat("Software ");

String str3 = str2.concat("Solutions");
Sopln(str1); Durga
Sopln(str2); Durga Software
Sopln(str3); Durga Software Solutions

```

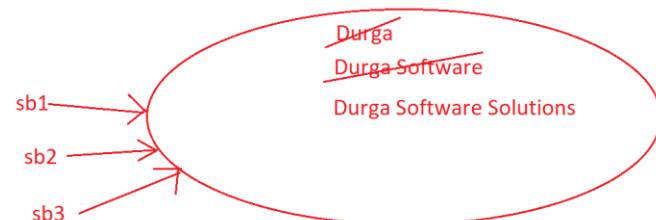


```

StringBuffer sb1 = new StringBuffer("Durga ");
StringBuffer sb2 = sb1.append("Software ");

StringBuffer sb3 = sb2.append("Solutions");
Sopln(sb1);// Durga Software Solutions
Sopln(sb2);//Durga Software Solutions
Sopln(sb3);//Durga Software Solutions

```



Q) What is the difference between Object and Instance?

Ans:

Object is a memory block to store data.

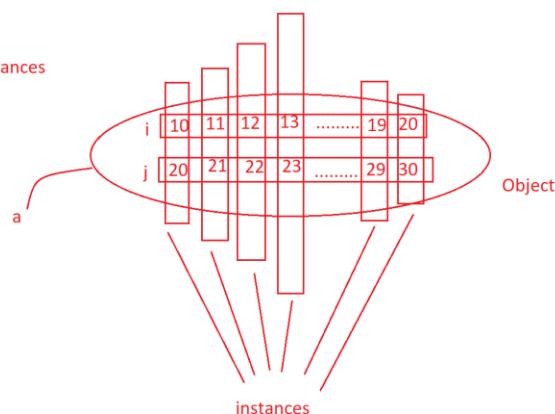
Instance is the layer of data or copy of data that existed at a particular point of time in an object.

```

class A{
    int i = 10;
    int j = 20;
}
class Test{
    public static void main(String[] args){
        A a = new A();
        for(int x = 0; x < 10; x++){
            a.i+=a.i+1;
            a.j+=a.j+1;
        }
    }
}

```

one Object
nearly 11 instances



Constructors:

1. Constructor is a Java feature, it can be used to create objects.
2. The role of the constructor in object creation is to provide initial values inside the objects.
3. In Java applications, constructors are executed exactly at the time of creating objects only, not before creating objects and not after creating objects.
4. In java applications, constructors are utilized to provide initializations to the class level variables.
5. Constructors names must be the same as the respective class name.
6. Constructors are not having return types.
7. Constructors are not allowing the access modifiers like static , final, abstract,.....
8. Constructors are able to allow the access modifiers like public, protected, <default> and private.
9. Constructors are able to allow throws keyword to bypass the exception from the present constructor to the caller of the constructor.

Syntax:

```
[AccessModifier] ClassName([ParamList])[throws ExceptionList]{  
    ----instructions----  
}
```

In java applications, constructor name must be same as the respective class name, in the case if we provide different name to the constructor not same as the class name then compiler will raise an error like “Invalid Method declaration, return type required”, because the compiler treats the provided constructor as normal java method and it is checking the provided constructor syntax against to the method syntax, in methods return type is mandatory.

EX:

```
class A{
    B(){
        System.out.println("A-Con");
    }
}
class Test{
    public static void main(String[] args){
        A a = new A();
    }
}
```

Status: Compilation Error.

In Java applications, Constructors do not have return types, in the case if we provide a return type to the constructor then the provided constructor will be converted to a java method and if we access that constructor as like a java method then it will be executed.

EX:

```
class A{
    void A(){
        System.out.println("A-Con");
    }
}
class Test{
    public static void main(String[] args){
        A a = new A();
        a.A();
    }
}
```

In Java applications, Constructors are not allowing the access modifiers like static, final, abstract,....., in the case if we provide the access modifiers like static, final, abstract,... to the constructor then the compiler will raise an error like “Modifier XXX not allowed here”.

EX:

```
class A{
    static A(){
        System.out.println("A-Con");
    }
}
class Test{
    public static void main(String[] args){
        A a = new A();
    }
}
```

Status: Compilation Error.

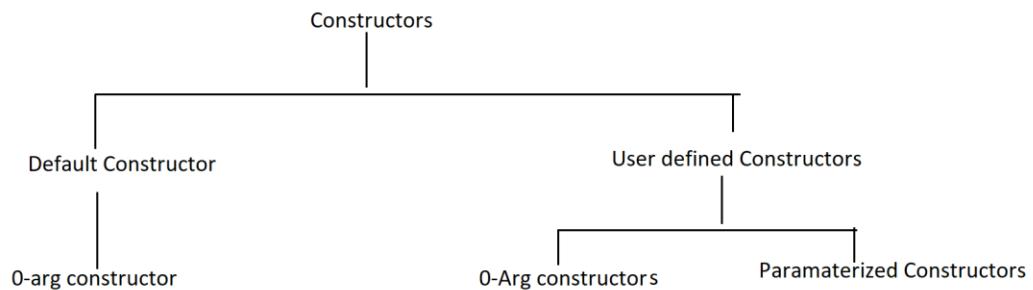
In Java applications, constructors are able to allow the access modifiers like public, protected, <default> and private , in the case if we declare a constructor as private then it must be accessed within the same class, that is, objects for the class must be created inside the same class, not possible to create objects in outside of the respective class.

EX:

```
class A{
    private A(){
        System.out.println("A-Con");
    }
}
class Test{
    public static void main(String[] args){
        A a = new A();
    }
}
```

Status: Compilation Error

In Java, constructors are divided into the following two types.



Default Constructor:

Default constructor is a 0-arg constructor, it will be provided by the compiler when we have not provided any constructor explicitly in the class, if we provide at least one constructor in the class then the compiler will not provide default constructor.

The default constructor will have the same scope of the respective class.

EX:

Test.java

```
public class Test{  
}
```

```
D:\java6>javac Test.java
```

```
D:\java6>javap Test
Compiled from "Test.java"
public class Test extends java.lang.Object{
    public Test();
}
```

Note: In Java, by default , all default constructors are 0-arg constructors, but all 0-arg constructors need not be default constructors, some 0-arg constructors that are provided by the compiler are default constructor, some other 0-arg constructors which are provided by the developers are user defined constructors.

User Defined Constructors:

These are the constructors provided by the developers as per their application requirement.

There are two types of User defined constructors.

1. 0-arg constructors
2. Parameterized Constructors

If we declare any constructor without the parameters then that constructor is a 0-arg constructor.

If we declare any constructor with at least one parameter then that constructor is a Parameterized constructor.

EX:

```
class Account{

    String accNo;
    String accHolderName;
    String accType;
    int balance;

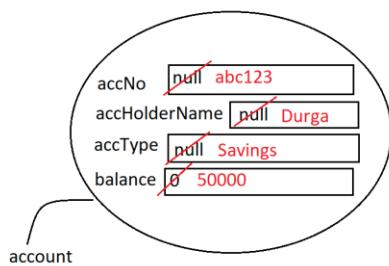
    public void setAccountDetails(){
        accNo = "abc123";
    }
}
```

```

        accHolderName = "Durga";
        accType = "Savings";
        balance = 50000;
    }
    public void displayAccountDetails(){
        System.out.println("Account Details");
        System.out.println("-----");
        System.out.println("Account Number : "+accNo);
        System.out.println("Account Holder Name : "+accHolderName);
        System.out.println("Account Type : "+accType);
        System.out.println("Account Balance : "+balance);
    }
}
class Test{

    public static void main(String[] args){
        Account account = new Account();
        account.setAccountDetails();
        account.displayAccountDetails();
    }
}

```



In the above program, we have created object for the class `Account` by executing default constructor which was provided by the compiler, at the time of creating object default values are

stored for the instance variables in the Account object, when we access setAccountDetails() method original values are stored for the variables inside the object as second data inside the object, not as first data.

As per the application requirement , we want to provide original data as first data at the time of creating object , not after creating object, to achieve this requirement we have to use constructor in place of setAccountDetails() method

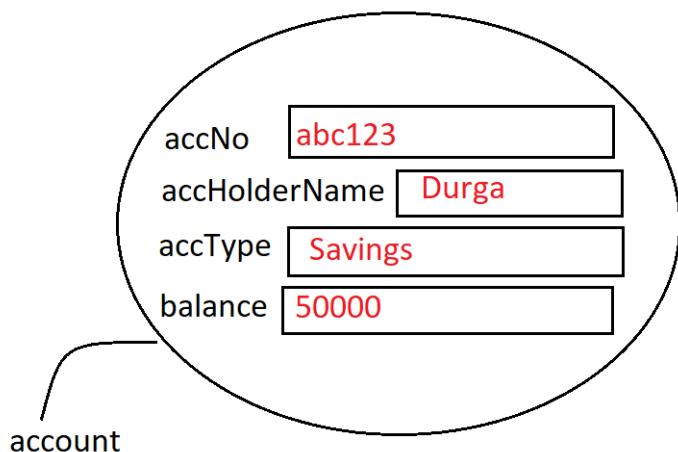
EX:

```
class Account{  
  
    String accNo;  
    String accHolderName;  
    String accType;  
    int balance;  
  
    public Account(){  
        accNo = "abc123";  
        accHolderName = "Durga";  
        accType = "Savings";  
        balance = 50000;  
    }  
    public void displayAccountDetails(){  
        System.out.println("Account Details");  
        System.out.println("-----");  
        System.out.println("Account Number : "+accNo);  
        System.out.println("Account Holder Name : "+accHolderName);  
        System.out.println("Account Type : "+accType);  
        System.out.println("Account Balance : "+balance);  
    }  
}  
class Test{
```

```

public static void main(String[] args){
    Account account = new Account();
    account.displayAccountDetails();
}

```



In the above application, if we create multiple accounts in a bank application, that is if we create multiple account objects then all the account objects are having the same data.

EX:

```

class Account{

    String accNo;
    String accHolderName;
    String accType;
    int balance;

    public Account(){
        accNo = "abc123";
    }
}

```

```
        accHolderName = "Durga";
        accType = "Savings";
        balance = 50000;
    }
    public void displayAccountDetails(){
        System.out.println("Account Details");
        System.out.println("-----");
        System.out.println("Account Number : "+accNo);
        System.out.println("Account Holder Name : "+accHolderName);
        System.out.println("Account Type : "+accType);
        System.out.println("Account Balance : "+balance);
    }
}
class Test{

    public static void main(String[] args){
        Account account1 = new Account();
        account1.displayAccountDetails();
        System.out.println();

        Account account2 = new Account();
        account2.displayAccountDetails();
        System.out.println();

        Account account3 = new Account();
        account3.displayAccountDetails();
    }
}
```

D:\java6>javac Test.java

D:\java6>java Test
Account Details

Account Number : abc123
Account Holder Name : Durga
Account Type : Savings
Account Balance : 50000

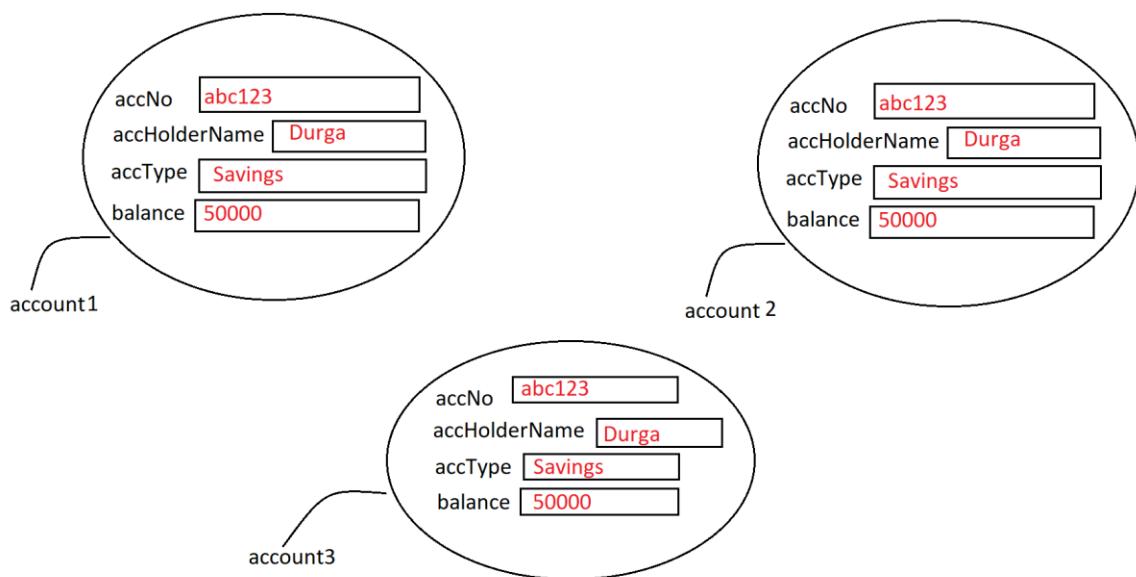
Account Details

Account Number : abc123
Account Holder Name : Durga
Account Type : Savings
Account Balance : 50000

Account Details

Account Number : abc123
Account Holder Name : Durga
Account Type : Savings
Account Balance : 50000

D:\java6>



As per the application requirement , we want to provide different data in different objects of the same Account class, to achieve this requirement we have to use “Parameterized Constructor”.

EX:

```
class Account{

    String accNo;
    String accHolderName;
    String accType;
    int balance;

    public Account(String acc_No, String acc_Holder_Name, String
acc_Type, int acc_Balance){
        accNo = acc_No;
        accHolderName = acc_Holder_Name;
        accType = acc_Type;
        balance = acc_Balance;
    }
    public void displayAccountDetails(){
        System.out.println("Account Details");
        System.out.println("-----");
        System.out.println("Account Number : "+accNo);
        System.out.println("Account Holder Name : "
"+accHolderName);
        System.out.println("Account Type : "+accType);
        System.out.println("Account Balance : "+balance);
    }
}

class Test{

    public static void main(String[] args){
        Account account1 = new Account("a111", "Durga",
"Savings", 50000);
        account1.displayAccountDetails();
        System.out.println();
    }
}
```

```
        Account account2 = new Account("a222", "Vishnu",
"Savings", 60000);
        account2.displayAccountDetails();
        System.out.println();

        Account account3 = new Account("a333", "Ramesh",
"Savings", 70000);
        account3.displayAccountDetails();
    }
}
```

D:\java6>javac Test.java

D:\java6>java Test

Account Details

Account Number : a111
Account Holder Name : Durga
Account Type : Savings
Account Balance : 50000

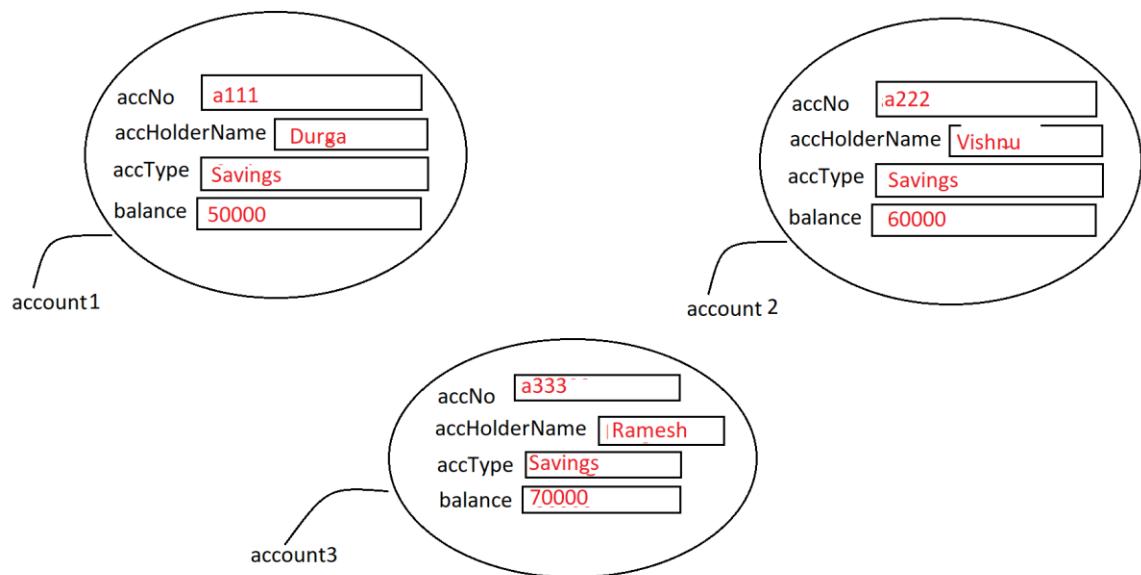
Account Details

Account Number : a222
Account Holder Name : Vishnu
Account Type : Savings
Account Balance : 60000

Account Details

Account Number : a333
Account Holder Name : Ramesh
Account Type : Savings
Account Balance : 70000

D:\java6>



Q) What is Constructor Overloading?

Ans:

If we declare more than one method with the same name and with the different parameter list then it is called “Method Overloading”.

Similarly, If we declare more than one constructor with the same name and with the different parameter list then it is called “Constructor Overloading”.

EX:

```
class Calculator{
    int i, j, k;

    Calculator(){
    }
    Calculator(int x){}
```

```
i = x;
}
Calculator(int x, int y){
    i = x;
    j = y;
}
Calculator(int x, int y, int z){
    i = x;
    j = y;
    k = z;
}
public void add(){
    System.out.println("ADD : "+(i+j+k));
}

}
class Test{

public static void main(String[] args){
    Calculator cal1 = new Calculator();
    cal1.add();

    Calculator cal2 = new Calculator(10);
    cal2.add();

    Calculator cal3 = new Calculator(10,20);
    cal3.add();

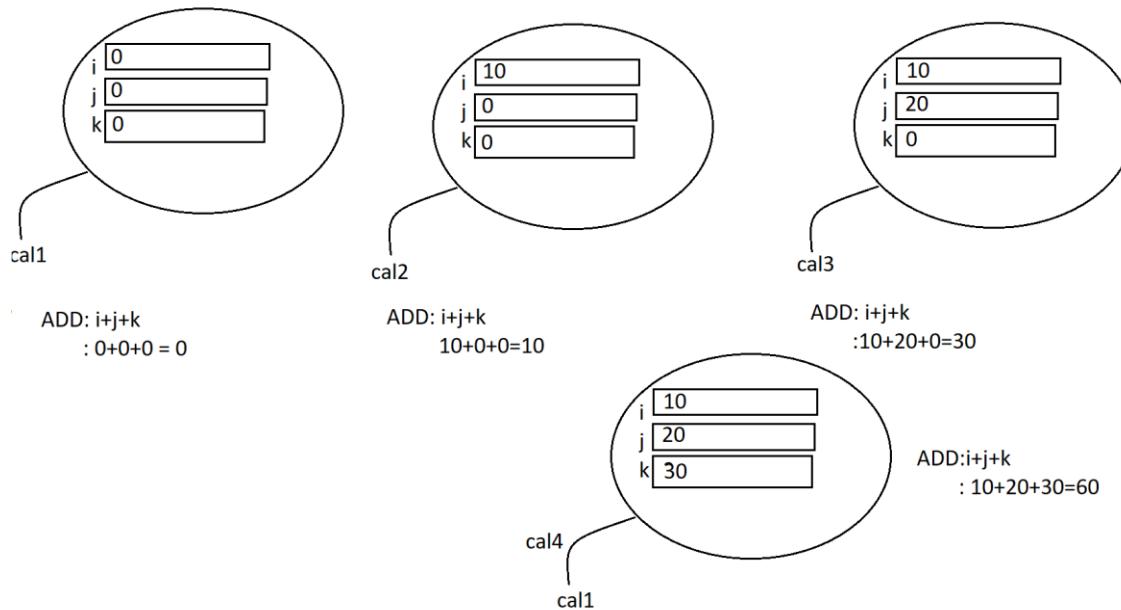
    Calculator cal4 = new Calculator(10,20,30);
    cal4.add();
}
}
```

D:\java6>javac Test.java

D:\java6>java Test

```
ADD : 0  
ADD : 10  
ADD : 30  
ADD : 60
```

D:\java6>



Q) What are the advantages of IDE?

Ans:

- 1. IDE is “Integrated Development Environment”.
- 2. Not required to use command prompt to compile and to execute the program.
- 3. Auto-compilation , when we type the program , automatically compilation is going on internally, no need to perform compilation explicitly.
- 4. API help, If we type one or two letters of any class name or interfaces... automatically matched classes names and interfaces names are available in a list to select.

5. Inbuilt Debugging Tool, it helps us to debug the code , that is to trace the code.
 6. Inbuilt support for the Database integration, we can integrate the databases like Oracle, MySQL,...With this, we can open oracle , Mysql databases through Eclipse IDE.
 7. Inbuilt support for Servers integration, we can integrate the servers like Tomcat, weblogic, Wildfly,.... For server side programming.
- -----

EX: Eclipse, IntelliJ Idea, Netbeans,

Eclipse IDE:

-
1. Download And Install Eclipse IDE
 2. Open Eclipse IDE and Create a Java Project.
 3. Create a Main class and main() method.
 4. Provide application Logic logic
 5. Execute the application

Instance Context / Instance Flow of Execution:

In Java applications, when we load a class bytecode to the memory ,automatically a separate context will be created called “Static Context” to perform all the activities while loading class bytecode.

In Java applications, when we create an object at heap memory ,automatically a separate context will be created called “Instance Context” to perform all the activities while creating and initializing the object.

In Java applications, instance context is represented in the form of the following three elements.

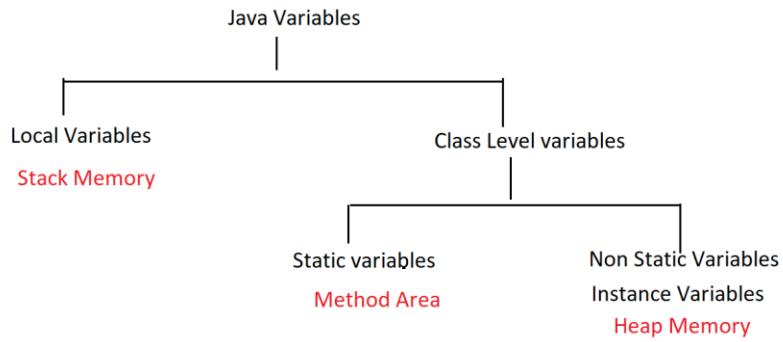
1. Instance Variables
2. Instance Methods
3. Instance Blocks

Instance Variables:

1. If any variable changes its value from one instance to another instance of an object then that variable is called “Instance Variable”.
2. In Java applications, we are able to declare instance variables at class level as non-static variables.
3. In Java applications, instance variables are recognized and initialized just before executing the respective class constructor automatically.
4. In the current class we can access instance variables directly without creating an object, but if we access instance variables outside of the current class then we have to create an object for the respective class and we have to use the generated reference variable.

Note: If we access an instance variable by using a reference variable containing null value then JVM will raise an exception like `java.lang.NullPointerException`.

5. In Java applications, instance variables data will be stored inside the objects at heap memory.
6. In Java applications, a separate copy of the instance variable data will be created for each and every object, if we perform modifications on an instance variable in an object then that modification is available up to the respective object, that modifications will not be reflected to the other objects.



EX:

```

package com.durgasoft.test;

class A{

    int i = 10;

}

public class Test {

    public static void main(String[] args) {

        A a1 = new A();

        System.out.println(a1.i); //10

        A a2 = new A();

        System.out.println(a2.i); //10
    }
}
  
```

```
A a3 = new A();  
  
System.out.println(a3.i); //10  
  
System.out.println();  
  
  
  
  
a1.i = 20;  
  
System.out.println(a1.i); // 20  
  
System.out.println(a2.i); // 10  
  
System.out.println(a3.i); // 10  
  
  
  
  
A a = null;  
  
System.out.println(a.i);  
  
}  
  
}  
  
10  
  
10  
  
10  
  
20  
  
10
```

10

Exception in thread "main"
java.lang.NullPointerException: Cannot read field "i"
because "a" is null

at app01/com.durgasoft.test.Test.main(Test.java:22)

Instance Method:

-
1. Instance method is a Java method, it is a set of instructions representing a particular action of an entity.
 2. Instance methods are recognized and executed at the moment when we access that method.
 3. If we declare any non-static method in a class then that method is an instance method.
 4. Within the same class we can access instance methods directly without using any reference variable, without using any keyword and without creating objects, but if we want to access instance methods outside of the current class then we have to create an object for the respective class and we have to use the generated reference variable.

Note: If we access any instance method by using a reference variable containing null value then JVM will raise an exception like `java.lang.NullPointerException`.

EX:

```
package com.durgasoft.test;  
  
class A{  
  
    void m1() {
```

```
        System.out.println("m1-A");

    }

}

public class Test {

    public static void main(String[] args) {

        A a = new A();

        a.m1();

        A a1 = null;

        a1.m1();

    }

}
```

m1-A

Exception in thread "main"
java.lang.NullPointerException: Cannot invoke
"com.durgasoft.test.A.m1()" because "a1" is null

at app01/com.durgasoft.test.Test.main(Test.java:14)

EX:

```
package com.durgasoft.test;
```

```
class A{  
    int i = m1();  
  
    A(){  
  
        System.out.println("A-Con");  
  
    }  
  
    int m1() {  
  
        System.out.println("m1-A");  
  
        return 10;  
  
    }  
  
}  
  
public class Test {  
  
    public static void main(String[] args) {  
  
        A a = new A();  
  
    }  
  
}
```

OP:

m1-A

A-Con

Instance Block:

-
1. It is a set of instructions , it will be recognized and executed automatically just before executing the respective class constructor like instance variables.
 2. Instance blocks are having the same power of constructors but it is not utilized like constructors in the application development.

Syntax:

```
{  
    ----instructions----  
}
```

EX:

```
package com.durgasoft.test;  
  
class A{  
  
}  
  
    System.out.println("IB-A");  
  
}  
  
A(){  
  
    System.out.println("A-Con");  
  
}  
  
}
```

```
public class Test {  
    public static void main(String[] args) {  
        A a = new A();  
    }  
}
```

OP:

IB-A

A-Con

EX:

```
package com.durgasoft.test;  
  
class A{  
    A(){  
        System.out.println("A-Con");  
    }  
    int m1() {  
        System.out.println("m1-A");  
        return 10;  
    }  
}
```

```
{  
    System.out.println("IB-A");  
}  
  
int i = m1();  
  
}  
  
public class Test {  
    public static void main(String[] args) {  
        A a = new A();  
    }  
}
```

OP:

IB-A

m1-A

A-Con

EX:

```
package com.durgasoft.test;  
  
class A{  
    int i = m1();
```

```
A(){  
    System.out.println("A-Con");  
}  
{  
    System.out.println("IB-A");  
}  
  
int m1() {  
    System.out.println("m1-A");  
    return 10;  
}  
}  
  
public class Test {  
    public static void main(String[] args) {  
        A a1 = new A();  
        A a2 = new A();  
    }  
}
```

OP:

m1-A

IB-A

A-Con

m1-A

IB-A

A-Con

EX:

```
package com.durgasoft.test;

class A{

    int i = m1();

    {

        System.out.println("IB1-A");

    }

    int m2() {

        System.out.println("m2-A");

        return 20;

    }

    A(){
}
```

```
System.out.println("A-Con");  
}  
  
int j = m2();  
  
{  
  
System.out.println("IB2-A");  
}  
  
int m1() {  
  
System.out.println("m1-A");  
  
return 10;  
}  
  
}  
  
public class Test {  
  
public static void main(String[] args) {  
  
A a = new A();  
}  
  
}  
  
OP:  
  
m1-A
```

IB1-A

m2-A

IB2-A

A-Con

'this' Keyword:

'this' is a Java keyword, it is able to represent the current class object.

In Java applications, we are able to utilize **'this'** keyword in the following four ways.

1. To refer current class variables
2. To refer current class methods
3. To refer current class constructor
4. To return Current class Object

To Refer Current class Variables:

To refer to current class variables if we want to use **'this'** then we have to use the following syntax.

this.varName

In Java applications, if we have the same variables at local and at class level, where to refer class level variable over the local variable there we will use **'this'** keyword.

EX:

```
package com.durgasoft.test;

class A{

    int i = 10;

    int j = 20;

    A(int i, int j){

        System.out.println(i+" "+j);

        System.out.println(this.i+" "+this.j);

    }

}

public class Test {

    public static void main(String[] args) {

        A a = new A(30,40);

    }

}

OP:
```

30 40

10 20

In general, in java bean classes, we have to declare a separate set of setXXX() method and getXXX() method for each and every class level variable, in general, in the setXXX() method will use a parameter with the same name of the respective class level variable, inside the setXXX() method we have to assign parameter variable value to the respective class level variable , so here we have to refer class level variable over the local variable , here to refer class level variable over the local variable we have to use “this” keyword.

EX:

Employee.java

```
package com.durgasoft.beans;
```

```
public class Employee {
```

```
    private int eno;
```

```
    private String ename;
```

```
    private float esal;
```

```
    private String eaddr;
```

```
    public int getEno() {
```

```
        return eno;
```

```
}

public void setEno(int eno) {

    this.eno = eno;

}

public String getName() {

    return ename;

}

public void setName(String ename) {

    this.ename = ename;

}

public float getEsal() {

    return esal;

}

public void setEsal(float esal) {

    this.esal = esal;

}

public String getEaddr() {

    return eaddr;
```

```
}

public void setEaddr(String eaddr) {

    this.eaddr = eaddr;

}

}
```

Test.java

```
package com.durgasoft.test;

import com.durgasoft.beans.Employee;

public class Test {

    public static void main(String[] args) {

        Employee employee = new Employee();

        employee.setEno(111);

    }

}
```

```

employee.setEname("Durga");

employee.setEsal(50000.0f);

employee.setEaddr("Hyd");

System.out.println("Employee Details");

System.out.println("-----");

System.out.println("Employee Number : "+employee.getEno());

System.out.println("Employee Name : "+employee.getEname());

System.out.println("Employee Salary : "+employee.getEsal());

System.out.println("Employee Address : "+employee.getEaddr());

}

}

```

To refer Current class methods by using this keyword:

If we want to refer to the current class method by using ‘this’ keyword then we have to use the following syntax.

```
this.methodName([ParamValues]);
```

Note: In the current class, to access current class methods it is not required to use any keyword , directly we can access current class methods without using reference variables and without using 'this' keyword.

EX:

```
package com.durgasoft.test;
```

```
class A{
```

```
    void m1() {
```

```
        System.out.println("m1-A");
```

```
        m2();
```

```
        this.m2();
```

```
}
```

```
    void m2() {
```

```
        System.out.println("m2-A");
```

```
}
```

```
}
```

```
public class Test {
```

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

```
A a = new A();  
a.m1();  
}  
}
```

Output:

m1-A

m2-A

m2-A

To refer current class constructors by using ‘this’ keyword:

If we want to access the current class constructor by using this keyword then we have to use the following syntax.

this([ParamValues]);

EX:

```
class A{  
    A(){  
        this(10);  
        System.out.println("A-Con");  
    }  
    A(int i){  
        this(22.22f);  
        System.out.println("A-int-param-con");  
    }  
}
```

```

    }
    A(float f){
        this(33.333);
        System.out.println("A-float-param-con");
    }
    A(double d){
        System.out.println("A-double-param-con");
    }
}
class Test{
    public static void main(String[] args){
        A a = new A();
    }
}

```

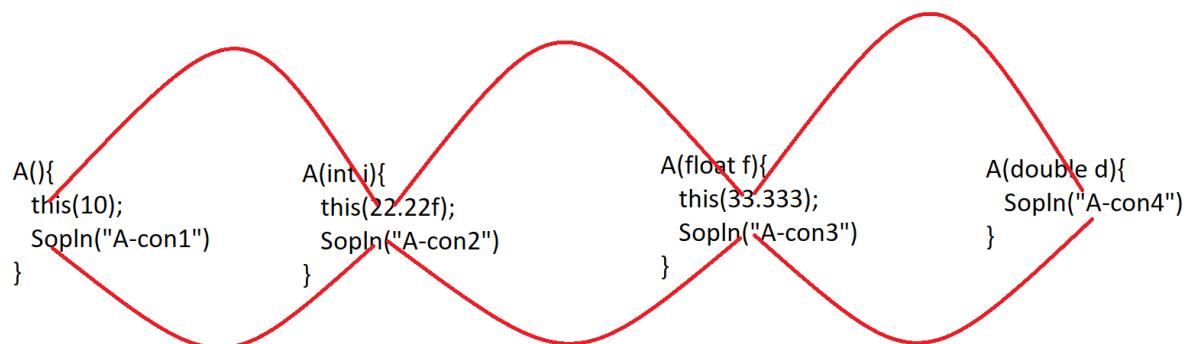
Output:

```

A-double-param-con
A-float-param-con
A-int-param-con
A-Con

```

In the above program, all the current constructors are executed in a chain fashion by using this keyword , so this feature is called “Constructor Chaining”.



In the above program we have written more than one constructor with the same name and with a different parameter list then this feature is called “Constructor Overloading”.

If we want to refer to the current class constructor by using ‘this’ keyword then the respective this statement must be the first statement .

EX:

```
class A{
    A(){
        System.out.println("A-Con");
        this(10);
    }
    A(int i){
        System.out.println("A-int-param-con");
        this(22.22f);
    }
    A(float f){
        System.out.println("A-float-param-con");
        this(33.3333);
    }
    A(double d){
        System.out.println("A-double-param-con");
    }
}
class Test{
    public static void main(String[] args){
        A a = new A();
    }
}
```

Status: Compilation Error.

If we want to refer to the current class constructor by using ‘this’ keyword then we have to use the respective this statement in the other current class constructor only, not in the normal java method.

EX:

```
class A{
    A(){
        System.out.println("A-Con");
    }
    A(int i){
        System.out.println("A-int-param-con");
    }
    void m1(){
        this(10);
        System.out.println("m1-A");
    }
}
class Test{
    public static void main(String[] args){
        A a = new A();
        a.m1();
    }
}
```

Status: Compilation Error

Q) Is it possible to refer to more than one current class constructor by using this keyword from a single current class constructor?

Ans:

No, it is not possible to refer to more than one current class constructor by using this keyword from a single current class

constructor, because in the constructors this statement must be the first statement.

EX:

--

```
class A{
    A(){
        this(22.22f);
        this(33.3333);
        System.out.println("A-Con");
    }
    A(int i){
        System.out.println("A-int-param-con");
    }
    A(float f){
        System.out.println("A-float-param-con");
    }
}
class Test{
    public static void main(String[] args){
        A a = new A();
    }
}
```

Status: Compilation Error

To return Current Class Object from the method:

If we want to return a current class object by using this keyword then we have to use the following syntax.

```
return this;
```

EX:

```
package com.durgasoft.test;

class A{

    A getRef1(){

        A a = new A();

        return a;

    }

    A getRef2() {

        return this;

    }

}

class Test{

    public static void main(String[] args){

        A a = new A();

        System.out.println(a); // A@a111

        System.out.println();

        System.out.println(a.getRef1()); // A@a222

        System.out.println(a.getRef1()); // A@a333

    }

}
```

```
System.out.println(a.getRef1()); // A@a444
System.out.println();
System.out.println(a.getRef2()); // A@a111
System.out.println(a.getRef2()); // A@a111
System.out.println(a.getRef2()); // A@a111
}

}
```

OP:

```
com.durgasoft.test.A@626b2d4a
com.durgasoft.test.A@5e265ba4
com.durgasoft.test.A@156643d4
com.durgasoft.test.A@123a439b
com.durgasoft.test.A@626b2d4a
```

com.durgasoft.test.A@626b2d4a

com.durgasoft.test.A@626b2d4a

Note: In the above program , when we access getRef1() method every time JVM executes new keyword , so JVM will create new object every time , it will create duplicate objects for the class, in this context, if we want to return the same object on which we have accessed this method then we have to return ‘this’.

‘static’ Keyword:

‘static’ is a java keyword, it is able to improve shareability in java applications.

In Java applications, we are able to utilize static keyword in the following four ways.

1. Static Variables
2. Static Methods
3. Static Blocks
4. Static Import

Static Variables:

-
1. Static variable is a java variable, it will be recognized and initialized at the time of loading the respective class bytecode to the memory.
 2. Static variable is a java variable, it will share its last modified value to the past objects[Previous Objects] which we have already created and to the future objects which we are going to create.
 3. Static variables must be declared at class level only, it is not possible to declare static variables as local variables. If we declare any static variable as a local variable then the compiler will raise an error.

4. In Java applications, we are able to access static variables either by creating object for the respective class or by using the respective class name directly. It is suggestible to access static variables by using class names.

Note: In Java applications, if we access any instance variable by using a reference variable containing null value then JVM will raise an exception like `java.lang.NullPointerException`, but if we access any static variable by using a reference variable containing null value then JVM will not raise any exception, here JVM will get the value of the static variable.

5. In Java applications, we are able to use ‘this’ keyword to access current class static variable.

6. Static variables data will be stored in the Method Area.

EX:

```
package com.durgasoft.test;

class A{

    static int i = 10;

    int j = 20;

    void m1() {

        //static int k = 30; --> Error

        System.out.println("m1-A");

        System.out.println(this.i);

    }
}
```

```
}
```

```
class Test{
```

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

```
        A a = new A();
```

```
        System.out.println(a.i);
```

```
        System.out.println(A.i);
```

```
    A a1 = null;
```

```
    //System.out.println(a1.j);-->java.lang.NullPointerException
```

```
        System.out.println(a1.i);
```

```
        a.m1();
```

```
}
```

```
}
```

EX:

```
package com.durgasoft.test;

class A{

    int i = 10;

    static int j = 10;

}

class Test{

    public static void main(String[] args){

        A a1 = new A();

        System.out.println(a1.i+" "+a1.j);

        a1.i = a1.i+ 1;

        a1.j = a1.j + 1;

        System.out.println(a1.i+" "+a1.j);

        A a2 = new A();

        System.out.println(a2.i+" "+a2.j);

        a2.i = a2.i + 1;

        a2.j = a2.j + 1;

        System.out.println(a1.i+" "+a1.j);

    }

}
```

```
System.out.println(a2.i+" "+a2.j);

A a3 = new A();

System.out.println(a3.i+" "+a3.j);

a3.i = a3.i + 1;

a3.j = a3.j + 1;

System.out.println(a1.i+" "+a1.j);

System.out.println(a2.i+" "+a2.j);

System.out.println(a3.i+" "+a3.j);

}

}

Output:

10 10

11 11

10 11

11 12
```

11 12

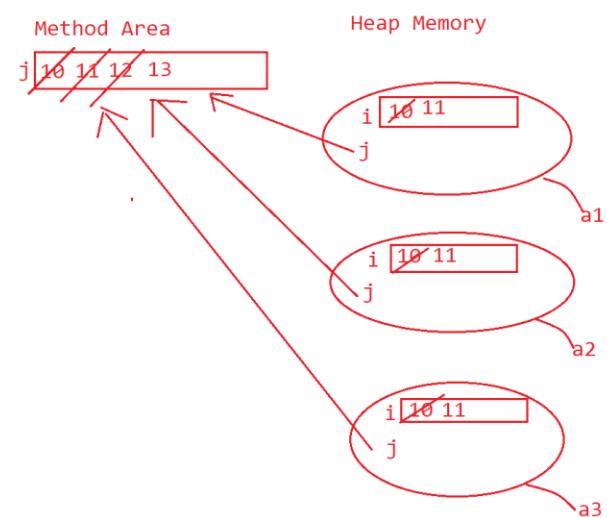
10 12

11 13

11 13

11 13

```
class A{
    int i = 10;
    static int j = 10;
}
class Test{
    public static void main(String[] args){
        A a1 = new A();
        System.out.println(a1.i+" "+a1.j); 10 10
        a1.i = a1.i+1; a1.j = a1.j+1;
        System.out.println(a1.i+" "+a1.j); 11 11
        A a2 = new A();
        System.out.println(a2.i+" "+a2.j); 10 11
        a2.i = a2.i+1; a2.j = a2.j+1;
        System.out.println(a1.i+" "+a1.j); 11 12
        System.out.println(a2.i+" "+a2.j); 11 12
        A a3 = new A();
        System.out.println(a3.i+" "+a3.j); 10 12
        a3.i = a3.i+1; a3.j = a3.j+1;
        System.out.println(a1.i+" "+a1.j); 11 13
        System.out.println(a2.i+" "+a2.j); 11 13
        System.out.println(a3.i+" "+a3.j); 11 13
    }
}
```



Q) What are the differences between Instance variables and Static variables?

Ans:

1. To declare an instance variable no need to use any special keyword.

To declare a static keyword we need a ‘static’ keyword.

2. A separate copy of the instance variable will be created at each every object.

Single copy of the static variable will be shared to all the objects of the respective class.

3. The scope of the instance variable is up to a single object.

The scope of the static variable is up to all the objects of the respective class.

4. Instance variables will share their value to a single object, that is less shareability.

Static variables will share their values to all the objects, that is more shareability.

5. Instance variables data will be stored inside heap memory in the form of objects.

Static variables data will be stored inside the method area.

6. Instance variables are recognized and initialized just before executing the respective class constructor.

Static variables are recognized and initialized at the time of loading the respective bytecode to the memory.

7. To access instance variables we must create an object for the respective class.

To access static variables either we can create objects or we can use the respective class name directly.

8. `java.lang.NullPointerException` is possible in the case of instance variables.

`java.lang.NullPointerException` is not possible in the case of static variables.

9. We are able to access Instance variables inside the instance methods only , not possible to access instance variables inside the static methods.

We are able to access static variables in both static methods and instance methods.

Static Method:

Static method is a set of instructions , it will be recognized and executed the moment when we access that method.

Static methods are able to allow only static members of the current class , static methods are unable to allow instance members of the current class.

Note: If we want to access instance members inside the static method then we have to create an object for the respective class and we have to use the generated reference variable explicitly.

In Java applications, we are able to access static methods either by creating objects for the respective class or by using the respective class name directly, where class name is suggestive to access static methods.

Note: If we access an instance method by using a reference variable containing null value then JVM will raise an exception like `java.lang.NullPointerException`, but if we access a static method by using a reference variable containing null value then JVM will not raise any exception.

Static Methods are not allowing ‘this’ keyword in their body, but we are able to use ‘this’ keyword to access current class static methods.

EX:

```
package com.durgasoft.test;

class A{

    int i = 10;

    static int j = 20;

    static void m1() {

        System.out.println("m1-A");

        //System.out.println(i); --> Error

        System.out.println(j);

        //System.out.println(this.j); --> Error

    }

    void m2() {

        System.out.println("m2-A");

        System.out.println(i);

        System.out.println(j);

    }

}
```

```
this.m1();  
}  
}  
class Test{  
public static void main(String[] args){  
    A a = new A();  
    a.m1();  
    A.m1();  
  
    A a1 = null;  
    //a1.m2(); --> java.lang.NullPointerException  
    a1.m1();  
    a.m2();  
}  
}
```

Output:

m1-A

m1-A

20

m1-A

20

m2-A

10

20

m1-A

20

Q) Is it possible to display a line of text on command prompt without using the main() method?

Ans:

Yes, it is possible to display a line of text on command prompt without using the main() method, but we must use the “Static variable-Static Method” combination.

EX:

```
class Test{
    static int i = m1();
    static int m1(){
        System.out.println("Welcome To Durgasoft");
        System.exit(0); // To terminate the program abnormally.
        return 10;
    }
}
```

```
}
```

```
D:\java6>java6.bat
```

```
D:\java6>set path=C:\Java\jdk1.6.0_45\bin;
```

```
D:\java6>javac Test.java
```

```
D:\java6>java Test  
Welcome To Durgasoft
```

```
D:\java6>java7.bat
```

```
D:\java6>set path=C:\Java\jdk1.7.0_80\bin;
```

```
D:\java6>javac Test.java
```

```
D:\java6>java Test  
Error: Main method not found in class Test, please define the  
main method as:  
public static void main(String[] args)
```

```
D:\java6>
```

Internal Flow:

When we execute a Test class , JVM will perform the following actions.

1. JVM will load Test class bytecode to the memory.
2. AT the time of loading Test class bytecode, JVM will recognize and initialize static variables.
3. As part of static variable initialization, JVM will access the m1() method call, with this JVM will execute the m1() method.

4. As part of the m1() method execution, JVM will display the requirement message on command prompt.
5. As part of the m1() method execution, when JVM executes System.exit(0) JVM will terminate the program execution immediately.

The above question and answer are valid up to JAVA 6 version, they are invalid from JAVA 7 version, because

In JAVA 6 version JVM will load Main class bytecode to the memory without checking whether main() method existed or not in the main class, after loading main class JVM will check main() method availability.

From Java 7 version, first JVM will check whether main () method exist or not , if main() method exist then only JVM will load main class bytecode to the memory, if main() does not exist in the main class then JVM will not main class bytecode to the memory.

Static blocks:

-
1. Static block is a set of instructions , which are recognized and executed automatically at the time of loading the respective class bytecode to the memory.
 2. Static blocks are able to allow static members of the current class, static blocks are not allowing instance members of the current class.

Note: If we want to access instance members inside the static block then we have to create an object for the respective class and we have to use the generated reference variable.

3. Static blocks are not allowing ‘this’ keyword in their body.

Syntax:

```
static{
    -----
    -----
    -----

}

EX:

class A{
    int i = 10;
    static int j = 20;
    static{
        System.out.println("SB-A");
        //System.out.println(i);--> Error
        System.out.println(j);
        //System.out.println(this.j); ---> Error
    }
}
class Test{
    public static void main(String[] args){
        A a = new A();
    }
}
```

Q) Is it possible to display a line of text on command prompt without using the main() method, static variable and static method?

Ans:

Yes, It is possible to display a line of text on command prompt without using main(), static variable and static method , but we must use a static block.

EX:

--

```
class Test{
    static{
```

```
        System.out.println("Welcome To Durgasoft");
        System.exit(0);
    }
}
```

D:\java6>java6.bat

D:\java6>set path=C:\Java\jdk1.6.0_45\bin;

D:\java6>javac Test.java

D:\java6>java Test
Welcome To Durgasoft

D:\java6>java7.bat

D:\java6>set path=C:\Java\jdk1.7.0_80\bin;

D:\java6>javac Test.java

D:\java6>java Test
Error: Main method not found in class Test, please define the
main method as:
public static void main(String[] args)

D:\java6>

When we execute Test class, JVM will load Test class bytecode to the memory, at the time of loading Test class bytecode to the memory, JVM will execute the provided static block, it will display the required message on command prompt, when JVM executes System.exit(0) JVM will terminate the program abnormally without displaying any exception related messages.

The above question and answer are valid up to JAVA 6 version , these are invalid right from JAVA 7 version, because,

Up to JAVA 6 version JVM will load main class bytecode to the memory irrespective of the main() method availability.

From JAVA 7 version , First JVM will check whether main() method is available or not in main class, if main() method is available then only JVM will load main class bytecode to the memory, if main() method is not available then JVM will not load main class bytecode to the memory.

Q)Is it possible to display a line of text on command prompt without using the main() method, static variables , static method and static block?

Ans:

Yes, it is possible to display a line of text on command prompt without using main() method, static variable, static method and static block, but we must use “Static Anonymous inner class of Object class”.

EX:

```
class Test{
    static Object obj = new Object(){
        {
            System.out.println("Welcome To Durgasoft");
            System.exit(0);
        }
    };
}
```

```
D:\java6>java6.bat
```

```
D:\java6>set path=C:\Java\jdk1.6.0_45\bin;
```

```
D:\java6>javac Test.java
```

```
D:\java6>java Test  
Welcome To Durgasoft
```

```
D:\java6>java7.bat
```

```
D:\java6>set path=C:\Java\jdk1.7.0_80\bin;
```

```
D:\java6>javac Test.java
```

```
D:\java6>java Test  
Error: Main method not found in class Test, please define the  
main method as:  
public static void main(String[] args)
```

```
D:\java6>
```

Note: The above question and answer are valid up to JAVA 6 version, they are invalid from JAVA 7 version, because, in JAVA 6 version JVM will load main class bytecode to the memory without checking main() method availability, but from JAVA 7 version onwards JVM will load main class bytecode to the memory after checking main() method availability and after getting confirmation of the main() method availability in main class.

Static import:

In general, in java applications, if we have any class in a package and if we want to use that class in any java file then we have to import the respective package to the present file and we have to use the respective class in the java file.

EX:

```
import com.durgasoft.core.*;
import java.util.ArrayList;
import java.io.BufferedReader;
-----
-----
```

If we want to import the static members[not instance members] of a particular class to the present java file then we have to use “Static import”.

If we import static members of a particular class by using “static import” in the present java file then we are able to access that static members without using class name and without using reference variables of the respective class, we can access that static members as local members.

Syntax#1:

```
import static packageName.className.*;
```

It is able to import all static members of the specified class from the specified package.

Syntax#2:

```
import static packageName.className.staticMember;
```

It is able to import the specified static member from the specified package.

EX:

In `java.lang.Thread` class , there are three static variables.

```
public static final int MIN_PRIORITY = 1;
public static final int NORM_PRIORITY = 5;
public static final int MAX_PRIORITY = 10;
```

EX:

In `java.lang.System` class, the variables like `out` , `err`, `in` are static members

EX:

```
import static java.lang.Thread.*;
import static java.lang.System.out;
class Test{
    public static void main(String[] args){
        out.println(MIN_PRIORITY);
        out.println(NORM_PRIORITY);
        out.println(MAX_PRIORITY);
    }
}
```

D:\java6>javac Test.java

D:\java6>java Test
1
5
10

D:\java6>

Static Context / Static Flow of execution:

Static context is represented by the following three elements.

1. Static Variables
2. Static Methods
3. Static Blocks

Where static variables and static blocks are recognized and executed automatically at the time of loading the respective class bytecode to the memory, where static methods are recognized and executed the moment when we access that method.

EX:

package com.durgasoft.test;

class A{

```
static {  
    System.out.println("SB-A");  
}  
  
static int m1() {  
    System.out.println("m1-A");  
    return 10;  
}  
  
static int i = m1();  
}  
  
class Test{  
    public static void main(String[] args){  
        A a = new A();  
    }  
}
```

OP:

SB-A

m1-A

EX:

```
package com.durgasoft.test;

class A{

    static int i = m1();

    static {

        System.out.println("SB-A");

    }

    static int m1() {

        System.out.println("m1-A");

        return 10;

    }

}

class Test{

    public static void main(String[] args){

        A a1 = new A();

        A a2 = new A();

    }

}

OP:
```

m1-A

SB-A

EX:

```
package com.durgasoft.test;

class A{
    A(){
        System.out.println("A-Con");
    }
    static {
        System.out.println("SB-A");
    }
    int m1() {
        System.out.println("m1-A");
        return 10;
    }
    static int i = m2();
    {
        System.out.println("IB-A");
    }
    int j = m1();
    static int m2() {
        System.out.println("m2-A");
        return 20;
    }
}
class Test{
    public static void main(String[] args){
        A a = new A();
    }
}
```

SB-A

m2-A

IB-A

m1-A

A-Con

EX:

```
package com.durgasoft.test;

class A{

    System.out.println("IB-A");

}

int m1() {

    System.out.println("m1-A");
}
```

```
    return 10;

}

int i = m1();

A(){

    System.out.println("A-Con");

}

static {

    System.out.println("SB-A");

}

static int m2() {

    System.out.println("m2-A");

    return 20;

}

static int j = m2();

}

class Test{
```

```
public static void main(String[] args){  
    A a1 = new A();  
    A a2 = new A();  
}  
}
```

SB-A

m2-A

IB-A

m1-A

A-Con

IB-A

m1-A

A-Con

Class.forName():

Consider the following program.

```
package com.durgasoft.test;
```

```
class A{
```

```
    static {
```

```

        System.out.println("Class Loading....");
    }

    A(){
        System.out.println("Object Creating.....");
    }

}

class Test{

    public static void main(String[] args){

        A a = new A();
    }

}

```

When we create an object for a particular class, JVM will perform the following two tasks automatically.

1. Loading the respective class bytecode to the memory.
2. Creating object for the respective class.

As per the requirement , we want to load a particular class bytecode to the memory , not to create any object for the respective loaded class , to achieve this requirement we have to use the following method from java.lang.Class.

```
public static Class.forName(String className) throws
ClassNotFoundException
```

EX: Class cls = Class.forName("Employee");

EX:

```
package com.durgasoft.test;

class A{

    static {
        System.out.println("Class Loading....");
    }

    A(){
        System.out.println("Object Creating.....");
    }
}

class Test{

    public static void main(String[] args) throws
ClassNotFoundException{
    Class cls =
    Class.forName("com.durgasoft.test.A");

}
}
```

When we execute the above instruction , JVM will perform the following actions.

1. JVM will take the provided class name from `forName()` method.
2. JVM will search for the respective class's `.class` file at the following locations.
 - a. At the current location from where we are executing the program.
 - b. At the java predefined library
 - c. At the locations referred by “classpath” environment variable.
3. If the required `.class` file is not available at all the above locations
then JVM will raise an exception like
“`java.lang.ClassNotFoundException`”.
4. If the required `.class` file is available at either of the above locations
then JVM will load its bytecode to the memory.
5. When a class bytecode is loaded at Method Area,
automatically JVM
will create an object for `java.lang.Class` with the metadata of the
loaded class, where the metadata include the following details.
 - a. Name of the class.
 - b. Access modifiers of the class
 - c. Super class details
 - d. Implemented interfaces details
 - e. Variables details
 - f. Constructors details
 - g. Methods details

newInstance() Method

If we load any class bytecode to the memory by using `Class.forName()` method and if we want to create an object for the loaded class then we have to use the following method from `java.lang.Class` .

```
public Object newInstance()throws InstantiationException,  
IllegalAccessException
```

EX:

```
package com.durgasoft.test;  
  
class A{  
  
    static {  
  
        System.out.println("Class Loading.....");  
  
    }  
  
    A(){  
  
        System.out.println("Object Creating.....");  
  
    }  
  
}  
  
class Test{
```

```

public static void main(String[] args) throws
ClassNotFoundException, InstantiationException,
IllegalAccessException{

    Class cls =
Class.forName("com.durgasoft.test.A");

    Object obj = cls.newInstance();

}

}

```

When we execute `cls.newInstance()` method , JVM will perform the following actions.

1. JVM will go to the loaded class, where JVM will search for non private and 0-arg constructor.
2. If 0-arg and non private constructor is available in the loaded class then JVM will execute that constructor and JVM will create an object for the respective class.
3. If the constructor is a parameterized constructor, there is no 0-arg constructor in the loaded class then JVM will raise an exception like `java.lang.InstantiationException`.
4. If the constructor is a private constructor , there is no other constructor then JVM will raise an exception like `java.lang.IllegalAccessException`.
5. If the constructor is private and parameterized then JVM will raise `java.lang.InstantiationException`.

EX:

```
package com.durgasoft.test;
```

```
class A{
```

```
    static {
```

```
        System.out.println("Class Loading....");

    }

A(int i){

    System.out.println("Object Creating.....");

}

}

class Test{

    public static void main(String[] args) throws
ClassNotFoundException, InstantiationException,
IllegalAccessException{

    Class cls =
Class.forName("com.durgasoft.test.A");

    Object obj = cls.newInstance();

}

}

Class Loading....
```

Exception in thread "main"
java.lang.InstantiationException: com.durgasoft.test.A

EX:

```
package com.durgasoft.test;
```

```
class A{

    static {

        System.out.println("Class Loading....");

    }

    private A(){

        System.out.println("Object Creating....");

    }

}

class Test{

    public static void main(String[] args) throws
ClassNotFoundException, InstantiationException,
IllegalAccessException{

        Class cls =
Class.forName("com.durgasoft.test.A");

        Object obj = cls.newInstance();

    }

}

Class Loading....  
  
Exception in thread "main"
java.lang.IllegalAccessException: class
```

com.durgasoft.test.Test cannot access a member of class
com.durgasoft.test.A with modifiers "private"

EX:

```
package com.durgasoft.test;

class A{

    static {

        System.out.println("Class Loading.....");

    }

    private A(int i){

        System.out.println("Object Creating.....");

    }

}

class Test{

    public static void main(String[] args) throws
ClassNotFoundException, InstantiationException,
IllegalAccessException{

        Class cls =
Class.forName("com.durgasoft.test.A");

        Object obj = cls.newInstance();

    }

}
```

}

Class Loading....

Exception in thread "main"
java.lang.InstantiationException: com.durgasoft.test.A

Note: IN Jdbc applications, we have to load and register the driver , that is we have to load driver class bytecode to the memory, for this we have to use Class.forName() method.

EX: Class cls = Class.forName("oracle.jdbc.OracleDriver");

Note: In Servlets execution, Server has to perform Servlet Loading and Servlet Instantiation in Servlet lifecycle, where to perform Servlet Loading Server has to use Class.forName() method , where to perform Servlet Instantiation Server has to execute newInstance() method.

EX:

Class cls = Class.forName("WelcomeServlet"); → Servlet Loading
Object obj = cls.newInstance(); -----> Servlet Instantiation

Note: EJB is a server side component, it will be executed by the application Server by following its lifecycle, where in EJBs lifecycle Application Server has to perform EJB loading and EJBs instantiation, where to perform EJBs loading Application Server has to execute Class.forName() method , where to perform EJBs instantiation Application Server has to execute cls.newInstance() method.

Factory Method:

If any Java method returns a particular class object reference value then that Java method is called “Factory Method”, where the

Factory method is able to return either the same class object or different class object.

Factory method is an idea provided by “Factory Method Design Pattern”.

EX:

```
package com.durgasoft.test;

class A {

    private A() {

        System.out.println("A-Con");

    }

    void m1() {

        System.out.println("m1-A");

    }

    static A getInstance() { //Factory Method

        A a = new A();

        return a;

    }

}

class Test {
```

```
public static void main(String[] args) {  
  
    A a = A.getInstance();  
  
    a.m1();  
  
}  
  
}
```

There are two types of Factory Methods .

1. Static Factory Method
2. Instance Factory Method

Static Factory Method:

If any static method returns a particular class object reference value then that static method is Static Factory Method.

EX:

```
Class cls = Class.forName("A");  
NumberFormat nf = NumberFormat.getInstance();  
Runtime rt = Runtime.getRuntime();
```

Instance Factory Method:

If any instance method returns a particular class object then that instance method is called “Instance Factory Method”.

EX: Majority of the String class methods are Instance Factory Methods.

```
EX: String str = "Durga Software Solutions";  
String str1 = str.toUpperCase();  
String str2 = str.trim();  
String str3 = str.concat(" Hyderabad");  
-----  
-----
```

Note: In general, Factory methods will be used in the preparation of Singleton classes.

Singleton Classes:

If any class allows you to create a single object for it then that class is called “Singleton Class”.

In Java , Singleton class is an idea provided by the “Singleton Design Pattern”.

To create Singleton classes in java applications we have to use the following steps.

1. Declare an user defined class.
2. Declare a private constructor
3. Declare a static reference variable of the same class with null value.
4. Declare a static factory method with the following implementation.
 - a. Check whether any object is created for the current class or not by checking static reference variable value is null or not, if the static reference variable value is null then there is no object for the respective class previously, if the static reference variable value is not null then there is an object for the current class previously.
 - b. If no object is created previously then create a new object for the current class and return that object reference.
 - c. If any object has existed previously then return the existing object without creating a new object.
5. In main class, in main() method , Access static factory method
multiple times and get the same reference value.

EX:

```
package com.durgasoft.test;

class A {

    static A a = null;// a=a111

    private A(){
        //System.out.println("A-Con");
    }

    static A getRef() {
        if(a == null) {
            a = new A();//A@a111
        }
        return a;
    }

}

class Test {

    public static void main(String[] args) {
        A a1 = A.getRef();// a1 = A@a111
    }
}
```

```
A a2 = A.getRef(); // a2 = A@a111
A a3 = A.getRef(); // a3 = A@a111

System.out.println(a1); // A@a111
System.out.println(a2); // A@a111
System.out.println(a3); // A@a111

}
```

OP:

```
com.durgasoft.test.A@626b2d4a
com.durgasoft.test.A@626b2d4a
com.durgasoft.test.A@626b2d4a
```

Or

```
package com.durgasoft.test;

class A {
```

```
static A a = null;// a=a111

static {

    a = new A();

}

private A(){

    //System.out.println("A-Con");

}

static A getRef() {

    return a;

}

}

class Test {

    public static void main(String[] args) {

        A a1 = A.getRef(); // a1 = A@a111

        A a2 = A.getRef(); // a2 = A@a111
```

```
A a3 = A.getRef(); // a3 = A@a111

System.out.println(a1); // A@a111
System.out.println(a2); // A@a111
System.out.println(a3); // A@a111

}
```

In java applications, a static block will be executed exactly one time. If we create an object for a particular class inside a static block then only one object will be created for the respective class, so it makes the class as Singleton class.

Or

```
package com.durgasoft.test;

class A {

    private static A a = new A(); // a=a111

    private A(){
        //System.out.println("A-Con");
    }
}
```

```
}

static A getRef() {

    return a;
}

}

class Test {

    public static void main(String[] args) {

        A a1 = A.getRef();// a1 = A@a111

        A a2 = A.getRef();// a2 = A@a111

        A a3 = A.getRef();// a3 = A@a111

        System.out.println(a1);// A@a111

        System.out.println(a2);// A@a111

        System.out.println(a3);// A@a111
    }
}
```

}

If we create an object along with a static reference variable then only one object will be created for the respective class , because static variables will be recognized and initialized only one time at the time of loading the respective bytecode to the memory.

In general, in MVC based web applications we will use a servlet as controller, where Servlet is a java class.

As per the MVC rules and regulations , per MVC web application only one controller must be provided , that is, the controller servlet class must allow to create only one object , not to allow to create multiple objects, hence the Controller Servlet class must be a “Singleton” class.

‘final’ Keyword:

‘final’ is a Java keyword, it can be used to declare constant expressions.

The main advantage of the ‘final’ keyword is “Security”.

In Java applications we are able to utilize the final keyword in the following three ways.

1. final variables
2. final methods
3. final classes

final variables:

It is a Java variable, it will not allow modifications on its value, if we are trying to perform modifications over its value then the compiler will raise an error.

EX:

```
final int i = 10;  
i = i + 10; ---> Error
```

In general, final variables are not allowing re-assigments.

EX:

In the bank applications, once account number is created then there is no chance of updating account number , here remaining account details may be updated but account number must not be updated, so accNo variable must be declared as final variable.

EX:

In Gmail , Facebook,.... Once id is created then it is not possible to change id value , but remaining details like address details, school and college details.... May be changed, so that id value must be declared as final variable.

In the loops, loop variables must not be declared as final variables, because loop variables must have changes in their values in order to rotate the loop.

EX:

```
for(final int i = 0; i < 10; i++){// ---> Error  
-----  
}
```

‘final’ Methods:

‘final’ method is a Java method, it will not allow change in its functionality, that is, it will not allow overriding operation.

EX:

```
class Teacher{  
    public final void teach(){  
        System.out.println("Teacher teaches on Black Board");  
    }  
}  
class Professor extends Teacher{  
    public void teach(){
```

```
        System.out.println("Profession teaches on Marker
Board");
    }
}
Status: Compilation Error
```

EX:

EX:

```
class Teacher{
    public void teach(){
        System.out.println("Teacher teaches on Black Board");
    }
}
class Professor extends Teacher{
    public final void teach(){
        System.out.println("Profession teaches on Marker
Board");
    }
}
Status: no Compilation Error
```

EX:

```
class Account{
    public final int getMinBal(){
        return 25000;
    }
}
class SalaryAccount extends Account{
    public int getMinBal(){
        return 10000;
    }
}
Status: Compilation Error
```

'final' Classes:

‘final’ class is a Java class, it will not inherit, that is it will not allow subclasses.

In the case of inheritance, a superclass must not be declared as final, but we may declare a subclass as final class.

EX:

```
class A{  
}  
class B extends A{  
}  
Status: Valid
```

EX:

```
final class A{  
}  
class B extends A{  
}  
Status: Invalid
```

EX:

```
class A{  
}  
final class B extends A{  
}  
Status: Valid
```

In java applications, we are able to declare constant variables by using ‘final’ keyword, but Java has given a suggestion to declare constant variables like with “public static final”.

EX1:

In `java.lang.Thread` class, there are three constant variables with “public static final”.

```
public static final int MIN_PRIORITY = 1;  
public static final int NORM_PRIORITY = 5;  
public static final int MAX_PRIORITY = 10;
```

EX2:

In `java.lang.System` class , there are variables with “public static final”.

```
public static final InputStream in;  
public static final PrintStream out;  
public static final PrintStream err;
```

Where the main intention of declaring constant variables with the public is to access that variable throughout the application.

Where the main intention of declaring constant variables with static is to share its value to every object of the respective class and to access that variable without creating an object for the respective class that is accessing that variable by using class name.

Where the main intention of declaring constant variables with final is to fix its value throughout the application.

EX:

```
package com.durgasoft.test;  
  
class UserStatus{  
  
    public static final String AVAILABLE = "User Available";  
  
    public static final String BUSY = "User Busy";  
  
    public static final String IDLE = "User Idle";
```

```
}
```

```
public class Test {
```

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

```
        System.out.println(UserStatus.AVAILABLE);
```

```
        System.out.println(UserStatus.BUSY);
```

```
        System.out.println(UserStatus.IDLE);
```

```
}
```

```
}
```

EX:

```
package com.durgasoft.test;
```

```
class User{
```

```
    public static final int MIN_AGE = 18;
```

```
    public static final int MAX_AGE = 25;
```

```
}
```

```
public class Test {
```

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

```

        System.out.println(User.MIN_AGE);

        System.out.println(User.MAX_AGE);

    }

}

```

To declare and manage the constant variables in java applications if we use the above approach then we are able to get the following problems.

Consider the following program

```

package com.durgasoft.test;

class UserStatus{

    public static final String AVAILABLE = "User Available";

    public static final String BUSY = "User Busy";

    public static final String IDLE = "User Idle";

}

public class Test {

    public static void main(String[] args) {

        System.out.println(UserStatus.AVAILABLE);

        System.out.println(UserStatus.BUSY);

        System.out.println(UserStatus.IDLE);

    }

}

```

1. In the above convention , we must write “public static final” for each and every variable, it is a burden to the developers , so we need an alternative where all variables are by default “public static final”.
2. This approach is able to allow different types to represent one thing, it will reduce typedness in java application, it allows type unsafe operations, here we need an alternative , where all the constant variables must be of the same class type by default, no need to write any specific data type.
3. In this approach, when we access constant variables , Constant variable's values are displayed, where the values of the constant variables may or may not represent the actual intention of the constant variables, here we need an alternative there all the constant variables must be named constants, that is the constants which are not having values but if we access that constant variables their names are displayed as values.

To overcome all the above problems we need an alternative that is “enum”.

“enum” is a container, it can be used to declare constant variables.

Inside the enum,

1. All constant variables are “public static final” by default, no need to declare explicitly.
2. All the constant variables are of the same enum type by default, it improves Typed ness in java applications and it allows typesafe operations.
3. All the constant variables are “Named Constant variables”, they will display their names instead of their values when we access them.

Syntax:

```
[AccessModifier] enum EnumName{  
---List Of Constants---  
}
```

Where Access Modifiers may be public ,protected, <default> and private.

EX:

```
package com.durgasoft.test;

enum UserStatus{

    AVAILABLE,BUSY, IDLE;

}

public class Test {

    public static void main(String[] args) {

        System.out.println(UserStatus.AVAILABLE);

        System.out.println(UserStatus.BUSY);

        System.out.println(UserStatus.IDLE);

    }

}
```

When we compile the above program then the compiler will perform the following translations.

1. Compiler will convert every enum to a final class.
2. Compiler will convert every enum class as a child class to java.lang.Enum , where java.lang.Enum is an intern subclass to java.lang.Object class.
3. Compiler will convert every constant variable of the same enum type.
4. Compiler will convert every constant variable with the “public static final”.

D:\java6>javap UserStatus

```
Compiled from "Test.java"
final class UserStatus extends java.lang.Enum<UserStatus> {
    public static final UserStatus AVAILABLE;
    public static final UserStatus BUSY;
    public static final UserStatus IDLE;
    public static UserStatus[] values();
    public static UserStatus valueOf(java.lang.String);
    static {};
}
```

Q) Is it possible to define inheritance relations between enums?

Ans:

No, it is not possible to extend one enum to another enum , because in java applications every enum is a final class, inheritance is not possible between final classes.

EX:

```
enum enum1{
}
enum enum2 extends enum1{
}
Status: Compilation Error
```

Q) Is it possible to declare normal variables,normal constructors and normal methods along with constant variables inside the enum?

Ans:

Yes, it is possible to write normal variables,methods , constructors along with constant variables inside the enum, because in java applications every enum is a final class, we are able to write normal variables, constructors, methods inside the final classes.

EX:

```
package com.durgasoft.test;

enum Apple{

    A(500),B(300),C(100);

    private int price;

    Apple(int price) {

        this.price = price;

    }

    public int getPrice() {

        return price;

    }

}

public class Test {

    public static void main(String[] args) {

        System.out.println("A-Grade Apple Price : "+Apple.A.getPrice());

        System.out.println("B-Grade Apple Price : "+Apple.B.getPrice());

        System.out.println("C-Grade Apple Price : "+Apple.C.getPrice());

    }

}
```

If we compile the above code then Compiler will perform the following translation.

```
enum Apple{
    A(500),B(300),C(100);
    int price;
    Apple(int price){
        this.price = price;
    }
    public int getPrice(){
        return price;
    }
}

final class Apple extends java.lang.Enum{
    public static final Apple A = new Apple(500);
    public static final Apple B = new Apple(300);
    public static final Apple C = new Apple(100);
    int price;
    Apple(int price){
        this.price = price;
    }
    public int getPrice(){
        return price;
    }
}
```

```

enum Apple{
    A(500),B(300),C(100);
    int price;
    Apple(int price) {
        this.price = price;
    }
    public int getPrice() {
        return price;
    }
}

int price1 = Apple.A.getPrice();
int price2 = Apple.B.getPrice();
int price3 = Apple.C.getPrice();

```

The diagram illustrates the state of the Apple enum after its creation. Three separate oval shapes represent the instances:

- Apple.A**: Contains the value **price = 500**.
- Apple.B**: Contains the value **price = 300**.
- Apple.C**: Contains the value **price = 100**.

Each oval is connected by a line to its corresponding label: **Apple.A**, **Apple.B**, and **Apple.C**.

Importance of main() method in JAVA:

Q) What is the requirement of the main() method in JAVA?

Ans:

1. In java applications we need a method to provide application logic and to execute that provided application logic by JVM automatically, here the required method is the **main()** method.
2. To define starting point and ending point to the application execution we need **main()** method, because **main()** method starting point is the starting point of the application

execution and `main()` method ending point is the ending point of the application execution.

Syntax:

```
public static void main(String[] args){  
-----  
-----  
-----  
}
```

Note: `main()` method is not a predefined method and it is not a user defined method , it is a conventional method with the fixed prototype and with the user defined implementation, it is a minimum method for every java application to write and manage the application logic , it must be provided by the developers automatically when they want to prepare java applications.

Q)What is the requirement to declare the `main()` method with “public”?

Ans:

The main intention of declaring `main()` method as public is to bring `main()` method scope to the JVM in order to access `main()` method by JVM.

Case#1: If we declare the `main()` method as “private” then that `main()` method will have scope up to the respective class, it will not come to the JVM as JVM is available in JAVA software.

Case#2: If we declare `main()` method as default then that `main()` method will have scope up to the respective package , it will not come to the JVM as JVM is available in JAVA software.

Case#3: If we declare the `main()` method as protected then that `main()` method will have scope up to the respective package and

the child classes in other packages, but not to the JVM as JVM is available in JAVA software.

Case#4: If we declare the main() method as public then the main() method will have scope throughout the System including Java software and JVM.

D:\java6\Test.java

```
-----
package com.durgasoft.test;
public class Test{
    public static void main(String[] args){
        -----
    }
}
```

D:\java6>java Test

```
C:\Java\jdk1.8.0\java
Package com.sun.jvm;
public class JVM{
    public static void main(String[] args){
        Test.main(---);
    }
}
```

Q)If we declare the main() method without public in java applications then what will be the result in java applications?

Ans:

```
-----
In Java applications , if we declare main() method without public then compiler will not raise any error, because Compiler will treat main() method as normal java method , we can write normal
```

java methods without public, but JVM will provide the following Exception or message.

JAVA6: Main method not public

JAVA7: Error: Main method not found in class Test, please define the main

method as :

```
public static void main(String[] args)
```

EX:

```
public class Test {  
    static void main(String[] args) {  
        System.out.println("main()-Test");  
    }  
}
```

D:\java6>java6.bat

D:\java6>set path=C:\Java\jdk1.6.0_45\bin;

D:\java6>javac Test.java

D:\java6>java Test

Main method not public.

D:\java6>java7.bat

D:\java6>set path=C:\Java\jdk1.7.0_80\bin;

D:\java6>javac Test.java

D:\java6>java Test

Error: Main method not found in class Test, please define the main method as:

```
public static void main(String[] args)
```

D:\java6>

Q)What is the requirement to declare the main() method as "static"?

Ans:

In Java applications, the main() method must be accessed by JVM in order to start application execution, here JVM was prepared in such a way that to access main() method by using class name, as per the internal implementation of JVM , we must declare main() as static method , because in java applications only static methods are accessed by using class name.

Q)In Java applications, if we declare main() method without static then what will be the result in java applications?

Ans:

In Java applications, if we declare main() method without static then the compiler will not raise any error, because compiler will treat main() method as normal java method, we can declare normal java methods without static, but JVM will provide the following message or Exception.

JAVA6: java.lang.NoSuchMethodError: main

JAVA7: Error: Main method is not static in class Test, please define the

main method as :
 public static void main(String[] args)

EX:

```
public class Test {  
    public void main(String[] args) {  
        System.out.println("main()-Test");  
    }  
}
```

```
}
```

```
D:\java6>java6.bat
```

```
D:\java6>set path=C:\Java\jdk1.6.0_45\bin;
```

```
D:\java6>javac Test.java
```

```
D:\java6>java Test
```

```
Exception in thread "main" java.lang.NoSuchMethodError: main
```

```
D:\java6>java7.bat
```

```
D:\java6>set path=C:\Java\jdk1.7.0_80\bin;
```

```
D:\java6>javac Test.java
```

```
D:\java6>java Test
```

```
Error: Main method is not static in class Test, please define the  
main method as:
```

```
public static void main(String[] args)
```

```
D:\java6>
```

Q) What is the requirement to provide a “void” return type to the `main()` method?

Ans:

In Java applications, we must provide the complete application logic inside the `main()` method, that is, we have to start

application logic at starting point of the main() method and we have to end application logic at ending point of the main() method , because Main Thread will start application execution at starting point of the main() method and It will terminate application execution at ending point of the main().

In the above context, if we want to terminate application logic at the ending point of the main() method , we must not return any value from main() method, if we don't want to return any value from main() method we must use “void” as return type.

Q)If we write main() method without void return type in java applications then what will be the result in java applications?

Ans:

If we declare main() method without void return type then the compiler will not raise any error, but JVM will provide the following exception messages.

JAVA6: java.lang.NoSuchMethodError: main

JAVA7: Error: main method must return a value of type void in class Test,

please define main method as :

public static void main(String[] args)

EX:

```
public class Test {  
    public static int main(String[] args) {  
        System.out.println("main()-Test");  
        return 10;  
    }  
}
```

D:\java6>java6.bat

```
D:\java6>set path=C:\Java\jdk1.6.0_45\bin;
```

```
D:\java6>javac Test.java
```

```
D:\java6>java Test
```

```
Exception in thread "main" java.lang.NoSuchMethodError: main
```

```
D:\java6>java7.bat
```

```
D:\java6>set path=C:\Java\jdk1.7.0_80\bin;
```

```
D:\java6>javac Test.java
```

```
D:\java6>java Test
```

```
Error: Main method must return a value of type void in class  
Test, please
```

```
define the main method as:
```

```
public static void main(String[] args)
```

```
D:\java6>
```

Note: The name of the method “main” is to reflect its importance in java applications.

Q)What is the requirement to provide parameters to the main() method?

Ans:

In Java applications, we are able to provide input data in the following three ways.

1. Static Input
2. Dynamic Input
3. Command Line Input

Static Input:

If we provide input data to the java program at the time of writing the program then that input data is called “Static Input”.

EX:

```
class Calculator{  
    int fval = 10;  
    int sval = 20;  
    void add(){  
        System.out.println(fval+sval);  
    }  
    void sub(){  
        System.out.println(fval-sval);  
    }  
    void mul(){  
        System.out.println(fval*sval);  
    }  
}
```

Dynamic Input:

If we provide input data to the java applications at Runtime then that input data is called “Dynamic Input”.

EX:

```
D:\java6>javac Test.java  
D:\java6>java Test  
First Value : 10  
Second Value : 20  
Addition : 30
```

Command Line Input:

If we provide input data to the java allocation along with “java” command on command prompt then that input data is called “Command Line Input”.

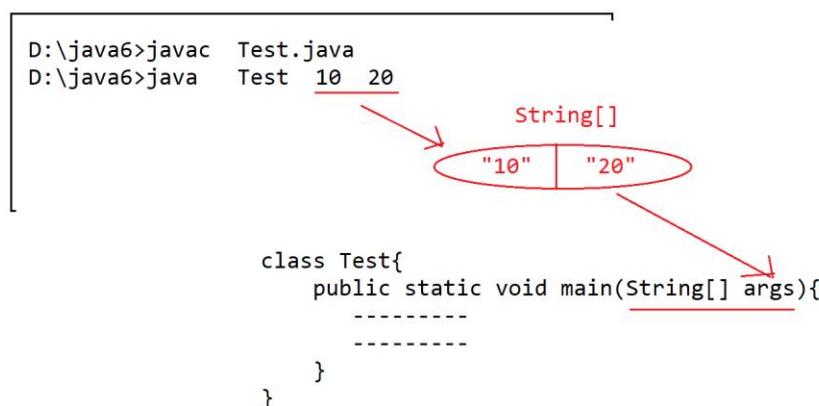
EX:

```
D:\java6>java Test 10 20
```

If we provide command line input along with java command on command prompt then JVM will perform the following actions.

1. JVM will read all the command line input which we provided along with java command on command prompt.
2. JVM will store them in the form of String[] .
3. JVM will pass the generated String[] as parameter to main() method at the time of accessing main() method.

The main intention of the main() method parameters is to store all the command line input in the form of String[] and to provide all these command line input into the java applications.



EX:

```
public class Test {  
    public static void main(String[] args) {  
        System.out.println("No of Command Line Arguments : "  
"+args.length);  
        System.out.print("List Of Command Line Arguments : ");  
        for(int i = 0; i < args.length; i++){  
            System.out.print(args[i]+ " ");  
        }  
    }  
}
```

```
    }  
}  
}
```

D:\java6>javac Test.java

```
D:\java6>java Test 10 22.22f true 'A' "abc"  
No of Command Line Arguments : 5  
List Of Command Line Arguments : 10 22.22f true 'A' abc  
D:\java6>java Test 10 20  
No of Command Line Arguments : 2  
List Of Command Line Arguments : 10 20  
D:\java6>
```

Q)What is the requirement to provide String data type as parameter to the main() method?

Ans:

In general, from application to application we may provide different types of command line input as per the requirement, even if we provide different types of command line input to the java application , main() method parameter must store all the types of command line input, in JAVA/J2EE applications only String data type is having capability to store any type of data, hence main() method must have String data type as parameter in order to store any type of command line input.

Q)What is the requirement to provide an array as a parameter to the main() method?

Ans:

From application to application we may provide a variable number of command line input as per the requirement, even though if we provide a variable number of command line arguments, our main() method parameter must store all the variable number of command line input, in JAVA / J2EE applications arrays are having capability to store multiple values, Hence main() method must have array as parameter inorder to store variable number of command line arguments.

Q) If we declare main() method without String[] parameter then what will be the result in java application?

Ans:

If we declare main() method without String[] parameter then Compiler will not raise any error, because Compiler will treat main() method as normal java method, we can declare normal java methods without the String[] parameter, but JVM will provide the following Exception messages.

JAVA6: java.lang.NoSuchMethodError. main

JAVA7: Error: main method not found in class Test,please defined the main

method as :

```
public static void main(String[] args)
```

EX:

```
public class Test {  
    public static void main(int[] args) {  
        System.out.println("main()-Test");  
    }  
}
```

```
}
```

```
D:\java6>java6.bat
```

```
D:\java6>set path=C:\Java\jdk1.6.0_45\bin;
```

```
D:\java6>javac Test.java
```

```
D:\java6>java Test
```

```
Exception in thread "main" java.lang.NoSuchMethodError: main
```

```
D:\java6>java7.bat
```

```
D:\java6>set path=C:\Java\jdk1.7.0_80\bin;
```

```
D:\java6>javac Test.java
```

```
D:\java6>java Test
```

```
Error: Main method not found in class Test, please define the  
main method as:
```

```
public static void main(String[] args)
```

```
D:\java6>
```

EX:

```
public class Test {  
    public static void main(String[] args) {  
        for(int i = 0; i < args.length; i++){  
            System.out.println(args[i]);  
        }  
    }  
}
```

```
D:\java6>javac Test.java
```

```
D:\java6>java Test Durga Software Solutions
Durga
Software
Solutions
```

```
D:\java6>java Test "Durga Software Solutions"
Durga Software Solutions
```

D:\java6>

Q) Is it possible to provide more than one main() in a single java application?

Ans:

Yes, it is possible to provide more than one main() method in a single java application, but in multiple classes. In this case, which class name we provided along with the “java” command that class provided main() method will be executed.

EX:

```
class A {
    public static void main(String[] args) {
        System.out.println("main()-A");
    }
}
class B {
    public static void main(String[] args) {
        System.out.println("main()-B");
    }
}
class C {
    public static void main(String[] args) {
        System.out.println("main()-C");
    }
}
```

```
}
```

```
D:\java6>javac Test.java
```

```
D:\java6>java A  
main()-A
```

```
D:\java6>java B  
main()-B
```

```
D:\java6>java C  
main()-C
```

```
D:\java6>
```

In the above context, it is possible to access one class main() method from another class main() method , just by thinking of the main() as a normal static method with the parameter String[] .

EX:

```
class A {  
    public static void main(String[] args) {  
        System.out.println("main()-A");  
        String[] str = {"AAA", "BBB", "CCC"};  
        B.main(str);  
    }  
}  
class B {  
    public static void main(String[] args) {  
        System.out.println("main()-B");  
        C.main(args);  
    }  
}  
class C {  
    public static void main(String[] args) {  
        System.out.println("main()-C");  
    }  
}
```

```
    }
}
```

D:\java6>javac Test.java

D:\java6>java A
main()-A
main()-B
main()-C

D:\java6>

Q)Is it possible to overload the main() method in Java applications?

Ans:

Yes, it is possible to overload main() method in java applications, but it is not possible to override main() method in java applications, because in java applications static method overloading is possible , but static method overriding is not possible.

In the above context, if we provide more than one main() method with the same name and with a different parameter list then JVM will access only the main() method which has a String[] parameter.

EX:

```
class Test {
    public static void main(String[] args) {
        System.out.println("main(String[] args)-Test");
    }
    public static void main(int[] args){
        System.out.println("main(int[] args)-Test");
    }
}
```

```
    }
    public static void main(float[] args){
        System.out.println("main(float[] args)-Test");
    }
}
```

D:\java6>javac Test.java

D:\java6>java Test
main(String[] args)-Test

D:\java6>

Q)Find the valid syntaxes to the main() method from the following list?

1. public static void main(String[] args) ----> Valid
2. public static void main(String[] abc) -----> Valid
3. public static void main(int[] args) -----> Invalid
4. public static void main(String args) -----> Invalid
5. public static void main(String args1, String args2)--->
 Invalid
6. public static void main(String[][] args) --> Invalid
7. public static void main(String args[]) -----> Valid
8. public static void main(String []args) -----> Valid
9. public static void main(string[] args) -----> Invalid
10. public static void main(String ... args) --> Valid
11. public static void Main(String[] args) -----> Invalid
12. public static int main(String[] args) -----> Invalid
13. public final void main(String[] args) -----> Invalid
14. public static final void main(String[] args) --> Valid
15. protected static void main(String[] args) ----->
 INvalid
16. static public void main(String[] args) -----> Valid

IntelliJ Idea

1. Download And Install IntelliJ Idea
2. Open IntelliJ Idea and Create Java project
3. Write application logic in the main() method.
4. Execute Java application

Relationships in Java:

If we want to provide effective implementation in any project then we have to provide the following factors.

1. Less Memory
 2. Less Execution Time
 3. Code Reusability
-
-

If we want to provide all the factors in the applications then we have to define Relationships between classes and Design patterns over the code.

There are three types Relationships between classes.

1. HAS-A Relationship
2. IS-A Relationship
3. USES-A Relationship

Q) What is the difference between a HAS-A Relationship and IS-A Relationship?

Ans:

HAS-A Relationship is able to define Associations between entities, here the associations between entities will improve communication between classes and data navigation between classes.

IS-A Relationship is able to define inheritance between classes, here inheritance relationship is able to improve Code Reusability.

Associations In Java:

Association is a join relation between two entities, when we can access one entity member in another entity.

The main advantage of Associations is to improve Data Navigation from one entity to another entity and Communication between entities.

To implement associations in Java applications we have to declare either single or array of other entity references in the present entity class.

```
class Account{  
}  
class Technology{  
}  
class Employee{  
    int eno;  
    String ename;  
    float esal;  
    Account account;//One-To-One association  
    Technology[] techs;// One-To-Many Associations  
}
```

There are four types of Associations.

1. One-To-One Association
2. One-To-Many Association

3. Many-To-One Association
4. Many-To-Many Association

All the above associations are achieved by using Dependency Injection.

The process of injecting a dependency object into another object is called Dependency Injection.

There are two ways to achieve dependency injection.

1. Constructors Dependency Injection
2. Setter Method Dependency Injection

Constructors Dependency Injection:

The process of injecting dependency objects into another object through a constructor is called “Constructor Dependency Injection”.

EX:

```
class Account{  
}  
class Employee{  
    Account account;  
    Employee(Account account){  
        this.account = account;  
    }  
}
```

Setter Method Dependency Injection:

The process of injecting an object into another object through a setter method is called Setter method dependency injection.

EX:

```
class Account{  
}
```

```
class Employee{  
    private Account account;  
    public void setAccount(Account account){  
        this.account = account;  
    }  
}
```

One-To-One Association:

It is a relation between classes, where one instance of an entity should be matched with exactly one instance of another entity.

EX: Every Employee has an individual Salary Account.

EX on Constructor Dependency Injection:

Account.java

```
package com.durgasoft.entities;
```

```
public class Account {  
    private String accNo;  
    private String accHolderName;  
    private String accType;  
    private int balance;  
  
    public String getAccNo() {  
        return accNo;  
    }  
  
    public void setAccNo(String accNo) {  
        this.accNo = accNo;  
    }  
  
    public String getAccHolderName() {  
        return accHolderName;  
    }  
  
    public void setAccHolderName(String accHolderName) {
```

```
        this.accHolderName = accHolderName;
    }

    public String getAccType() {
        return accType;
    }

    public void setAccType(String accType) {
        this.accType = accType;
    }

    public int getBalance() {
        return balance;
    }

    public void setBalance(int balance) {
        this.balance = balance;
    }
}
```

Employee.java

```
package com.durgasoft.entities;

public class Employee {
    private int eno;
    private String ename;
    private float esal;
    private String eaddr;
    Account account;

    public Employee(int eno, String ename, float esal, String eaddr,
Account account) {
        this.eno = eno;
        this.ename = ename;
        this.esal = esal;
        this.eaddr = eaddr;
        this.account = account;
    }
}
```

```

public void getEmployeeDetails(){
    System.out.println("Employee Details");
    System.out.println("-----");
    System.out.println("Employee Number      : "+eno);
    System.out.println("Employee Name        : "+ename);
    System.out.println("Employee Salary      : "+esal);
    System.out.println("Employee Address     : "+eaddr);

    System.out.println("Account Details");
    System.out.println("-----");
    System.out.println("Account Number       : "+account.getAccNo());
    System.out.println("Account Holder Name : "+account.getAccHolderName());
    System.out.println("Account Type         : "+account.getAccType());
    System.out.println("Account Balance      : "+account.getBalance());
}

}

```

Main.java

```

import com.durgasoft.entities.Account;
import com.durgasoft.entities.Employee;

public class Main {
    public static void main(String[] args) {
        Account account = new Account();
        account.setAccNo("abc123");
        account.setAccHolderName("Durga");
        account.setAccType("Savings");
        account.setBalance(50000);

        Employee employee = new Employee(111, "Durga", 25000, "Hyd",
account);
        employee.getEmployeeDetails();
    }
}

```

```
    }
}
```

Employee Details

```
Employee Number      : 111
Employee Name       : Durga
Employee Salary     : 25000.0
Employee Address    : Hyd
```

Account Details

```
Account Number      : abc123
Account Holder Name : Durga
Account Type        : Savings
Account Balance     : 50000
```

EX on Setter Method Dependency Injection:

Account.java

```
package com.durgasoft.entities;

public class Account {
    private String accNo;
    private String accHolderName;
    private String accType;
    private int balance;

    public String getAccNo() {
        return accNo;
    }

    public void setAccNo(String accNo) {
        this.accNo = accNo;
    }

    public String getAccHolderName() {
        return accHolderName;
    }
}
```

```
public void setAccHolderName(String accHolderName) {
    this.accHolderName = accHolderName;
}

public String getAccType() {
    return accType;
}

public void setAccType(String accType) {
    this.accType = accType;
}

public int getBalance() {
    return balance;
}

public void setBalance(int balance) {
    this.balance = balance;
}
}
```

Employee.java

```
package com.durgasoft.entities;

public class Employee {
    private int eno;
    private String ename;
    private float esal;
    private String eaddr;
    private Account account;

    public int getEno() {
        return eno;
    }

    public void setEno(int eno) {
        this.eno = eno;
    }
}
```

```
}

public String getEname() {
    return ename;
}

public void setEname(String ename) {
    this.ename = ename;
}

public float getEsal() {
    return esal;
}

public void setEsal(float esal) {
    this.esal = esal;
}

public String getEaddr() {
    return eaddr;
}

public void setEaddr(String eaddr) {
    this.eaddr = eaddr;
}

public Account getAccount() {
    return account;
}

public void setAccount(Account account) {
    this.account = account;
}

public void getEmployeeDetails(){
    System.out.println("Employee Details");
    System.out.println("-----");
    System.out.println("Employee Number : "+eno);
```

```

        System.out.println("Employee Name      : "+ename);
        System.out.println("Employee Salary    : "+esal);
        System.out.println("Employee Address   : "+eaddr);
        System.out.println();

        System.out.println("Account Details");
        System.out.println("-----");
        System.out.println("Account Number     : ");
"+account.getAccNo());
        System.out.println("Account Holder Name : ");
"+account.getAccHolderName());
        System.out.println("Account Type       : ");
"+account.getAccType());
        System.out.println("Account Balance    : ");
"+account.getBalance());
    }
}

```

Main.java

```

import com.durgasoft.entities.Account;
import com.durgasoft.entities.Employee;

public class Main {
    public static void main(String[] args) {
        Account account = new Account();
        account.setAccNo("abc123");
        account.setAccHolderName("Durga");
        account.setAccType("Savings");
        account.setBalance(50000);

        Employee employee = new Employee();
        employee.setEno(111);
        employee.setEname("Durga");
        employee.setEsal(50000);
        employee.setEaddr("Hyd");
    }
}

```

```

        employee.setAccount(account);

        employee.getEmployeeDetails();
    }
}

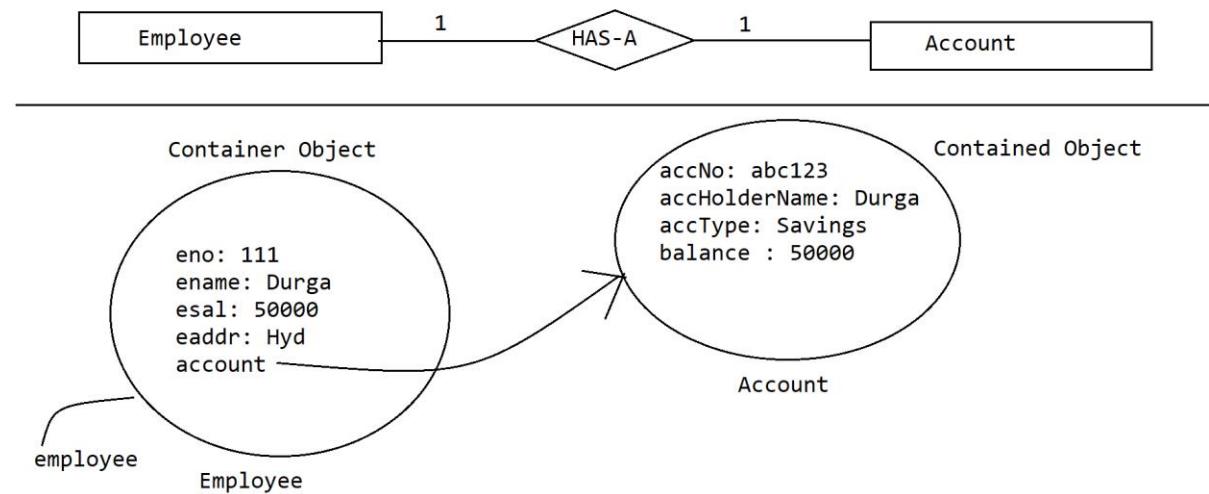
```

Employee Details

Employee Number : 111
 Employee Name : Durga
 Employee Salary : 50000.0
 Employee Address : Hyd

Account Details

Account Number : abc123
 Account Holder Name : Durga
 Account Type : Savings
 Account Balance : 50000



One-To-Many Association:

It is a relation between entity classes, where one instance of an entity should be mapped with multiple instances of another entity.

EX: Single Delivery Agent should have multiple Items.

EX on Constructor Dependency Injection:

Item.java

```
package com.durgasoft.entities;
```

```
public class Item {  
    private String itemId;  
    private String itemName;  
    private int price;  
    private String paymentMode;  
    private String deliveryDate;  
  
    public String getItemId() {  
        return itemId;  
    }  
  
    public void setId(String itemId) {  
        this.itemId = itemId;  
    }  
  
    public String getName() {  
        return itemName;  
    }  
  
    public void setName(String itemName) {  
        this.itemName = itemName;  
    }  
  
    public int getPrice() {  
        return price;  
    }  
  
    public void setPrice(int price) {
```

```
        this.price = price;
    }

    public String getPaymentMode() {
        return paymentMode;
    }

    public void setPaymentMode(String paymentMode) {
        this.paymentMode = paymentMode;
    }

    public String getDeliveryDate() {
        return deliveryDate;
    }

    public void setDeliveryDate(String deliveryDate) {
        this.deliveryDate = deliveryDate;
    }
}
```

DeliveryAgent.java

```
package com.durgasoft.entities;

public class DeliveryAgent {
    private String agentId;
    private String agentName;
    private String agentMobileNo;
    private String customerName;
    private Item[] items;

    public DeliveryAgent(String agentId, String agentName, String
agentMobileNo, String customerName, Item[] items) {
        this.agentId = agentId;
        this.agentName = agentName;
        this.agentMobileNo = agentMobileNo;
        this.customerName = customerName;
        this.items = items;
    }
}
```

```

public void getDeliveryDetails(){
    System.out.println("Item Delivery Details");
    System.out.println("-----");
    System.out.println("Delivery Agent Id      : "+agentId);
    System.out.println("Delivery Agent Name    : "+agentName);
    System.out.println("Delivery Agent Mobile  : "+agentMobileNo);
    System.out.println("Customer Name          : "+customerName);
    System.out.println();
    System.out.println("ITEM ID\tITEM
NAME\tPRICE\tPAYMODE\tDELIVERDATE");
    System.out.println("-----");
    for(int index = 0; index < items.length; index++){
        Item item = items[index];
        System.out.print(item.getItemId()+"\t");
        System.out.print(item.getItemName()+"\t\t");
        System.out.print(item.getPrice()+"\t");
        System.out.print(item.getPaymentMode()+"\t\t");
        System.out.print(item.getDeliveryDate()+"\n");
    }
}
}

```

Main.java

```

import com.durgasoft.entities.DeliveryAgent;
import com.durgasoft.entities.Item;

public class Main {
    public static void main(String[] args) {
        Item item1 = new Item();
        item1.setItemId("I-111");
        item1.setItemName("Mobile");
        item1.setPrice(25000);
        item1.setPaymentMode("COD");
        item1.setDeliveryDate("11-03-2023");
    }
}

```

```

        Item item2 = new Item();
        item2.setItemId("I-222");
        item2.setItemName("Shirt");
        item2.setPrice(2500);
        item2.setPaymentMode("UPI");
        item2.setDeliveryDate("11-03-2023");

        Item item3 = new Item();
        item3.setItemId("I-333");
        item3.setItemName("Laptop");
        item3.setPrice(50000);
        item3.setPaymentMode("UPI");
        item3.setDeliveryDate("11-03-2023");

        Item[] items = {item1, item2, item3};

        DeliveryAgent deliveryAgent = new DeliveryAgent("D111",
        "Anil", "9988776655", "Durga", items);
        deliveryAgent.getDeliveryDetails();

    }

}

```

EX on Setter Method Dependency Injection:

```

Item.java
package com.durgasoft.entities;

public class Item {
    private String itemId;
    private String itemName;
    private int price;
    private String paymentMode;
    private String deliveryDate;

    public String getItemId() {
        return itemId;
    }
}
```

```
}

public void setItemId(String itemId) {
    this.itemId = itemId;
}

public String getItemName() {
    return itemName;
}

public void setItemName(String itemName) {
    this.itemName = itemName;
}

public int getPrice() {
    return price;
}

public void setPrice(int price) {
    this.price = price;
}

public String getPaymentMode() {
    return paymentMode;
}

public void setPaymentMode(String paymentMode) {
    this.paymentMode = paymentMode;
}

public String getDeliveryDate() {
    return deliveryDate;
}

public void setDeliveryDate(String deliveryDate) {
    this.deliveryDate = deliveryDate;
}

}
```

DeliveryAgent.java

```
package com.durgasoft.entities;

public class DeliveryAgent {
    private String agentId;
    private String agentName;
    private String agentMobileNo;
    private String customerName;
    private Item[] items;

    public String getAgentId() {
        return agentId;
    }

    public void setAgentId(String agentId) {
        this.agentId = agentId;
    }

    public String getAgentName() {
        return agentName;
    }

    public void setAgentName(String agentName) {
        this.agentName = agentName;
    }

    public String getAgentMobileNo() {
        return agentMobileNo;
    }

    public void setAgentMobileNo(String agentMobileNo) {
        this.agentMobileNo = agentMobileNo;
    }
}
```

```
public String getCustomerName() {
    return customerName;
}

public void setCustomerName(String customerName) {
    this.customerName = customerName;
}

public Item[] getItems() {
    return items;
}

public void setItems(Item[] items) {
    this.items = items;
}

public void getDeliveryDetails(){
    System.out.println("Item Delivery Details");
    System.out.println("-----");
    System.out.println("Delivery Agent Id      : "+agentId);
    System.out.println("Delivery Agent Name     : "+agentName);
    System.out.println("Delivery Agent Mobile   : "+agentMobileNo);
    System.out.println("Customer Name           : "+customerName);
    System.out.println();
    System.out.println("ITEM ID\tITEM NAME\tPRICE\tPAYMODE\tDELIVERDATE");
    System.out.println("-----");
    for(int index = 0; index < items.length; index++){
        Item item = items[index];
        System.out.print(item.getItemId()+"\t");
        System.out.print(item.getItemName()+"\t\t");
        System.out.print(item.getPrice()+"\t");
        System.out.print(item.getPaymentMode()+"\t\t");
        System.out.print(item.getDeliveryDate()+"\n");
    }
}
```

```
        }
    }

Main.java
import com.durgasoft.entities.DeliveryAgent;
import com.durgasoft.entities.Item;

public class Main {
    public static void main(String[] args) {
        Item item1 = new Item();
        item1.setItemId("I-111");
        item1.setItemName("Mobile");
        item1.setPrice(2500);
        item1.setPaymentMode("COD");
        item1.setDeliveryDate("11-03-2023");

        Item item2 = new Item();
        item2.setItemId("I-222");
        item2.setItemName("Shirt");
        item2.setPrice(2500);
        item2.setPaymentMode("UPI");
        item2.setDeliveryDate("11-03-2023");

        Item item3 = new Item();
        item3.setItemId("I-333");
        item3.setItemName("Laptop");
        item3.setPrice(5000);
        item3.setPaymentMode("UPI");
        item3.setDeliveryDate("11-03-2023");

        Item[] items = {item1, item2, item3};

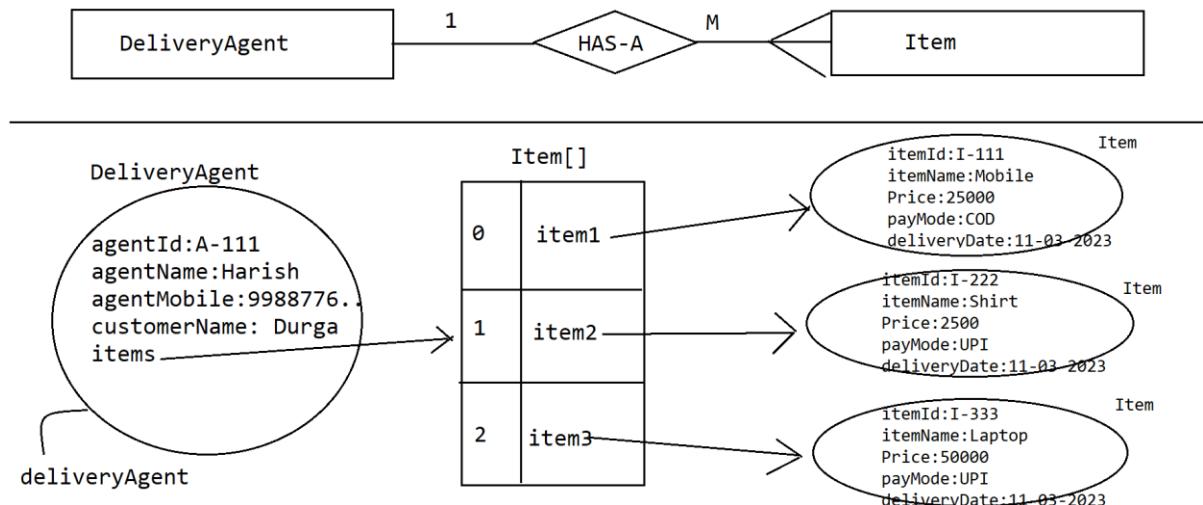
        DeliveryAgent deliveryAgent = new DeliveryAgent();
        deliveryAgent.setAgentId("A-111");
        deliveryAgent.setAgentName("Harish");
        deliveryAgent.setAgentMobileNo("9988776655");
        deliveryAgent.setCustomerName("Durga");
    }
}
```

```

        deliveryAgent.setItems(items); // OneToMany Association
        through Setter method dependency injection

        deliveryAgent.getDeliveryDetails();
    }
}

```



Many-To-One Association:

It is a relation between entity classes, where multiple instances of an entity should be mapped with exactly one instance of an entity.

EX:

Multiple Students joined with Single Branch

EX on COnstructor Dependency Injection:

Branch.java

```
package com.durgasoft.entities;
```

```
public class Branch {
```

```
private String branchId;
private String branchName;

public String getBranchId() {
    return branchId;
}

public void setBranchId(String branchId) {
    this.branchId = branchId;
}

public String getBranchName() {
    return branchName;
}

public void setBranchName(String branchName) {
    this.branchName = branchName;
}
}
```

Student.java

```
package com.durgasoft.entities;

public class Student {
    private String sid;
    private String sname;
    private String saddr;
    private Branch branch;

    public Student(String sid, String sname, String saddr, Branch
branch) {
        this.sid = sid;
        this.sname = sname;
        this.saddr = saddr;
        this.branch = branch;
    }

    public void getStudentDetails(){
```

```
        System.out.println("Student Details");
        System.out.println("-----");
        System.out.println("Student Id      : "+sid);
        System.out.println("Student Name    : "+sname);
        System.out.println("Student Address : "+saddr);
        System.out.println("Branch Id      :
"+branch.getBranchId());
        System.out.println("Branch Name     :
"+branch.getBranchName());
    }
}
```

Main.java

```
import com.durgasoft.entities.Branch;
import com.durgasoft.entities.Student;

public class Main {
    public static void main(String[] args) {
        Branch branch = new Branch();
        branch.setBranchId("B-111");
        branch.setBranchName("Computers");

        Student student1 = new Student("S-111", "AAA", "Hyd",
branch);
        Student student2 = new Student("S-222", "BBB", "Hyd",
branch);
        Student student3 = new Student("S-333", "CCC", "Hyd",
branch);

        student1.getStudentDetails();
        System.out.println();

        student2.getStudentDetails();
        System.out.println();

        student3.getStudentDetails();
    }
}
```

```
}
```

EX on Setter method Dependency Injection:

Branch.java

```
package com.durgasoft.entities;
```

```
public class Branch {  
    private String branchId;  
    private String branchName;  
  
    public String getBranchId() {  
        return branchId;  
    }  
  
    public void setBranchId(String branchId) {  
        this.branchId = branchId;  
    }  
  
    public String getBranchName() {  
        return branchName;  
    }  
  
    public void setBranchName(String branchName) {  
        this.branchName = branchName;  
    }  
}
```

Student.java

```
package com.durgasoft.entities;
```

```
public class Student {  
    private String sid;  
    private String sname;  
    private String saddr;  
    private Branch branch;  
  
    public String getSid() {  
        return sid;  
    }
```

```
}

public void setSid(String sid) {
    this.sid = sid;
}

public String getSname() {
    return sname;
}

public void setSname(String sname) {
    this.sname = sname;
}

public String getSaddr() {
    return saddr;
}

public void setSaddr(String saddr) {
    this.saddr = saddr;
}

public Branch getBranch() {
    return branch;
}

public void setBranch(Branch branch) {
    this.branch = branch;
}

public void getStudentDetails(){
    System.out.println("Student Details");
    System.out.println("-----");
    System.out.println("Student Id      : "+sid);
    System.out.println("Student Name    : "+sname);
    System.out.println("Student Address : "+saddr);
    System.out.println("Branch Id       : "+branch.getBranchId());
}
```

```
        System.out.println("Branch Name      :  
"+branch.getBranchName());  
    }  
}  
  
Main.java  
import com.durgasoft.entities.Branch;  
import com.durgasoft.entities.Student;  
  
public class Main {  
    public static void main(String[] args) {  
        Branch branch = new Branch();  
        branch.setBranchId("B-111");  
        branch.setBranchName("Computers");  
  
        Student student1 = new Student();  
        student1.setSid("S-111");  
        student1.setSname("AAA");  
        student1.setSaddr("Hyd");  
        student1.setBranch(branch);  
  
        Student student2 = new Student();  
        student2.setSid("S-222");  
        student2.setSname("BBB");  
        student2.setSaddr("Hyd");  
        student2.setBranch(branch);  
  
        Student student3 = new Student();  
        student3.setSid("S-333");  
        student3.setSname("CCC");  
        student3.setSaddr("Hyd");  
        student3.setBranch(branch);  
  
        student1.getStudentDetails();  
        System.out.println();  
  
        student2.getStudentDetails();  
        System.out.println();
```

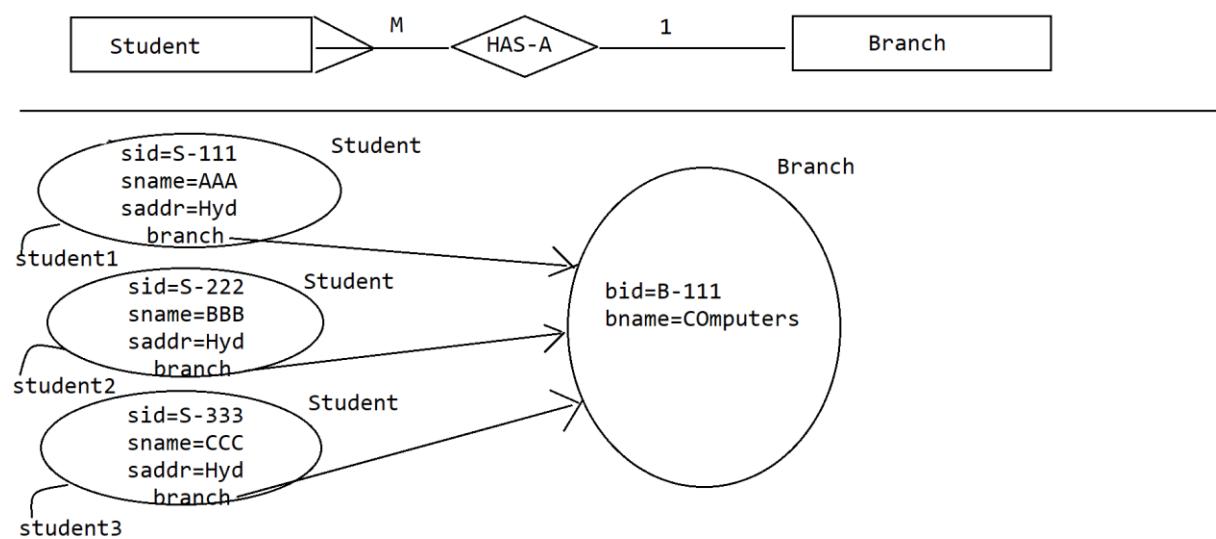
```

        student3.getStudentDetails();

    }

}

```



Many-To-Many Association:

It is a relation between entity classes, where multiple instances of an entity should be mapped with Multiple Instances of another entity.

EX: Multiple Students have joined with multiple Courses

EX On Constructor Dependency injection:

Course.java

```

package com.durgasoft.entities;

public class Course {
    private String cid;
    private String cname;
}

```

```
private int cfee;

public String getCid() {
    return cid;
}

public void setCid(String cid) {
    this.cid = cid;
}

public String getCname() {
    return cname;
}

public void setCname(String cname) {
    this.cname = cname;
}

public int getCfee() {
    return cf_fee;
}

public void setCfee(int cf_fee) {
    this.cf_fee = cf_fee;
}
}
```

Student.java

```
package com.durgasoft.entities;

public class Student {
    private String sid;
    private String sname;
    private String saddr;
    private Course[] courses;

    public Student(String sid, String sname, String saddr, Course[] courses) {
```

```

        this.sid = sid;
        this.sname = sname;
        this.saddr = saddr;
        this.courses = courses;
    }

    public void getStudentDetails(){
        System.out.println("Student Details");
        System.out.println("-----");
        System.out.println("Student Id      : "+sid);
        System.out.println("Student Name     : "+sname);
        System.out.println("Student Address   : "+saddr);
        System.out.println("CID\t\tCNAME\tCFEE");
        System.out.println("-----");
        for(int index = 0; index < courses.length; index++){
            Course course = courses[index];
            System.out.print(course.getCid()+"\t");
            System.out.print(course.getCname()+"\t");
            System.out.print(course.getFee()+"\n");
        }
    }
}

```

Main.java

```

import com.durgasoft.entities.Course;
import com.durgasoft.entities.Student;

public class Main {
    public static void main(String[] args) {
        Course course1 = new Course();
        course1.setCid("C-111");
        course1.setCname("JAVA");
        course1.setFee(30000);

        Course course2 = new Course();
        course2.setCid("C-222");
        course2.setCname("PYTHON");
        course2.setFee(20000);
    }
}

```

```

        Course course3= new Course();
        course3.setCid("C-333");
        course3.setCname(".NET");
        course3.setCfee(10000);

        Course[] courses1 = {course1, course2, course3};
        Course[] courses2 = {course1, course2};
        Course[] courses3 = {course1, course3};

        Student student1 = new Student("S-111", "AAA", "Hyd",
courses1);
        Student student2 = new Student("S-222", "BBB", "Hyd",
courses2);
        Student student3 = new Student("S-333", "CCC", "Hyd",
courses3);

        student1.getStudentDetails();
        System.out.println();

        student2.getStudentDetails();
        System.out.println();

        student3.getStudentDetails();

    }

}

```

EX On Setter Method Dependency Injection:

Course.java

```

package com.durgasoft.entities;

public class Course {
    private String cid;
    private String cname;
    private int cfee;

    public String getCid() {

```

```
        return cid;
    }

    public void setCid(String cid) {
        this.cid = cid;
    }

    public String getCname() {
        return cname;
    }

    public void setCname(String cname) {
        this.cname = cname;
    }

    public int getCfee() {
        return cfee;
    }

    public void setCfee(int cfee) {
        this.cfee = cfee;
    }
}
```

Student.java

```
package com.durgasoft.entities;

public class Student {
    private String sid;
    private String sname;
    private String saddr;
    private Course[] courses;

    public String getSid() {
        return sid;
    }

    public void setSid(String sid) {
```

```
        this.sid = sid;
    }

    public String getName() {
        return sname;
    }

    public void setName(String sname) {
        this.sname = sname;
    }

    public String getSaddr() {
        return saddr;
    }

    public void setSaddr(String saddr) {
        this.saddr = saddr;
    }

    public Course[] getCourses() {
        return courses;
    }

    public void setCourses(Course[] courses) {
        this.courses = courses;
    }

    public void getStudentDetails(){
        System.out.println("Student Details");
        System.out.println("-----");
        System.out.println("Student Id      : "+sid);
        System.out.println("Student Name    : "+sname);
        System.out.println("Student Address : "+saddr);
        System.out.println("CID\t\tCNAME\tCFEE");
        System.out.println("-----");
        for(int index = 0; index < courses.length; index++){
            Course course = courses[index];
            System.out.print(course.getCid()+"\t");
        }
    }
}
```

```
        System.out.print(course.getCname()+"\t");
        System.out.print(course.getFee()+"\n");
    }
}
}

Main.java
import com.durgasoft.entities.Course;
import com.durgasoft.entities.Student;

public class Main {
    public static void main(String[] args) {
        Course course1 = new Course();
        course1.setCid("C-111");
        course1.setCname("JAVA");
        course1.setFee(30000);

        Course course2 = new Course();
        course2.setCid("C-222");
        course2.setCname("PYTHON");
        course2.setFee(20000);

        Course course3= new Course();
        course3.setCid("C-333");
        course3.setCname(".NET");
        course3.setFee(10000);

        Course[] courses1 = {course1, course2, course3};
        Course[] courses2 = {course1, course2};
        Course[] courses3 = {course1, course3};

        Student student1 = new Student();
        student1.setSid("S-111");
        student1.setSname("AAA");
        student1.setSaddr("Hyd");
        student1.setCourses(courses1);

        Student student2 = new Student();
```

```

student2.setSid("S-222");
student2.setSname("BBB");
student2.setSaddr("Hyd");
student2.setCourses(courses2);

Student student3 = new Student();
student3.setSid("S-333");
student3.setSname("CCC");
student3.setSaddr("Hyd");
student3.setCourses(courses3);

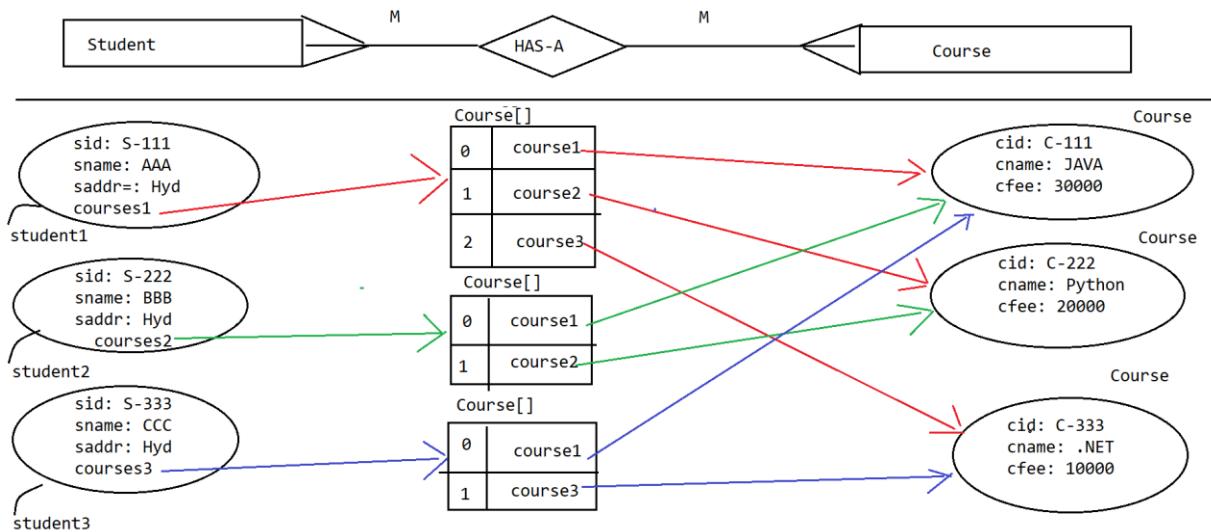
student1.getStudentDetails();
System.out.println();

student2.getStudentDetails();
System.out.println();

student3.getStudentDetails();

}
}

```



Q) Write a Java program to represent the entities like an Employee and the associated Account and Skillset[Technologies]?

Account.java

```
package com.durgasoft.entities;

public class Account {
    private String accNo;
    private String accHolderName;
    private String accType;
    private int balance;

    public String getAccNo() {
        return accNo;
    }

    public void setAccNo(String accNo) {
        this.accNo = accNo;
    }

    public String getAccHolderName() {
        return accHolderName;
    }

    public void setAccHolderName(String accHolderName) {
        this.accHolderName = accHolderName;
    }

    public String getAccType() {
        return accType;
    }

    public void setAccType(String accType) {
        this.accType = accType;
    }

    public int getBalance() {
```

```
        return balance;
    }

    public void setBalance(int balance) {
        this.balance = balance;
    }
}
```

Skill.java

```
package com.durgasoft.entities;

public class Skill {
    private String skillId;
    private String skillName;

    public String getSkillId() {
        return skillId;
    }

    public void setSkillId(String skillId) {
        this.skillId = skillId;
    }

    public String getSkillName() {
        return skillName;
    }

    public void setSkillName(String skillName) {
        this.skillName = skillName;
    }
}
```

Employee.java

```
package com.durgasoft.entities;

import java.util.Arrays;

public class Employee {
```

```

private int eno;
private String ename;
private float esal;
private String eaddr;

private Account account;
private Skill[] skillSet;

public Employee(int eno, String ename, float esal, String eaddr,
Account account, Skill[] skillSet) {
    this.eno = eno;
    this.ename = ename;
    this.esal = esal;
    this.eaddr = eaddr;
    this.account = account;
    this.skillSet = skillSet;
}

public void getEmployeeDetails() {
    System.out.println("Employee Details");
    System.out.println("-----");
    System.out.println("Employee Number      : "+eno);
    System.out.println("Employee Name        : "+ename);
    System.out.println("Employee Salary       : "+esal);
    System.out.println("Employee Address     : "+eaddr);

    System.out.println("Account Details");
    System.out.println("-----");
    System.out.println("Account Number       : ");
    "+account.getAccNo());
    System.out.println("Account Holder Name : ");
    "+account.getAccHolderName());
    System.out.println("Account Type         : ");
    "+account.getAccType());
    System.out.println("Account Balance      : ");
    "+account.getBalance());
}

```

```

        System.out.println("SkillId\tSkillName");
        System.out.println("-----");
        for(int index = 0; index < skillSet.length; index++){
            Skill skill = skillSet[index];
            System.out.print(skill.getSkillId()+"\t");
            System.out.print(skill.getSkillName()+"\n");
        }
    }
}

Main.java
import com.durgasoft.entities.Account;
import com.durgasoft.entities.Employee;
import com.durgasoft.entities.Skill;

public class Main {
    public static void main(String[] args) {
        Account account = new Account();
        account.setAccNo("a111");
        account.setAccHolderName("Durga");
        account.setAccType("Savings");
        account.setBalance(50000);

        Skill skill1 = new Skill();
        skill1.setSkillId("S-111");
        skill1.setSkillName("JAVA");

        Skill skill2 = new Skill();
        skill2.setSkillId("S-222");
        skill2.setSkillName("PYTHON");

        Skill skill3 = new Skill();
        skill3.setSkillId("S-333");
        skill3.setSkillName(".NET");

        Skill[] skillSet = {skill1, skill2, skill3};
    }
}

```

```
        Employee employee = new Employee(111, "Durga", 50000, "Hyd",
account, skillSet);
        employee.getEmployeeDetails();
    }
}
```

Q) Write a Java program to represent the entities like Movie, Reviews and Critics along with their associations?

Ans:

Q) Write a Java Program to represent the entities Institute , Students, Courses and Trainers along with their associations?

Ans:

In Object Orientation , all the associations are represented in the form of the following two ways.

1. Composition
2. Aggregation

Q)What are the differences between Composition and Aggregation?

Ans:

Composition is a Strong association between entities, where if we close a Container entity , automatically Contained entity will be closed, where Contained entity lifetime is depending on the Container entity lifetime.

EX: The association between Library and Books is Composition, because If we close the Library then there is no chance for Books to exist.

Aggregation is a weak association between entities, where if we close a Container entity then there is a chance for the C0ntained entity to exist, where the lifetime of the Contained entity is not dependent on the lifetime of the container.

EX: The association between Library and Students is Aggregation, because even if we close the Library then there is a chance for students to exist outside of the Library.

Inheritance in Java:

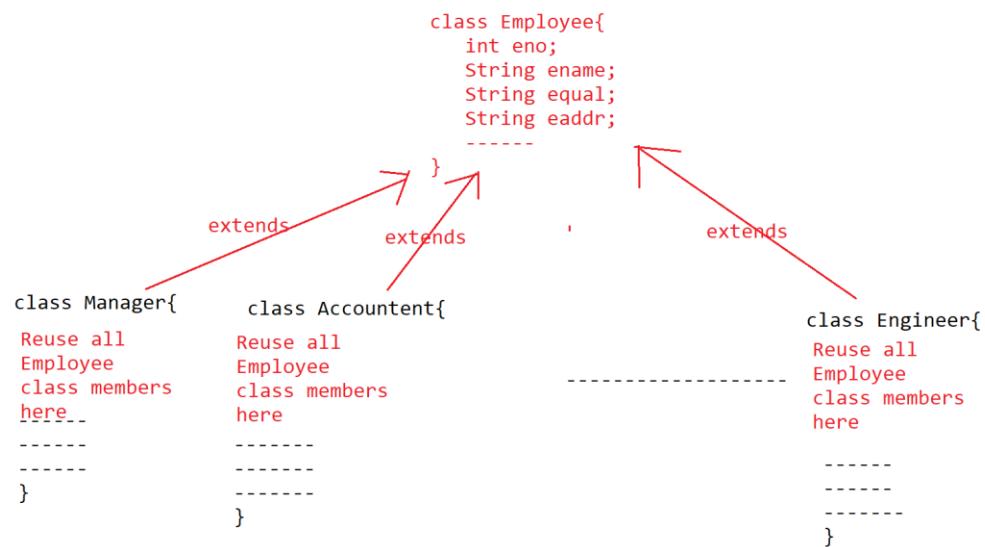
It is a relation between classes, it will provide variables and methods from one class to another class , where the class which is providing variables and methods to the another class is called “Superclass” and the class which is getting variables and methods from a superclass is called “Subclass”.

The main Advantage of Inheritance is “Code Reusability”.

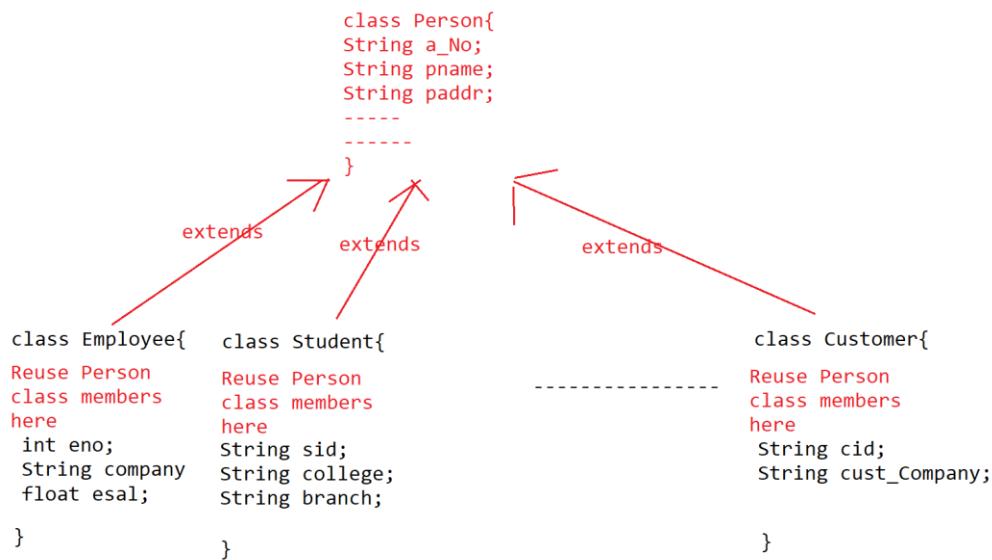
In inheritance, Declare variables and methods in the superclass and reuse these variables and methods in the sub classes.

EX1: In an Organization we are able to have a number of Employees like Manager, Accountant, Engineer,..... , where all these

Employees are having common properties and Behaviors like eno, ename, esal, equal,..... If we declare them separately at each and every class then Code duplication will be increased, where to improve Code Reusability we have to declare a super class for all the classes and we have to declare all the common properties and behaviors in the super class and reuse these common properties and behaviors in the subclasses without re-declaring them.



EX2:



In the Object orientation , there are two types of inheritances.

1. Single Inheritance
2. Multiple Inheritance

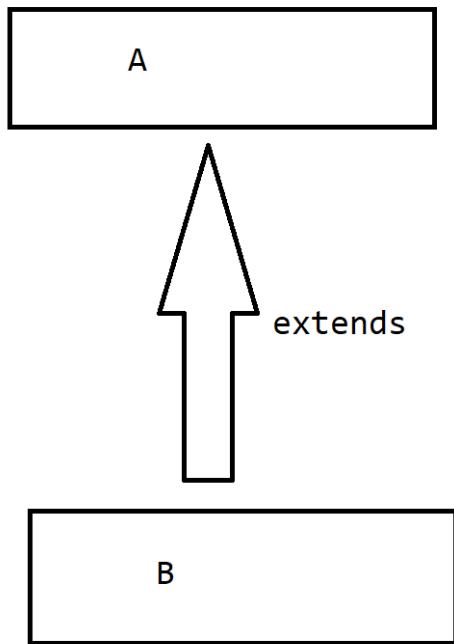
On the basis of the above two types of Inheritances, there are three more inheritances.

1. Multi Level Inheritance.
2. Hierarchical Inheritance.
3. Hybrid Inheritance.

Single Inheritance:

It is a relation between classes, it will provide variables and methods from only one superclass to one or more subclasses.

Java does support Single Inheritance.



EX:

```
class A{
    int i = 10;
    void m1(){
        System.out.println("m1-A");
    }
}
class B extends A{
    int j = 20;
    void m2(){
        System.out.println("m2-B");
        System.out.println(i);
        m1();
    }
}
public class Main {
    public static void main(String[] args) {
        A a = new A();
        System.out.println(a.i);
        a.m1();
```

```
//System.out.println(a.j); ---> Error  
//a.m2();-----> Error  
  
B b = new B();  
System.out.println(b.i);  
System.out.println(b.j);  
b.m1();  
b.m2();  
  
}  
}
```

2. Multiple Inheritance:

It is a relation between classes, it will provide variables and methods from more than one superclass to one or more subclasses.

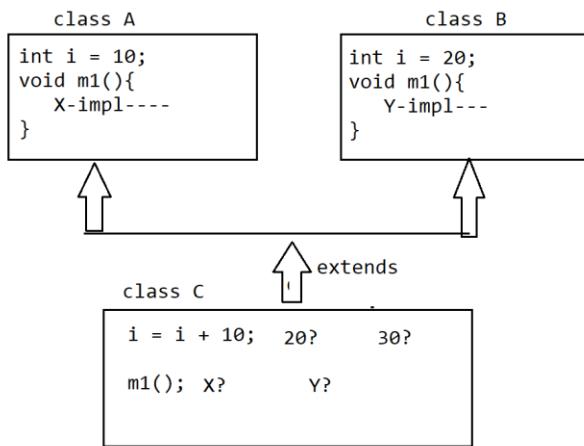
Java does not support Multiple Inheritance.

In the case of Multiple Inheritance, if we declare the same variable with different values and the same method with different implementations in the two super classes and if we access that variable and that method in the respective subclass then Compiler and JVM will get confusion that from which superclass that variable and method are accessed, but JAVA is a Simple programming language, so JAVA does not allow multiple inheritance.

To avoid Multiple inheritance in java applications , JAVA has defined “extends” keyword to allow only one super class name , not to allow more than superclass name in a class syntax.

```
class A extends B,C{-----> Invalid  
}
```

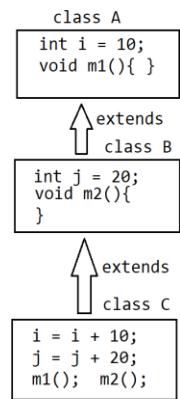
```
class A extends B{ ---> Valid  
}
```



Multi Level Inheritance:

It is the combination of single inheritances in more than one level.

Java does support Multiple level Inheritance.



EX:

```

class A{
    int i = 10;
}

```

```
void m1(){
    System.out.println("m1-A");
}
}

class B extends A{
    int j = 20;
    void m2(){
        System.out.println("m2-B");
    }
}

class C extends B{
    int k = 30;
    void m3(){
        System.out.println("m3-C");
    }
}

public class Main {
    public static void main(String[] args) {
        A a = new A();
        System.out.println(a.i);
        a.m1();
        System.out.println();

        B b = new B();
        System.out.println(b.i);
        System.out.println(b.j);
        b.m1();
        b.m2();
        System.out.println();

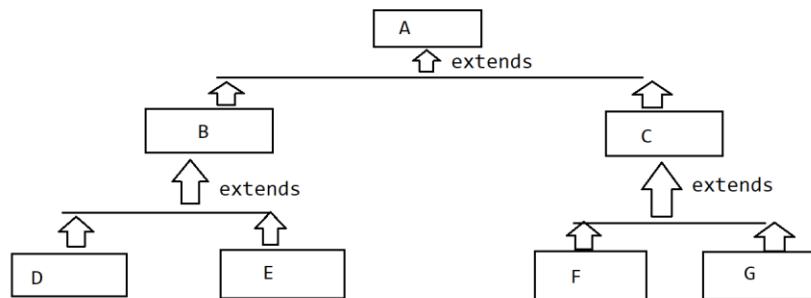
        C c = new C();
        System.out.println(c.i);
        System.out.println(c.j);
        System.out.println(c.k);
        c.m1();
        c.m2();
        c.m3();
    }
}
```

```
    }  
}
```

Hierarchical Inheritance:

It is the combination of Single Inheritances in a particular structure like Binary Tree Structure.

Java does support Hierarchical Inheritance.



```
class A{  
    int i = 10;  
    void m1(){  
        System.out.println("m1-A");  
    }  
}  
class B extends A{  
    int j = 20;  
    void m2(){  
        System.out.println("m2-B");  
    }  
}  
class C extends A{
```

```
int k = 30;
void m3(){
    System.out.println("m3-C");
}
}
class D extends B{
    int l = 40;
    void m4(){
        System.out.println("m4-D");
    }
}
class E extends B{
    int m = 50;
    void m5(){
        System.out.println("m5-E");
    }
}
class F extends C{
    int n = 60;
    void m6(){
        System.out.println("m6-F");
    }
}
class G extends C{
    int o = 70;
    void m7(){
        System.out.println("m7-G");
    }
}
public class Main {
    public static void main(String[] args) {
        A a = new A();
        System.out.println(a.i);
        a.m1();
        System.out.println();

        B b = new B();
        System.out.println(b.i);
    }
}
```

```
System.out.println(b.j);
b.m1();
b.m2();
System.out.println();

C c = new C();
System.out.println(c.i);
System.out.println(c.k);
c.m1();
c.m3();
System.out.println();

D d = new D();
System.out.println(d.i);
System.out.println(d.j);
System.out.println(d.l);
d.m1();
d.m2();
d.m4();
System.out.println();

E e = new E();
System.out.println(e.i);
System.out.println(e.j);
System.out.println(e.m);
e.m1();
e.m2();
e.m5();
System.out.println();

F f = new F();
System.out.println(f.i);
System.out.println(f.k);
System.out.println(f.n);
f.m1();
f.m3();
f.m6();
System.out.println();
```

```

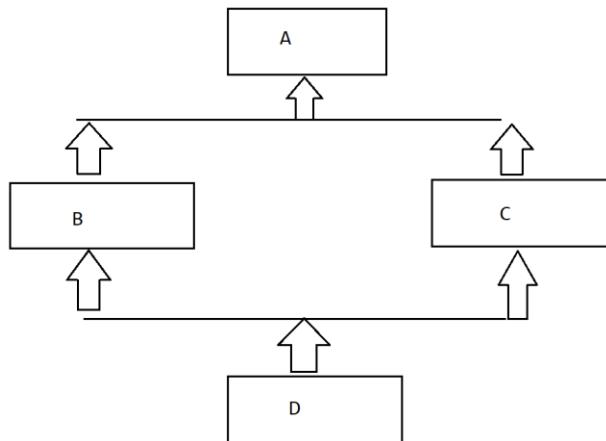
G g = new G();
System.out.println(g.i);
System.out.println(g.k);
System.out.println(g.o);
g.m1();
g.m3();
g.m7();
}
}

```

Hybrid Inheritance:

It is the combination of Single Inheritance and Multiple Inheritance.

Java does not support Hybrid Inheritance, because Hybrid Inheritance includes Multiple Inheritance.



IN the case of inheritance, we are able to access all super class members inside the subclasses directly without using any

reference variable, but subclass members are not available to the super classes.

```
class A{
    int i = 10;
    void m1(){
        System.out.println("m1-A");
        //System.out.println(j);--> Error
        //m2(); -----> Error
    }
}
class B extends A{

    void m2(){
        System.out.println("m2-B");
        System.out.println(i);
        m1();
    }
}
public class Main {
    public static void main(String[] args) {
        B b = new B();
        b.m2();
    }
}
```

In Java applications, by using superclass reference variables we are able to access only superclass members, we are unable to access subclass members, but by using subclass reference variables we are able to access both superclass members and subclass members.

```
class A{
    int i = 10;
    void m1(){
        System.out.println("m1-A");
    }
}
```

```

}

class B extends A{
    int j = 20;
    void m2(){
        System.out.println("m2-B");
    }
}

public class Main {
    public static void main(String[] args) {
        A a = new A();
        System.out.println(a.i);
        a.m1();
        //System.out.println(a.j);---> Error
        //a.m2(); ----> Error

        B b = new B();
        System.out.println(b.i);
        System.out.println(b.j);
        b.m1();
        b.m2();
    }
}

```

Static Context Inheritance:

IN Java applications, static context is represented in the form of the following three elements.

1. Static Variables
2. Static Methods
3. Static Blocks

Where Static Variables and Static blocks are recognized and executed at the time of loading the respective class bytecode to the memory, where Static methods are recognized and executed the moment when we access that method.

In Inheritance, when we create an object for a subclass, JVM has to load subclass bytecode to the memory, but as per the java rules and regulations JVM has to load superclass bytecode before loading sub class bytecode.

In the case of inheritance, JVM will load all the classes bytecode from super class to subclass order. In this context, if we provide static context in all the classes in inheritance then JVM will execute super class static context first then JVM has to execute subclass static context next.

EX:

```
class A{
    static{
        System.out.println("SB-A");
    }
}
class B extends A{
    static{
        System.out.println("SB-B");
    }
}
class C extends B{
    static{
        System.out.println("SB-C");
    }
}
public class Main {
    public static void main(String[] args) {
        C c = new C();
    }
}
```

OP:

SB-A
SB-B
SB-C

EX:

```
class A{
    static{
        System.out.println("SB-A");
    }
    static int m1(){
        System.out.println("m1-A");
        return 10;
    }
    static int i = m1();
}

class B extends A{
    static int j = m2();
    static int m2(){
        System.out.println("m2-B");
        return 20;
    }
    static{
        System.out.println("SB-B");
    }
}

class C extends B{
    static int m3(){
        System.out.println("m3-C");
        return 30;
    }
    static int k = m3();
    static{
        System.out.println("SB-C");
    }
}

public class Main {
    public static void main(String[] args) {
        C c1 = new C();
        C c2 = new C();
    }
}
```

OP
SB-A
m1-A
m2-B
SB-B
m3-C
SB-C

Instance Context in Inheritance:

In Java, instance context is represented in the form of the following three elements.

1. Instance Variables
2. Instance Methods
3. Instance Blocks

Instance variables and instance blocks are recognized and executed just before executing the respective class constructor, but instance methods are recognized and executed the moment when we access that method.

In the case of inheritance, when we create an object for a subclass then JVM has to execute subclass constructor, but as per the rules and regulations of java, before executing subclass constructor JVM has to execute superclass 0-arg constructor.

In the case of inheritance, Constructors execution order is from superclass to subclass, but in superclasses JVM will execute only 0-arg constructor, in the case if 0-arg constructor is not available in super classes then Compiler will raise an error.

EX:

```
class A{
```

```

A(){
    System.out.println("A-Con");
}
}

class B extends A{
B(){
    System.out.println("B-Con");
}
}

class C extends B{
C(){
    System.out.println("C-Con");
}
}

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

```

OP:

A-Con
B-Con
C-Con

EX:

```

class A{
A(){
    System.out.println("A-Con");
}
}

class B extends A{
B(int i){
    System.out.println("B-Con");
}
}

class C extends B{

```

```

C(){
    System.out.println("C-Con");
}
}

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

```

Status: Compilation Error

EX:

```

class A{
    A(){
        System.out.println("A-Con");
    }
}

class B extends A{
    B(int i){
        System.out.println("B-int-param-Con");
    }
    B(){
        System.out.println("B-Con");
    }
}

class C extends B{
    C(){
        System.out.println("C-Con");
    }
}

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

```

OP:

A-Con

B-Con

C-Con

EX:

```
class A{
    A(){
        System.out.println("A-Con");
    }
}
class B extends A{

}
class C extends B{
    C(){
        System.out.println("C-Con");
    }
}
public class Main {
    public static void main(String[] args) {
        C c = new C();
    }
}
```

OP:

A-Con

C-Con

EX:

```
class A{
    A(){
        System.out.println("A-Con");
    }
    {
        System.out.println("IB-A");
    }
    int i = m1();
    int m1(){
        System.out.println("m1-A");
    }
}
```

```

        return 10;
    }
}
class B extends A{
{
    System.out.println("IB-B");
}
int j = m2();
int m2(){
    System.out.println("m2-B");
    return 20;
}
B(){
    System.out.println("B-Con");
}
}
class C extends B{
int m3(){
    System.out.println("m3-C");
    return 30;
}
{
    System.out.println("IB-C");
}
C(){
    System.out.println("C-Con");
}
int k = m3();
}
public class Main {
public static void main(String[] args) {
    C c = new C();
}
}

```

OP:

IB-A

m1-A

A-Con

IB-B

m2-B

B-Con

IB-C

m3-C

C-Con

EX

```
class A{
    {
        System.out.println("IB-A");
    }
    int i = m1();
    A(){
        System.out.println("A-Con");
    }
    int m1(){
        System.out.println("m1-A");
        return 10;
    }
}
class B extends A{
    B(){
        System.out.println("B-Con");
    }
    int j = m2();
    {
        System.out.println("IB-B");
    }
    int m2(){
        System.out.println("m2-B");
        return 20;
    }
}
```

```
}

class C extends B{
    int m3(){
        System.out.println("m3-C");
        return 30;
    }
    C(){
        System.out.println("C-Con");
    }
    {
        System.out.println("IB-C");
    }
    int k = m3();
}

public class Main {
    public static void main(String[] args) {
        C c1 = new C();
        System.out.println();

        C c2 = new C();
    }
}
```

OP:
IB-A
m1-A
A-Con
m2-B
IB-B
B-Con
IB-C
m3-C
C-Con

IB-A
m1-A
A-Con

m2-B

IB-B

B-Con

IB-C

m3-C

C-Con

EX:

```
class A{
    static{
        System.out.println("SB-A");
    }
    A(){
        System.out.println("A-Con");
    }
    int m1(){
        System.out.println("m1-A");
        return 10;
    }
    static int i = m2();
    {
        System.out.println("IB-A");
    }
    static int m2(){
        System.out.println("m2-A");
        return 20;
    }
    int j = m1();
}
class B extends A{
    static{
        System.out.println("SB-B");
    }
```

```
{  
    System.out.println("IB-B");  
}  
static int m3(){  
    System.out.println("m3-B");  
    return 30;  
}  
int m4(){  
    System.out.println("m4-B");  
    return 40;  
}  
static int k = m3();  
int l = m4();  
B(){  
    System.out.println("B-Con");  
}  
}  
}  
class C extends B{  
    static{  
        System.out.println("SB-C");  
    }  
    static int m5(){  
        System.out.println("m5-C");  
        return 50;  
    }  
    static int m = m5();  
    C(){  
        System.out.println("C-Con");  
    }  
    {  
        System.out.println("IB-C");  
    }  
    int m6(){  
        System.out.println("m6-C");  
        return 60;  
    }  
    int n = m6();
```

```
}

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

OP:

SB-A

m2-A

SB-B

m3-B

SB-C

m5-C

IB-A

m1-A

A-Con

IB-B

m4-B

B-Con

IB-C

m6-C

C-Con

EX:

```
class A{
    static{
        System.out.println("SB-A");
    }
    A(){
        System.out.println("A-Con");
    }
    int m1(){

```

```
        System.out.println("m1-A");
        return 10;
    }
    static int i = m2();
{
    System.out.println("IB-A");
}
static int m2(){
    System.out.println("m2-A");
    return 20;
}
int j = m1();
}
class B extends A{
    static{
        System.out.println("SB-B");
    }
{
    System.out.println("IB-B");
}
static int m3(){
    System.out.println("m3-B");
    return 30;
}
int m4(){
    System.out.println("m4-B");
    return 40;
}
static int k = m3();
int l = m4();
B(){
    System.out.println("B-Con");
}
}
class C extends B{
    static{
        System.out.println("SB-C");
    }
}
```

```

    }
    static int m5(){
        System.out.println("m5-C");
        return 50;
    }
    static int m = m5();
    C(){
        System.out.println("C-Con");
    }
    {
        System.out.println("IB-C");
    }
    int m6(){
        System.out.println("m6-C");
        return 60;
    }
    int n = m6();
}
public class Main {
    public static void main(String[] args) {
        C c1 = new C();
        System.out.println();
        C c2 = new C();
    }
}

```

OP:

SB-A
 m2-A
 SB-B
 m3-B
 SB-C
 m5-C

IB-A

m1-A

A-Con

IB-B

m4-B

B-Con

IB-C

m6-C

C-Con

IB-A

m1-A

A-Con

IB-B

m4-B

B-Con

IB-C

m6-C

C-Con

‘super’ Keyword in Java

‘super’ is a Java keyword, it is able to refer to superclass objects from subclasses.

In Java applications, there are three ways to utilize super keyword.

1. To refer superclass Variables
2. To refer superclass methods
3. To refer superclass constructors

To refer superclass variables:

If we want to refer to superclass variables from subclasses by using ‘super’ keyword then we have to use the following syntax.

```
super.varName
```

Note: In java applications we will use the 'super' keyword to refer to superclass variables over the subclass variables when we have the same name for the subclass variables and superclass variables.

EX:

```
class A{
    int i = 10;
    int j = 20;
}
class B extends A{
    int i = 30;
    int j = 40;
    B(int i, int j){
        System.out.println(i+" "+j);
        System.out.println(this.i+" "+this.j);
        System.out.println(super.i+" "+super.j);
    }
}
public class Main {
    public static void main(String[] args) {
        B b = new B(50,60);
    }
}
```

OP:

```
50 60
30 40
10 20
```

EX:

```
class A{
    int i = 10;
    int j = 20;
```

```

}

class B extends A{
    int i1 = 30;
    int j1 = 40;
    B(int i1, int j1){
        System.out.println(i1 + j1);
        System.out.println(this.i1 + this.j1);
        System.out.println(super.i1 + super.j1);
    }
}
public class Main {
    public static void main(String[] args) {
        B b = new B(50,60);
    }
}

```

OP:

```

10 20
10 20
10 20

```

To refer to super class method:

If we want to refer to the superclass method from subclass by using super keyword then we have to use the following syntax.

```
super.methodName([ParamList]);
```

Note: In Java applications when we have the same method at subclass and at superclass, where to refer to the superclass method over the subclass method we have to use the “super” keyword.

EX:

```
class A{
```

```

void m1(){
    System.out.println("m1-A");
}
}

class B extends A{
    void m2(){
        System.out.println("m2-B");
        m1();
        this.m1();
        super.m1();
    }
    void m1(){
        System.out.println("m1-B");
    }
}

public class Main {
    public static void main(String[] args) {
        B b = new B();
        b.m2();
    }
}

```

OP:

m2-B
m1-B
m1-B
m1-A

To refer superclass constructors:

If we want to refer to superclass constructors from subclasses by using the ‘super’ keyword then we have to use the following syntax.

`super([ParamList]);`

In Java applications, when we access sub class constructor, first JVM must execute 0-arg constructor in superclass, after executing superclass 0-arg constructor only JVM will execute sub class constructor, in the case if 0-arg constructor is not available in the superclass and if we have a parameterized constructor in the superclass then compiler will raise an error.

In the above context, if we want to execute a particular constructor in place of 0-arg constructor in the superclass before subclass constructor execution then we have to use the “super” keyword in the subclass constructor.

EX:

```
class A{
    A(int i){
        System.out.println("A-int-param-Con");
    }
    A(){
        System.out.println("A-Con");
    }
}
class B extends A{
    B(){
        super(10);
        System.out.println("B-Con");
    }
}
public class Main {
    public static void main(String[] args) {
        B b = new B();
    }
}
```

OP:

```
A-int-param-Con
B-Con
```

If we want to use the ‘super’ keyword to refer to superclass constructors then we have to use the following conditions.

1. The respective super statement must be provided as a first statement.

EX:

```
class A{
    A(int i){
        System.out.println("A-int-param-Con");
    }
    A(){
        System.out.println("A-Con");
    }
}
class B extends A{
    B(){
        System.out.println("B-Con");
        super(10);
    }
}
public class Main {
    public static void main(String[] args) {
        B b = new B();
    }
}
```

Status: Compilation Error

2. super() statement must be provided in the subclass constructor only, not in normal java methods.

EX:

```
class A{
    A(int i){
        System.out.println("A-int-param-Con");
    }
    A(){
        System.out.println("A-Con");
    }
}
```

```

    }
}

class B extends A{
    B(){
        System.out.println("B-Con");
    }
    void m1(){
        super(10);
        System.out.println("m1-B");
    }
}
public class Main {
    public static void main(String[] args) {
        B b = new B();
    }
}

```

Status: Compilation Error

Q) Is it possible to refer to more than one superclass constructor from a single subclass constructor by using the “super” keyword?

Ans:

No, it is not possible to refer to more than one superclass constructor from a single subclass constructor by using super keyword, because super statement must be provided as the first statement in the subclass constructors.

EX:

```

class A{
    A(){
        System.out.println("A-Con");
    }
    A(int i){
        System.out.println("A-int-param-Con");
    }
}

```

```

    }
    A(float f){
        System.out.println("A-float-param-con");
    }

}

class B extends A{
    B(){
        super(10);
        super(22.22f);
        System.out.println("B-Con");
    }

}

public class Main {
    public static void main(String[] args) {
        B b = new B();
    }
}

```

Status: Compilation Error

Consider the following example.

```

class A{
    A(){
        System.out.println("A-Con");
    }
}

class B extends A{
    B(){
        System.out.println("B-Con");
    }
}

class C extends B{
    C(){
        System.out.println("C-Con");
    }
}

```

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

IN the above example, JVM will execute a 0-arg constructor in the superclass before executing sub class constructor due to the below internal reason.

1. Compiler Will go to each and every class , compiler will check whether any constructor exists explicitly or not, if no constructor exists explicitly then compiler will provide default constructor, if at least one constructor exists explicitly then compiler will not provide default constructor.
2. Compiler will go to each and every constructor at each and every class, where compiler will check whether any super() statement is available or not explicitly , if no super() statement is available explicitly then compiler will add a super() statement to the constructor in such a way that to access 0-arg constructor in the superclass. In the case if we provide any super() statement explicitly at any constructor then the compiler will not add a super() statement.
3. Compiler will check whether all the superclasses are having right constructors or not as per the super() statements which we have in the respective subclass constructors, if any superclass is not having right constructor as per the super() statement which we have in the subclass constructor then compiler will raise an error.

If we execute the above program then JVM will perform the following actions.

1. JVM access subclass constructor as part of subclass object creation.

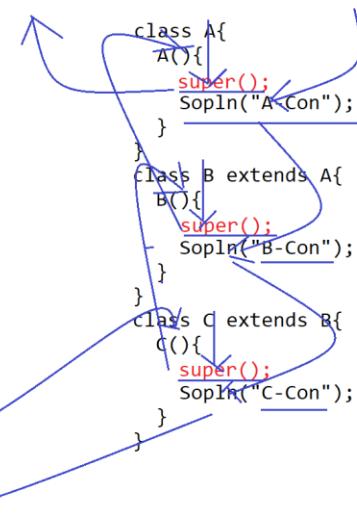
2. JVM will go inside the subclass constructor and it access super() statement.
3. As per the super() statement call in sub class constructor, JVM will go and execute the respective superclass constructor.
4. After executing the superclass constructor JVM will come back to the subclass constructor and execute the remaining part of the subclass constructor.

```
class A{
  A(){}
  Sopln("A-Con");
}
class B extends A{
  B(){}
  Sopln("B-Con");
}
class C extends B{
  C(){}
  Sopln("C-Con");
}
```

```
class Test{
  p s va main(String[] args){
    C c = new C();
  }
}
```

Compilation

A-Con
B-Con
C-Con



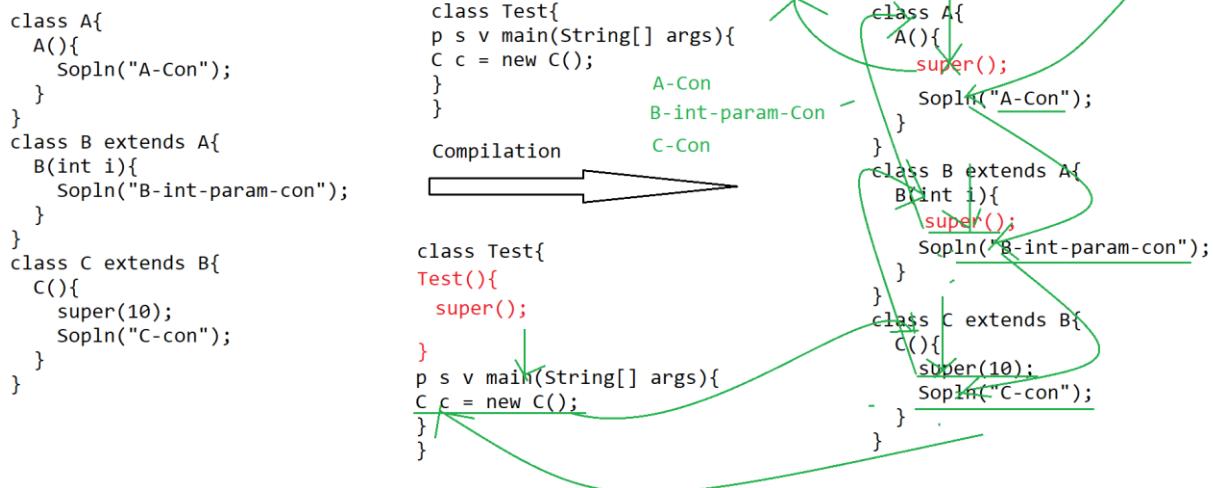
```
class A{
  A(){}
  Sopln("A-Con");
}
class B extends A{
  B(int i){
    Sopln("B-int-param-con");
  }
}
class C extends B{
  C(){}
  Sopln("C-Con");
}
```

```
class Test{
  p s v main(String[] args){
    C c = new C();
  }
}
```

Compilation

Compilation Error

```
class A{
  A(){}
  super();
  Sopln("A-Con");
}
class B extends A{
  B(int i){
    super();
    Sopln("B-int-param-con");
  }
}
class C extends B{
  C(){}
  super();
  Sopln("C-Con");
}
```



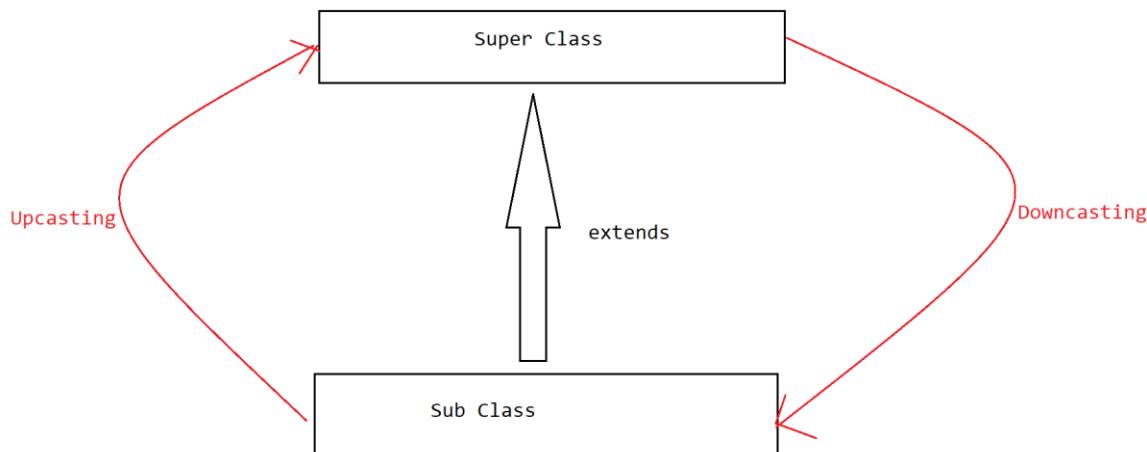
Class Level Type Casting:

The process of converting data from one user defined data type to another user defined data type is called “User Defined data Types Type casting” or simply Class Level Type Casting.

If we want to perform User defined data types Type casting we must have either extends relation or implements relation between two user defined data types.

There are two types of Class level type casting.

1. Upcasting
2. Downcasting



Upcasting:

The process of converting data from subclass type to superclass type is called Upcasting.

If we want to perform upcasting then we have to assign subclass reference variables to superclass reference variables.

EX:

```

class A{
    void m1(){
        System.out.println("m1-A");
    }
}
class B extends A{
    void m2(){
        System.out.println("m2-B");
    }
}
public class Main {
    public static void main(String[] args) { 
```

```

B b = new B();
b.m1();
b.m2();
A a = b;// Upcasting...
a.m1();

A a1 = new B();// Upcasting
a1.m1();
}
}

```

When we compile the above code, the compiler will perform the following actions.

Compiler will check whether the right side variable data type is compatible with the left side variable data type or not, if not compiler will raise an error, if the right side variable data type is compatible with left side variable data type then compiler will not raise any error and compiler will not perform any type casting.

Note: IN Java , all subclass types are compatible with the super class types, we can assign subclass reference variables to superclass reference variables directly, but superclass types are not compatible with subclass types directly, we are unable to assign superclass reference variables to subclass reference variable.

When we execute the above program JVM will perform the following actions.

1. JVM will convert the right side variable data type to the left side variable data type internally.
2. JVM will copy the value from the right side variable to the left side variable.

In Java applications, if we create an object in the subclass , automatically all the superclass members and subclass members will be stored in the subclass object.

In the above context,we want to access only superclass members not subclass members, to achieve this requirement we have to declare reference variable for Superclass, not for subclass, if we declare reference variable for subclass then we are able to access both superclass members and subclass members.

Creating an object for a subclass and declaring a reference variable for superclass is “Upcasting”.

In Java applications, in Method overriding, we must override superclass method with subclass method, to perform method overriding we must create an object for the subclass and we must create a reference variable for superclass.

To prove method overriding we must access the superclass method but we have to get output from the subclass method.

EX:

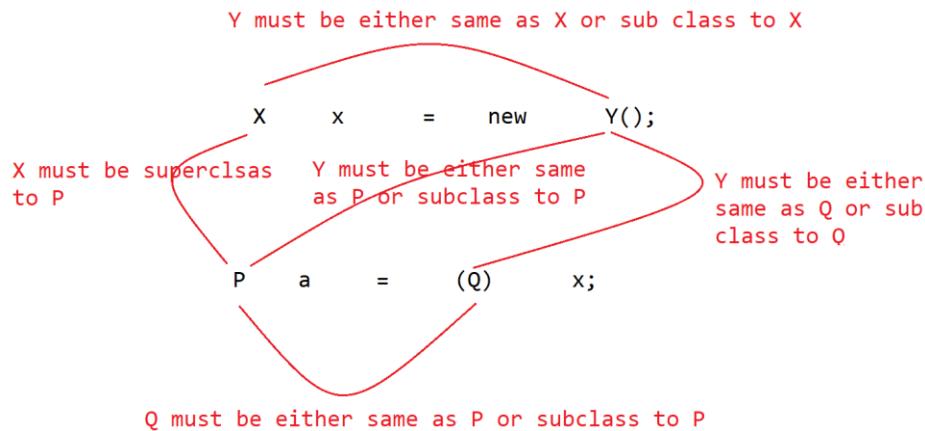
```
class A{
    void m1(){
        System.out.println("m1-A");
    }
}
class B extends A{
    void m1(){
        System.out.println("m1-B");
    }
}
public class Main {
    public static void main(String[] args) {
        A a = new B(); // Upcasting
        a.m1();
    }
}
```

}

Downcasting:

The process of converting data from superclass type to subclass type is called Downcasting.

To perform Downcasting we have to use the following pattern.



To perform downcasting we have to use the following cases.

```
class A{  
----  
}  
class B extends A{  
----  
}
```

Case#1:

```
A a = new A();  
B b = a;  
Status: Compilation Error
```

Reason: In Java applications, all subclass types are compatible with superclass types, so we can assign subclass reference variables to the superclass reference variables directly, but superclass types are not compatible with subclass types , we are unable to assign superclass reference variables to subclass reference variables directly, if we assign

superclass reference variables to the subclass reference variables directly then compiler will raise an error.

Note: If we want to assign a superclass reference variable to a subclass reference variable then we have to use the cast operator at the right side part of the expression.

Case#2:

```
A a = new A();  
B b = (B)a;
```

Status: No Compilation Error, but java.lang.ClassCastException

Reason:

In Java applications, we are able to keep subclass object reference value in superclass reference variable directly , but we are unable to keep superclass object reference value in subclass reference variable, if we are trying to keep superclass object reference value in subclass reference variable then JVM will raise an exception like java.lang.ClassCastException

Case#3:

```
A a = new B();  
B b = (B)a;
```

Status: No Compilation Error, No Exception

EX:

```
class A{  
    void m1(){  
        System.out.println("m1-A");  
    }  
}  
class B extends A{  
    void m2(){  
        System.out.println("m2-B");  
    }  
}  
public class Main {
```

```

public static void main(String[] args) {
    /*
    A a = new A();
    B b = a;
    */
    /*
    A a = new A();
    B b = (B)a;
    */
    A a = new B();
    B b = (B)a;
    b.m1();
    b.m2();

}

}

```

EX:

```

class Account{

}
class SavingsAccount extends Account{
    public void getSavingsAccountDetails(){
        System.out.println("This is Savings Account");
    }
}
class CurrentAccount extends Account{
    public void getCurrentAccountDetails(){
        System.out.println("This is current Account");
    }
}
class Employee{
    private int eno;
    private String ename;

```

```
private String edes;
private Account account;

public int getEno() {
    return eno;
}

public void setEno(int eno) {
    this.eno = eno;
}

public String getEname() {
    return ename;
}

public void setEname(String ename) {
    this.ename = ename;
}

public String getEdes() {
    return edes;
}

public void setEdes(String edes) {
    this.edes = edes;
}

public Account getAccount() {
    Account account = null;
    if(edes.equals("CEO")){
        account = new CurrentAccount();
    }else{
        account = new SavingsAccount();
    }
    return account;
}
```

```
}

}

public class Main {
    public static void main(String[] args) {
        Employee employee1 = new Employee();
        employee1.setEno(111);
        employee1.setEname("Anil");
        employee1.setEdes("Manager");

        System.out.println("Employee Details");
        System.out.println("-----");
        System.out.println("Employee Number : " +
"+employee1.getEno());
        System.out.println("Employee Name : " +
"+employee1.getEname());
        System.out.println("Employee Des : " +
"+employee1.getEdes());
        Account account1 = employee1.getAccount();
        SavingsAccount savingsAccount = (SavingsAccount)
account1;
        savingsAccount.getSavingsAccountDetails();
        System.out.println();

        Employee employee2 = new Employee();
        employee2.setEno(222);
        employee2.setEname("Durga");
        employee2.setEdes("CEO");
        System.out.println("EmployeeDetails");
        System.out.println("-----");
        System.out.println("Employee Number : " +
"+employee2.getEno());
        System.out.println("Employee Name : " +
"+employee2.getEname());
```

```

        System.out.println("Employee Des      :");
        "+employee2.getEdes());
        Account account2 = employee2.getAccount();
        CurrentAccount currentAccount = (CurrentAccount)
account2;
        currentAccount.getCurrentAccountDetails();

    }
}

```

Consider the following inheritance Model

```

class A{
}
class B extends A{
}
class C extends B{
}
class D extends C{
}

```

EX1:

```

A a = new B();
B b = a;
Status: Compilation Error.

```

EX2:

```

A a = new A();
B b = (B)a;
Status: No Compilation Error, ClassCastException

```

EX3:

```

A a = new B();
B b = (B)a;

```

Status: No Compilation Error, No Exception

EX4:

```
A a = new B();
```

```
B b = (C)a;
```

Status: No Compilation Error, ClassCastException

EX5:

```
A a = new C();
```

```
B b = (C)a;
```

Status: No Compilation Error, No ClassCastException

EX6:

```
A a = new D();
```

```
B b = (C)a;
```

Status: No Compilation Error, No ClassCastException

EX7:

```
A a = new C();
```

```
B b = (D)a;
```

Status: No Compilation Error, ClassCastException

EX8:

```
A a = new D();
```

```
D d = (D)(C)(B)a;
```

Status: No Compilation Error, No ClassCastException

EX9:

```
A a = new C();
```

```
C c = (C)(D)(B)a;
```

Status: No Compilation Error, ClassCastException

USES-A Relationship:

It is a relation between classes, it uses an entity up to a particular behavior of another entity.

EX: Transaction USES Account up to Deposit operation.

EX:

Account.java

```
package com.durgasoft.entities;

public class Account {
    private String accNo;
    private String accHolderName;
    private String accType;
    private double balance;

    public String getAccNo() {
        return accNo;
    }

    public void setAccNo(String accNo) {
        this.accNo = accNo;
    }

    public String getAccHolderName() {
        return accHolderName;
    }

    public void setAccHolderName(String accHolderName) {
        this.accHolderName = accHolderName;
    }

    public String getAccType() {
        return accType;
    }

    public void setAccType(String accType) {
        this.accType = accType;
    }
}
```

```

    }

    public double getBalance() {
        return balance;
    }

    public void setBalance(double balance) {
        this.balance = balance;
    }
}

```

Transaction.java

```

package com.durgasoft.entities;

public class Transaction {
    private String tranctionId;
    private String transactionName;

    public Transaction(String tranctionId, String
transactionName) {
        this.tranctionId = tranctionId;
        this.transactionName = transactionName;
    }

    public void deposit(Account account, int depAmount) {
        double totalBalance = account.getBalance() +
depAmount;
        account.setBalance(totalBalance);
        System.out.println("Transaction Details");
        System.out.println("-----");
        System.out.println("Transaction Id : " +
tranctionId);
        System.out.println("Transaction Type : " +
transactionName);
        System.out.println("Account Number : " +
account.getAccNo());
    }
}

```

```

        System.out.println("Account Holder Name : "
"+account.getAccHolderName());
        System.out.println("Account Type : "
"+account.getAccType());
        System.out.println("Total Balance : "
"+account.getBalance());
        System.out.println("Transaction Status : "
SUCCESS");
        System.out.println("=====ThanQ, Visit
Again=====");
    }
}

```

Main.java

```

import com.durgasoft.entities.Account;
import com.durgasoft.entities.Transaction;

public class Main {
    public static void main(String[] args) {
        Account account = new Account();
        account.setAccNo("abc123");
        account.setAccHolderName("Durga");
        account.setAccType("Savings");
        account.setBalance(50000);

        Transaction transaction = new Transaction("T-111",
"DEPOSIT");
        transaction.deposit(account, 15000);
    }
}

```

Polymorphism:

Polymorphism is a Greek word, where poly means Many and morphism means Structures or forms.

If one thing exists in more than one form then it is called Polymorphism.

The main advantage of Polymorphism is “Flexibility” in application development.

There are two types of Polymorphisms.

1. Static Polymorphism
2. Dynamic Polymorphism

Static Polymorphism

If the polymorphism is exhibited at compilation time then that polymorphism is called Static Polymorphism.

EX: Method Overloading.

Dynamic Polymorphism:

If the polymorphism is exhibited at runtime then that polymorphism is called Dynamic Polymorphism.

EX: Method Overriding.

Method Overloading:

The process of extending existing method functionality up to some new functionality is called Method Overloading.

In java applications , to perform method overloading we have to declare more than one method with the same name and with the different parameter list, that is a different method signature.

In the above context, parameter list differences may be available in either of the following ways.

1. Differences in the number of parameters.

```
void add(int i, int j){  
}  
void add(int i, int j, int k){  
}
```

2. Difference in the parameters data types.

```
void add(int i, int j){  
}  
void add(float f1, float f2){  
}
```

3. Difference in the order of the parameters.

```
void add(int i, float f){  
}  
void add(float f, int i){  
}
```

In Java applications , we are able to perform method overloading with or without the inheritance.

EX:

```
package org.example;  
  
class Calculator{  
    void add(int i, int j){  
        System.out.println("Add : "+(i+j));  
    }  
    void add(float f1, float f2){  
        System.out.println("Add : "+(f1+f2));  
    }  
}
```

```

    }
    void add(String str1, String str2){
        System.out.println("ADD : "+(str1+str2));
    }
}

public class Main {
    public static void main(String[] args) {
        Calculator cal = new Calculator();
        cal.add(10,20);
        cal.add(22.22f, 33.33f);
        cal.add("abc", "def");
    }
}

```

In the above method overloading, when we access the add method by passing the parameter values , here JVM will identify the right method on the basis of the number parameters or data types of the parameter values,... over the multiple methods with the same name , this process is called “Method Resolution”.

EX:

```

package org.example;
class Employee{
    void generateSalary(int basic, float hk, int ta, float
pf) {
        float salary = basic + (basic*hk)/100 + ta -
(basic*pf)/100;
        System.out.println("Salary : "+salary);
    }
    void generateSalary(int basic, float hk, int ta, float
pf, int bonus) {
        float salary = basic + (basic*hk)/100 + ta -
(basic*pf)/100 + bonus;
        System.out.println("Salary : "+salary);
    }
}

```

```
}

public class Main {
    public static void main(String[] args) {
        Employee employee = new Employee();
        employee.generateSalary(35000, 25.0f, 5000,
12.5f);
        employee.generateSalary(35000, 25.0f, 5000, 12.5f,
15000);
    }
}
```

Method Overriding:

The process of providing replacement for the existing method functionality with some other new method functionality is called Method Overriding.

To perform Method Overriding we need inheritance, where we have to define old functionality in the form of a superclass method and we have to define new functionality in the form of subclass method , here both superclass method prototype and subclass method prototype must be the same[Relatively same].

Steps to perform Method Overriding:

1. Declare a superclass with a method with old functionality which we want to override.
2. Declare a subclass with the same superclass method with new functionality.
3. In the main class, in the main() method, access the superclass method but get output from the subclass method.

To prove method overriding we have to use the following cases.

```
class A{
```

```
void m1(){
    System.out.println("Old Functionality");
}
}

class B extends A{
    void m1(){
        System.out.println("New Functionality");
    }
}
```

Case#1:

```
A a = new A();
a.m1();
```

Status: No Method Overriding, because Method Overriding needs a subclass object not superclass object.

Case#2:

```
B b = new B();
b.m1();
```

Status: When we create a subclass object , automatically the subclass method overrides the superclass method internally, but to prove method overriding we must access superclass method, for this we need superclass reference variable , not subclass reference variable.

Case#3:

```
A a = new B();
a.m1();
```

Status: When we create a subclass object ,automatically a superclass method is overridden with the respective subclasses method , where if we access superclass method then JVM will identify the respective overriding subclass method and JVM will executed the respective subclass method.

EX:

```
class A{
    void m1() {
        System.out.println("Old Function");
    }
}

class B extends A{
    void m1() {
        System.out.println("New Function");
    }
}

public class Main {
    public static void main(String[] args) {
        A a = new B();
        a.m1();
    }
}
```

EX:

```
class DBApp{
    public void getDriver(){
        System.out.println("Type-1 Driver");
    }
}

class NewDBApp extends DBApp{
    public void getDriver() {
        System.out.println("Type-4 Driver");
    }
}

public class Main {
    public static void main(String[] args) {
        DBApp dbApp = new NewDBApp();
        dbApp.getDriver();
    }
}
```

Rules and Regulations to Perform Method Overriding:

1. In method overriding, the subclass method return type must be the same as the superclass method return type, otherwise the compiler will raise an error.

EX:

```
class A{
    int m1() {
        System.out.println("m1-A");
        return 10;
    }
}
class B extends A{
    void m1() {
        System.out.println("m1-B");
    }
}
public class Main {
    public static void main(String[] args) {
        A a = new B();
        a.m1();
    }
}
Status: Compilation Error.
```

EX:

```
class A{
    int m1() {
        System.out.println("m1-A");
        return 10;
    }
}
class B extends A{
    int m1() {
        System.out.println("m1-B");
        return 20;
    }
}
```

```

        }
    }

public class Main {
    public static void main(String[] args) {
        A a = new B();
        a.m1();
    }
}

Status: No Compilation Error.
OP: m1-B

```

2. In Method Overriding, the Superclass method must not be declared private, if we declare a superclass method as private then it is not possible to access in the main class, it will raise an error if we access the superclass method in main class.

EX:

```

class A{
    private void m1(){
        System.out.println("m1-A");
    }
}

class B extends A{
    void m1(){
        System.out.println("m1-B");
    }
}

public class Main {
    public static void main(String[] args) {
        A a = new B();
        a.m1();
    }
}

```

Status: Compilation Error

3. In method overriding, the superclass method must not be declared as final, but subclass method may be final.

EX:

```
class A{
    final void m1(){
        System.out.println("m1-A");
    }
}
class B extends A{
    void m1(){
        System.out.println("m1-B");
    }
}
public class Main {
    public static void main(String[] args) {
        A a = new B();
        a.m1();
    }
}
```

Status: Compilation Error, Overridden method is final

EX:

```
class A{
    final void m1(){
        System.out.println("m1-A");
    }
}
class B extends A{
    final void m1(){
        System.out.println("m1-B");
    }
}
public class Main {
    public static void main(String[] args) {
        A a = new B();
        a.m1();
    }
}
```

```
    }
}
```

Status: Compilation Error, Overridden method is final

EX:

```
class A{
    void m1() {
        System.out.println("m1-A");
    }
}
class B extends A{
    final void m1() {
        System.out.println("m1-B");
    }
}
public class Main {
    public static void main(String[] args) {
        A a = new B();
        a.m1();
    }
}
```

Status: No Compilation Error

OP: m1-B

4. In **method overriding**, either superclass method or subclass method or both superclass method and subclass method must not be declared as static, if we declare either superclass method or subclass method as static then compiler will raise an error, if we declare both the superclass method and subclass method as static then compiler will not raise any error, in this context, if we access superclass method from main class then JVM will execute only superclass method in place of subclass method, because superclass

method over hide subclass method, so it is called “Method Over hiding” in Java.

EX:

```
class A{
    static void m1(){
        System.out.println("m1-A");
    }
}
class B extends A{
    void m1(){
        System.out.println("m1-B");
    }
}
public class Main {
    public static void main(String[] args) {
        A a = new B();
        a.m1();
    }
}
```

Status: Compilation Error, Overridden method is static.

EX:

```
class A{
    void m1(){
        System.out.println("m1-A");
    }
}
class B extends A{
    static void m1(){
        System.out.println("m1-B");
    }
}
public class Main {
    public static void main(String[] args) {
```

```
        A a = new B();
        a.m1();
    }
}

Status: Compilation Error
```

EX:

```
class A{
    static void m1(){
        System.out.println("m1-A");
    }
}

class B extends A{
    static void m1(){
        System.out.println("m1-B");
    }
}

public class Main {
    public static void main(String[] args) {
        A a = new B();
        a.m1();
    }
}

Status: No Compilation Error
Op: m1-A
```

5. In method Overriding, subclass method scope must be either the same as superclass method scope or wider than the superclass method scope.

EX:

```
class A{
    public void m1(){
        System.out.println("m1-A");
    }
}

class B extends A{
```

```

protected void m1() {
    System.out.println("m1-B");
}
}

public class Main {
    public static void main(String[] args) {
        A a = new B();
        a.m1();
    }
}

```

Status: Compilation Error.

EX:

```

class A{
    protected void m1(){
        System.out.println("m1-A");
    }
}

class B extends A{
    public void m1(){
        System.out.println("m1-B");
    }
}

public class Main {
    public static void main(String[] args) {
        A a = new B();
        a.m1();
    }
}

```

Status : No Compilation Error

OP: m1-B

6. In Method Overriding, subclass method access privileges must be either the same as the superclass

method access privileges or weaker than the superclass method access privileges.

Q) What are the differences between Method Overloading and Method Overriding?

Ans:

1. Method Overloading is the process of extending existing method functionality to some other new functionality.

Method Overriding is the process of providing replacement for the existing method functionality with some other new functionality.

2. After the method overloading, both the method's functionalities are available to access.

After Method overriding, we are able to access only new functionality, we are unable to access old functionality.

3. To perform method overloading we have to declare more than one method with the same name and with the different parameter list.

To perform method overriding we have to provide the same method prototype in both superclass method and subclass method with the different functionality.

4. In Method overloading we have to provide different method signatures.

In Method Overriding , we must provide the same method prototypes at both superclass and subclass.

5. In java applications, we are able to perform method overloading with or without the inheritance.

In Java applications, we are able to perform method overriding with the inheritance only, without inheritance it is not possible to perform method overriding.

Consider the following program.

```
class A{
    public void m1(){
        System.out.println("10000 loc");
    }
}

class B extends A{
    public void m1(){
        System.out.println("100 loc");
    }
}

public class Main {
    public static void main(String[] args) {
        A a = new B();
        a.m1();
    }
}
```

In the above program , when we create subclass object, automatically superclass method is overridden with subclass method, where if we access superclass method JVM will execute subclass method only, in this context, providing superclass method body is unnecessary, here we need an alternative where we need to declare superclass method without the body .

In Java /J2EE applications, if we want to declare a method without the implementation then we have to declare that method as an abstract method.

IN JAVA/J2EE applications, if we want to declare a method as an abstract method then the respective class must be an abstract class.

EX:

```
abstract class A{
    public abstract void m1();
}

class B extends A{
    public void m1(){
        System.out.println("m1-B");
    }
}

public class Main {
    public static void main(String[] args) {
        B b = new B();
        b.m1();
    }
}
```

Q)What are the differences between concrete methods and abstract methods?

Ans:

1. Concrete methods will have both method declaration and method implementation.

Abstract methods will have only method declaration without the implementation.

2. Concrete methods are possible in classes and abstract classes.

Abstract methods are possible in abstract classes and in

interfaces.

3. Concrete methods will provide less shareability.

Abstract methods will provide more shareability.

Q)What are the differences between concrete classes and abstract classes?

Ans:

1. Concrete classes are able to allow only concrete methods.
Abstract classes are able to allow both concrete methods and abstract methods.
2. To declare concrete classes we have to use the “class” keyword.

To declare abstract classes we need “abstract” keyword along with “class” keyword.

3. For concrete classes we are able to provide both reference variables and objects.

For the abstract classes we are able to provide only reference variables, we are unable to create objects.

4. In the case of concrete classes, method overriding is optional.

In the case of abstract classes , if we have abstract methods then it is mandatory to perform method overriding.

5. Concrete classes are able to provide less shareability.
Abstract classes are able to provide more shareability.

EX:

```
abstract class A{
    void m1() {
        System.out.println("m1-A");
    }
    abstract void m2();
    abstract void m3();
}

class B extends A{
    void m2() {
        System.out.println("m2-B");
    }
    void m3() {
        System.out.println("m3-B");
    }
    void m4() {
        System.out.println("m4-B");
    }
}

public class Main {
    public static void main(String[] args) {
        //A a = new A();---> Error
        A a = new B();
        a.m1();
        a.m2();
        a.m3();
        //a.m4(); --> Error

        B b = new B();
        b.m1();
        b.m2();
        b.m3();
        b.m4();
    }
}
```

Q) Is it possible to declare an abstract class without the abstract methods?

Ans:

Yes, it is possible to declare an abstract class without the abstract methods.

EX:

```
abstract class A{
    void m1() {
        System.out.println("m1-A");
    }
    void m2() {
        System.out.println("m2-A");
    }
    void m3() {
        System.out.println("m3-A");
    }
}
class B extends A{

}
public class Main {
    public static void main(String[] args) {
        A a = new B();
        a.m1();
        a.m2();
        a.m3();
    }
}
```

Note: To declare a class as an abstract class then it is not mandatory to provide at least one abstract method, but if we want to declare a method as an abstract method then the respective class must be an abstract class.

Q) If we declare an abstract class with some abstract methods and if we provide implementation for some of the abstract methods[Not for all the abstract methods] in the subclass then what will happen in the Java program?

Ans:

In a java applications,if we declare an abstract class with some abstract methods then it is mandatory for the subclasses to provide implementation for all the abstract methods , if we miss the implementation for any abstract method in the subclass then compiler will raise an error, because subclass is a concrete class but it has the abstract methods from the abstract class.

To overcome the above problem either we have to provide implementation for all the abstract methods in the subclass or we have to declare the respective subclass as an abstract class and we must provide implementation for all the abstract methods by taking another subclass in multi-level inheritance.

EX:

```
abstract class A{
    abstract void m1();
    abstract void m2();
    abstract void m3();
}

abstract class B extends A{
    void m1() {
        System.out.println("m1-B");
    }
}

class C extends B{
    void m2() {
        System.out.println("m2-C");
    }
}
```

```

    }
    void m3() {
        System.out.println("m3-C");
    }
}

public class Main {
    public static void main(String[] args) {
        A a = new C();
        a.m1();
        a.m2();
        a.m3();
    }
}

```

Q) Is it possible to extend a concrete class to an abstract class?

Ans:

Yes, in java applications it is possible to extend an abstract class to a concrete class and a concrete class to an abstract class.

EX:

```

class A{
    void m1() {
        System.out.println("m1-A");
    }
}

abstract class B extends A{
    void m2() {
        System.out.println("m2-B");
    }
    abstract void m3();
    abstract void m4();
}

class C extends B{

```

```

void m3() {
    System.out.println("m3-C");
}
void m4() {
    System.out.println("m4-C");
}
}

public class Main {
    public static void main(String[] args) {
        A a = new C();
        a.m1();
        B b = new C();
        b.m1();
        b.m2();
        b.m3();
        b.m4();

        C c = new C();
        c.m1();
        c.m2();
        c.m3();
        c.m4();
    }
}

```

Q) In Java applications, we are unable to create objects for abstract classes , but we are able to provide constructor inside the abstract class. How is it possible to provide a constructor without creating objects for the abstract class?

Ans:

Inside the abstract classes constructors are required in order to recognize and store instances of context of the abstract class in the respective subclass objects.

Note: In Java applications, instance context will be recognized and executed or initialized just before executing the respective class constructor , to recognize and initialize instance context in a class we need a constructor.

EX:

```
abstract class A{
    int i = 10;
    int j = 20;
    A() {
        System.out.println("A-Con");
    }
}
class B extends A{
    int k = 30;
    int j = 40;
    B() {
        System.out.println("B-Con");
    }
}
public class Main {
    public static void main(String[] args) {
        B b = new B();
    }
}
```

OP:

A-Con
B-Con

Interfaces:

Q)What are the differences between classes, abstract classes and interfaces?

Ans:

1. Classes are able to allow only concrete methods.

Abstract classes are able to allow both concrete methods and abstract methods.

Interfaces are able to allow only abstract methods.

2. To declare a class we have to use the “class” keyword.

To declare an abstract class we have to use the “abstract” keyword along with the “class” keyword.

To declare an interface we have to use the “interface” keyword.

3. For concrete classes we are able to provide both reference variables and objects.

For abstract classes and interfaces we are able to provide only reference variables, we are unable to create objects.

4. Inside the interfaces, by default all variables are public static final , not required to declare explicitly.

In the case of concrete classes and abstract classes no default cases are available for the variables.

5. Inside the interfaces, by default all methods are public and abstract , no need to declare explicitly.

In the case of concrete classes and abstract classes no default cases are available for methods.

6. Concrete classes are able to provide less shareability.

Abstract classes are able to provide middle level shareability.

Interfaces are able to provide more shareability.

7. It is not possible to extend more than one class or more than one abstract class to a class or to an abstract class.

In Java applications, we are able to extend more than one interface to a single interface.

8. In the case of classes, method overriding is optional.

In the case of abstract classes and interfaces, if we have abstract methods then it is mandatory to override abstract methods in the subclasses or in the implementation classes.

9. Constructors are possible in concrete classes and abstract classes, but constructors are not possible in interfaces.

10. Static blocks are possible in concrete classes and abstract classes , but static blocks are not possible in interfaces.

EX:

```
interface I{
    int x = 10;
    void m1();
    void m2();
    void m3();
}

class A implements I{
    public void m1(){
        System.out.println("m1-A");
    }
    public void m2(){
        System.out.println("m2-A");
    }
    public void m3(){
        System.out.println("m3-A");
    }
    public void m4(){
        System.out.println("m4-A");
    }
}

public class Main {
    public static void main(String[] args) {
        I i = new A();
        i.m1();
        i.m2();
        i.m3();
        //i.m4(); ---> Error

        A a = new A();
        a.m1();
        a.m2();
        a.m3();
        a.m4();

        System.out.println(I.x);
        System.out.println(i.x);
    }
}
```

```
        System.out.println(A.x);
        System.out.println(a.x);
    }
}
```

Q) If we declare an interface with some abstract methods and if we implement some of the abstract methods [not for all the abstract methods] in the implementation class then what will be the result in Java application?

Ans:

If we declare an interface with some abstract methods then it is mandatory to implement all these abstract methods in the respective implementation class , if we miss any abstract method implementation in the implementation class then compiler will raise an error, in this context to remove compilation error either we have to implement all abstract methods or we have to declare the implementation class as an abstract class and we must provide implementation for the remaining abstract methods by taking a subclass for the implementation class.

EX:

```
interface I{
    void m1();
    void m2();
    void m3();
}
abstract class A implements I{
    public void m1(){
        System.out.println("m1-A");
    }
}
```

```

        }
    }

class B extends A{
    public void m2(){
        System.out.println("m2-B");
    }
    public void m3(){
        System.out.println("m3-B");
    }
}

public class Main {
    public static void main(String[] args) {
        I i = new B();
        i.m1();
        i.m2();
        i.m3();

        A a = new B();
        a.m1();
        a.m2();
        a.m3();

        B b = new B();
        b.m1();
        b.m2();
        b.m3();
    }
}

```

Q) Is it possible to implement more than one interface in a single implementation class?

Ans:

Yes, it is possible to implement more than one interface in a single implementation class, but we must provide implementation for all the abstract methods of all the interfaces.

EX:

```
interface I1{
    void m1();
}

interface I2{
    void m2();
}

interface I3{
    void m3();
}

class A implements I1, I2, I3{
    public void m1(){
        System.out.println("m1-A");
    }

    public void m2(){
        System.out.println("m2-A");
    }

    public void m3(){
        System.out.println("m3-A");
    }
}

public class Main {
    public static void main(String[] args) {
        I1 i1 = new A();
        i1.m1();
        I2 i2 = new A();
        i2.m2();
        I3 i3 = new A();
        i3.m3();

        A a = new A();
    }
}
```

```
a.m1();  
a.m2();  
a.m3();  
}  
}
```

Q) Is it possible to extend more than one interface to a single interface?

Ans:

Yes, it is possible to extend more than one interface to a single interface.

EX:

```
interface I1{  
    void m1();  
}  
interface I2{  
    void m2();  
}  
interface I3 extends I1, I2{  
    void m3();  
}  
class A implements I3{  
    public void m1(){  
        System.out.println("m1-A");  
    }  
    public void m2(){  
        System.out.println("m2-A");  
    }  
    public void m3(){  
        System.out.println("m3-A");  
    }  
}
```

```

}

public class Main {
    public static void main(String[] args) {
        I1 i1 = new A();
        i1.m1();
        I2 i2 = new A();
        i2.m2();
        I3 i3 = new A();
        i3.m1();
        i3.m2();
        i3.m3();

        A a = new A();
        a.m1();
        a.m2();
        a.m3();
    }
}

```

In general, in Java applications, we will use interfaces only for declaring services [abstract methods] but these services are implemented either by other module members or by third party organizations.

EX:

```

interface DBDriver{ // SUN Microsystems
    public void registerDriver();
    public void connect();
}

class OracleDBDriver implements DBDriver{ // Oracle

    @Override
    public void registerDriver() {
        System.out.println("OracleDBDriver is registered
with Java application");
    }
}

```

```
}

@Override
public void connect() {
    System.out.println("Connection established between
Java application and Oracle Database");
}
}

class MySQLDBDriver implements DBDriver{// MySQL

@Override
public void registerDriver() {
    System.out.println("MySQLDBDriver is registered
with Java application");
}

@Override
public void connect() {
    System.out.println("Connection established between
Java application and MySQL Database");
}
}

class SqlServerDBDriver implements DBDriver{// SqlServer

@Override
public void registerDriver() {
    System.out.println("SqlServerDBDriver is
registered with Java application");
}

@Override
public void connect() {
    System.out.println("Connection established between
Java application and SqlServer Database");
}
}
```

```

public class Main { // JDBC Application
    public static void main(String[] args) {
        DBDriver oracleDriver = new OracleDBDriver();
        oracleDriver.registerDriver();
        oracleDriver.connect();
        System.out.println();

        DBDriver mysqlDriver = new MySQLDBDriver();
        mysqlDriver.registerDriver();
        mysqlDriver.connect();
        System.out.println();

        DBDriver sqlserverDriver = new
        SqlServerDBDriver();
        sqlserverDriver.registerDriver();
        sqlserverDriver.connect();
    }
}

```

IN MVC based applications , we will use the following layers.

1. Presentation Layer
2. Controller Layer
3. Service Layer
4. DAO Layer

Presentation Layer:

1. It is a User interface layer, through this layer users can interact with applications.
2. In enterprise applications, we will use Html, Java script, CSS, Bootstrap, Angular, React JS,....
3. In standalone applications, we will use command prompt or console as Presentation layer.

Controller Layer:

It is a "Front Controller", it will listen to the requests from the Presentation layer and it will access Service layer methods.

It is a normal Java class[In the case of web applications , it is a Servlet]

In the Standalone applications , we will use the Main class and main() method as controllers.

Service Layer:

In general, service layer will provide middleware services to the applications like Security, Data Validations, Transaction Service,.....

In general, the Service layer will listen to the Controller layer and access Dao layer methods.

To prepare the Service layer we have to use the following steps.

1. Declare the services by taking an interface.
2. Provide implementation class for the service interface and provide implementations for all the service methods, where access Dao methods.

DAO Layer:

In general, the DAO layer is responsible for interacting with the Storage Areas like Databases.

It will listen to the Service Layer and perform the database operations.

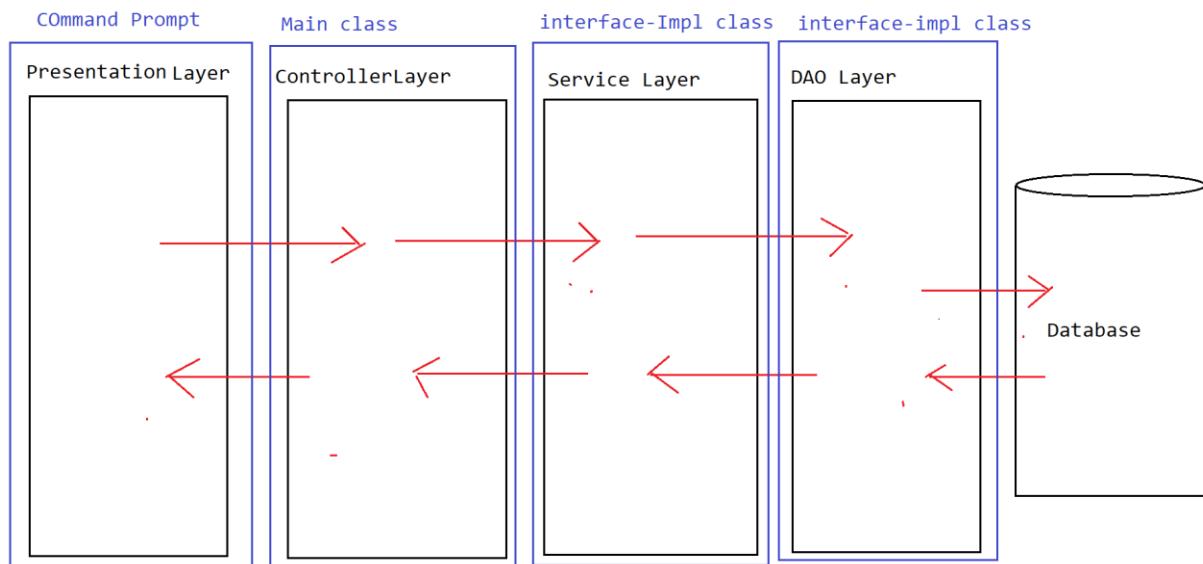
To prepare the Dao layer we have to use the following steps.

1. Declare DAO interface with all the DAO methods.

2. Provide an implementation class to the DAO interface and provide implementation for all the DAO methods.

Note: In an enterprise application, DAO classes, Services classes must be singleton classes and they must be provided through Factory classes and Factory Methods.

Note: To transfer Data from one layer to another layer we have to use Java bean class, here the data bean class is called "DTO" [Data Transfer Object].



Employee.java

```
package com.durgasoft.dto;

public class Employee {
    private int eno;
    private String ename;
    private float esal;
    private String eaddr;

    public int getEno() {
        return eno;
    }
}
```

```
public void setEno(int eno) {
    this.eno = eno;
}

public String getEname() {
    return ename;
}

public void setEname(String ename) {
    this.ename = ename;
}

public float getEsal() {
    return esal;
}

public void setEsal(float esal) {
    this.esal = esal;
}

public String getEaddr() {
    return eaddr;
}

public void setEaddr(String eaddr) {
    this.eaddr = eaddr;
}

}

EmployeeDao.java
package com.durgasoft.dao;

import com.durgasoft.dto.Employee;

public interface EmployeeDao {
    public String add(Employee employee);
```

```
    public Employee search(int eno);
    public String update(Employee employee);
    public String delete(int eno);
}
```

EmployeeDaoImpl.java

```
package com.durgasoft.dao;

import com.durgasoft.dto.Employee;

public class EmployeeDaoImpl implements EmployeeDao {

    @Override
    public String add(Employee employee) {
        return "Success";
    }

    @Override
    public Employee search(int eno) {
        Employee employee = new Employee();
        employee.setEno(111);
        employee.setEname("AAA");
        employee.setEsal(5000);
        employee.setEaddr("Hyd");
        return null;
    }

    @Override
    public String update(Employee employee) {

        return "success";
    }

    @Override
    public String delete(int eno) {
        return "success";
    }
}
```

```
    }  
}
```

EmployeeService.java

```
package com.durgasoft.service;  
  
import com.durgasoft.dto.Employee;  
  
public interface EmployeeService {  
    public String addEmployee(Employee employee);  
    public Employee searchEmployee(int eno);  
    public String updateEmployee(Employee employee);  
    public String deleteEmployee(int eno);  
}
```

EmployeeServiceImpl.java

```
package com.durgasoft.service;  
  
import com.durgasoft.dao.EmployeeDao;  
import com.durgasoft.dto.Employee;  
import com.durgasoft.factories.EmployeeDaoFactory;  
import com.durgasoft.factories.EmployeeServiceFactory;  
  
public class EmployeeServiceImpl implements  
EmployeeService{  
    @Override  
    public String addEmployee(Employee employee) {  
        EmployeeDao employeeDao =  
EmployeeDaoFactory.getEmployeeDao();  
        String status = employeeDao.add(employee);  
        return status;  
    }  
  
    @Override  
    public Employee searchEmployee(int eno) {
```

```

        EmployeeDao employeeDao =
EmployeeDaoFactory.getEmployeeDao();
        Employee employee = employeeDao.search(eno);
        return employee;
    }

@Override
public String updateEmployee(Employee employee) {
    EmployeeDao employeeDao =
EmployeeDaoFactory.getEmployeeDao();
    String status = employeeDao.update(employee);
    return status;
}

@Override
public String deleteEmployee(int eno) {
    EmployeeDao employeeDao =
EmployeeDaoFactory.getEmployeeDao();
    String status = employeeDao.delete(eno);
    return status;
}
}

```

```

EmployeeServiceFactory.java
package com.durgasoft.factories;

import com.durgasoft.service.EmployeeService;
import com.durgasoft.service.EmployeeServiceImpl;

public class EmployeeServiceFactrory {
    private static EmployeeService employeeService;
    static {
        employeeService = new EmployeeServiceImpl();
    }

    public static EmployeeService getEmployeeService() {

```

```
        return employeeService;
    }
}

EmployeeDaoFactory.java
```

```
package com.durgasoft.factories;

import com.durgasoft.dao.EmployeeDao;
import com.durgasoft.dao.EmployeeDaoImpl;

public class EmployeeDaoFactory {
    private static EmployeeDao employeeDao;
    static{
        employeeDao = new EmployeeDaoImpl();
    }

    public static EmployeeDao getEmployeeDao() {
        return employeeDao;
    }
}
```

```
Main.java
```

```
import com.durgasoft.dto.Employee;
import com.durgasoft.factories.EmployeeServiceFactory;
import com.durgasoft.service.EmployeeService;
import com.durgasoft.service.EmployeeServiceImpl;

public class Main {
    public static void main(String[] args) {
        EmployeeService employeeService =
EmployeeServiceFactory.getEmployeeService();
        /*
        Employee employee = new Employee();
        employee.setEno(111);
        employee.setEname("AAA");
        employee.setEsal(5000.0f);
```

```

employee.setAddr ("Hyd") ;

String status =
employeeService.addEmployee (employee) ;
System.out.println (status) ;
*/
/*
Employee emp =
employeeService.searchEmployee (222) ;
if (emp == null) {
    System.out.println ("Employee Does not Exist") ;
} else {
    System.out.println ("Employee Details") ;
    System.out.println ("-----") ;
    System.out.println ("Employee Number : " +
+emp.getEno ()) ;
    System.out.println ("Employee Name : " +
+emp.getEname ()) ;
    System.out.println ("Employee Salary : " +
+emp.getEsal ()) ;
    System.out.println ("Employee Address : " +
+emp.getEaddr ()) ;

}
*/
/*
Employee emp = new Employee () ;
emp.setEno (111) ;
emp.setEname ("XXX") ;
emp.setEsal (10000) ;
emp.setAddr ("Chennai") ;
String status =
employeeService.updateEmployee (emp) ;
System.out.println (status) ;
*/

```

```
        String status =  
employeeService.deleteEmployee(111);  
        System.out.println(status);  
    }  
}
```

Marker Interfaces:

Marker interface is a java interface, it does not have any abstract method , but they are having special capabilities and they will provide these capabilities to the objects at runtime when we implement these marker interfaces.

EX: `java.io.Serializable`

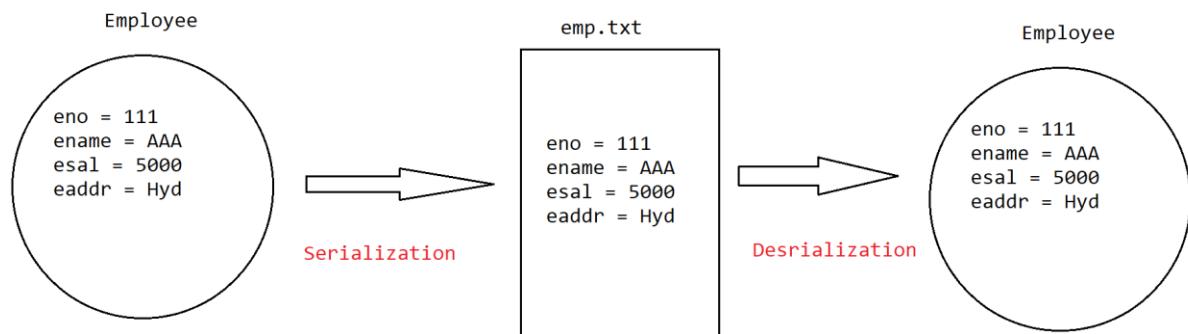
EX: `java.lang.Cloneable`

`java.io.Serializable`:

The process of separating data from an object is called **Serialization**.

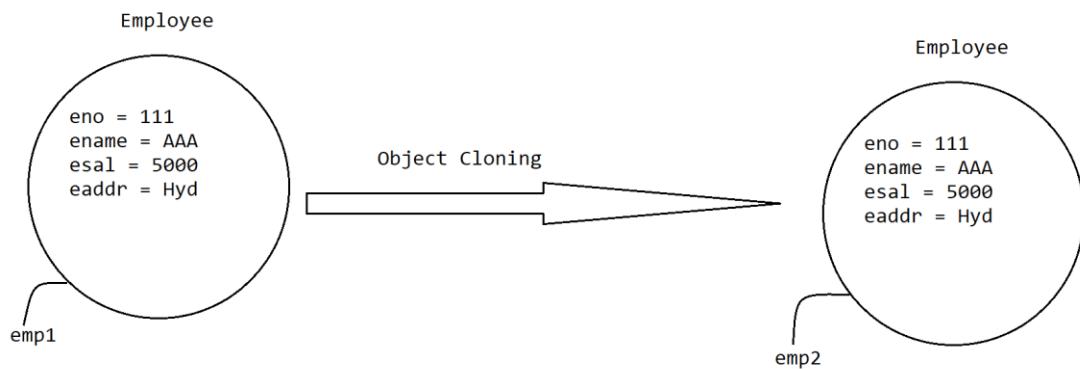
The process of reconstructing an object on the basis of the data is called **Deserialization**.

In java applications, by default all java objects are not eligible for **Serialization** and **Deserialization**, but only the objects which are implementing `java.io.Serializable` marker interface are eligible for **Serialization** and **Deserialization**.



`java.lang.Cloneable :`

The process of generating Duplicate object for an object is called Object Cloning.



In general, in java applications, by default all java objects are not eligible for Object cloning, only the

objects whose classes are implementing `java.lang.Cloneable` marker interface are eligible for Object cloning.

Adapter Classes:

In general, in java applications we may use number of interfaces and number of implementation classes.

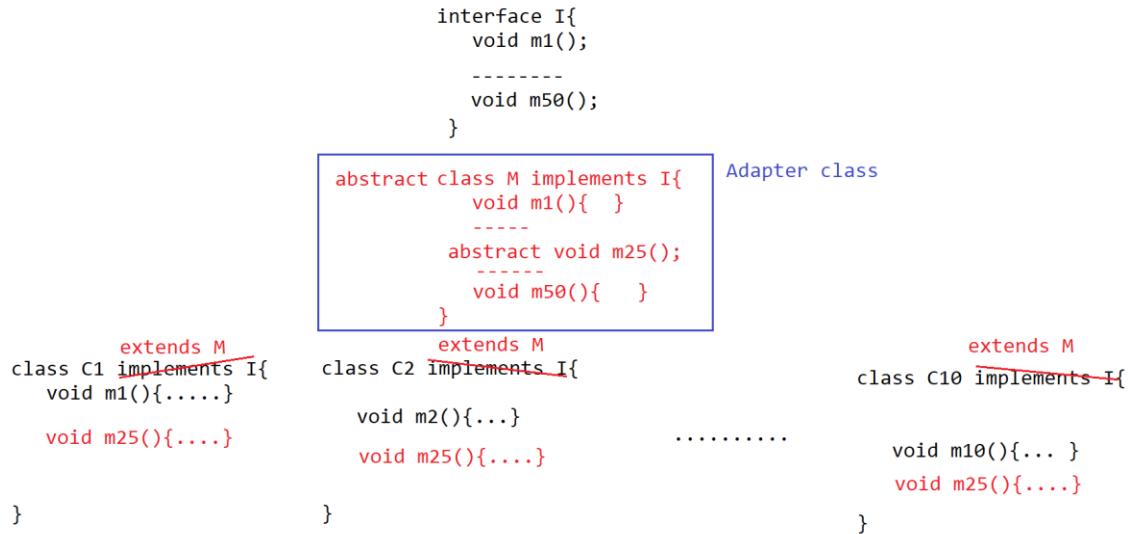
In Java applications, if we implement an interface in an implementation class then it is mandatory to implement all the methods of the respective interface irrespective of the actual requirement, this approach may increase unnecessary methods implementations in java applications.

To overcome the above design problem , we have to use a design pattern called "Adapter Design Pattern".

For the above design problem, Adapter design pattern has provided a solution in the form of the following steps.

1. Declare a mediator class between interface and implementation classes.
2. Implement the interface in the mediator class and provide empty implementation for all the methods.
3. At each and every implementation class , remove "implements" keyword and extend the mediator class to the implementation class.
4. Override the requirementmethods instead of implementing all methods.
5. In the above solution, Developers must create objects for the implementation classes, not for the mediator class, so it is suggestible to declare mediator class as an abstract class.
6. In the above solution, if we need any method common to all the implementation classes with variable implementations then it is suggestible to declare

that method as an abstract method in the mediator class.



EX1:

In GUI applications, to close the frame we have to use `windowClosing()` method from `WindowListener` interface which has 7 methods, due to `windowClosing()` method we must implement `WindowListener` interface in a class, in this situation, we must provide implementation for all the remaining 6 methods unnecessarily, it will increase unnecessary methods implementation in java applications, here to avoid unnecessary methods implementation JAVA has provided an adapter class in the form of "WindowAdapter".

```

public interface WindowListener{
    public void windowOpened(WindowEvent we);
    public void windowClosing(WindowEvent we);
    public void windowClosed(WindowEvent we);
    public void windowIconified(WindowEvent we);
    public void windowDeiconified(WindowEvent we);
    public void windowActivated(WindowEvent we);
    public void windowDeactivated(WindowEvent we);
}

^
public abstract class WindowAdapter implements WindowListener{
    public void widnowOpened(WindowEvent we){    }
    public void widnowClosing(WindowEvent we){    }
    public void widnowClosed(WindowEvent we){    }
    public void widnowIconified(WindowEvent we){    }
    public void widnowDeiconified(WindowEvent we){    }
    public void widnowActivated(WindowEvent we){    }
    public void widnowDeactivated(WindowEvent we){    }
}

^
public class WindowHandler extends WindowAdapter{
    public void windowClosing(WindowEvent we){ .... }
}

```

EX2:

In web applications, we are able to prepare Servlet classes by implementing Servlet interface, where Servlet interface has 5 methods, where in five methods we need only one method that is `service()` method.

Because of `service()` method if we implement Servlet interface in any class then we must provide the implementation for the remaining 4 methods unnecessarily, this approach will increase unnecessary methods in web applications, to overcome this design problem, Servlet API has provided an adapter class in the form of "GenericServlet".

```

public interface Servlet{
    public void init(ServletConfig config);
    public void service(ServletRequest req, ServletResponse resp);
    public ServletConfig getServletConfig();
    public String getServletInfo();
    public void destroy();
}

public abstract class GenericServlet implements Servlet{
    public void init(ServletConfig config){ }
    public abstract void service(ServletRequest req, ServletResponse res);
    public ServletConfig getServletConfig(){ }
    public String getServletInfo(){ }
    public void destroy(){ }
}

public class MyServlet extends GenericServlet{
    public void service(ServletRequest req, ServletResponse res){
        -----
    }
}

```

Object Cloning:

The process of generating a duplicate object from an object is called Object cloning.

In Java applications, by default all objects are not eligible for object cloning, only the objects whose classes are implementing `java.lang.Cloneable` marker interface are eligible for Object cloning.

To perform Object cloning in java applications we have to use the following steps.

1. Declare an user defined class .
2. Implement `java.lang.Cloneable` marker interface.
3. Override `Object` class provided `clone()` method in user defined class, where access superclass `clone()` by using "super" keyword.
4. In the main class, in the `main()` method , create an object[Original] for the User defined class.
5. Access `clone()` method to get Duplicate objects.

EX:

```
class Student implements Cloneable{
    private String sid;
    private String sname;
    private String saddr;

    public Student(String sid, String sname, String saddr)
    {
        this.sid = sid;
        this.sname = sname;
        this.saddr = saddr;
    }

    @Override
    public Object clone() throws
CloneNotSupportedException {
    Object obj = super.clone();
    return obj;
}

    public void getStudentDetails(){
        System.out.println("Student Details");
        System.out.println("-----");
        System.out.println("Student Id      : "+sid);
        System.out.println("Student Name    : "+sname);
        System.out.println("Student Address : "+saddr);
    }
}

public class Main {
    public static void main(String[] args) throws
CloneNotSupportedException {
        Student student1 = new Student("S-111", "Durga",
"Hyd");
        System.out.println("Student Details From Original
Object");
        student1.getStudentDetails();
```

```
        System.out.println("Original Student Object Ref :  
"+student1);  
        System.out.println();  
  
        Student student2 = (Student) student1.clone();  
        System.out.println("Student Details from Duplicate  
Object");  
        student2.getStudentDetails();  
        System.out.println("Duplicate Student Object ref :  
"+student2);  
    }  
}
```

OP:

```
Student Details From Original Object  
Student Details  
-----  
Student Id      : S-111  
Student Name    : Durga  
Student Address : Hyd  
Original Student Object Ref : Student@119d7047
```

```
Student Details from Duplicate Object  
Student Details  
-----  
Student Id      : S-111  
Student Name    : Durga  
Student Address : Hyd  
Duplicate Student Object ref : Student@776ec8df
```

There are two types of Object cloning in Java

1. Shallow Cloning
2. Deep Cloning

Q) What are the differences between Shallow Cloning and Deep Cloning?

Ans:

1. Shallow cloning is a default cloning mechanism which was used by JAVA internally, no need to define cloning logic explicitly.

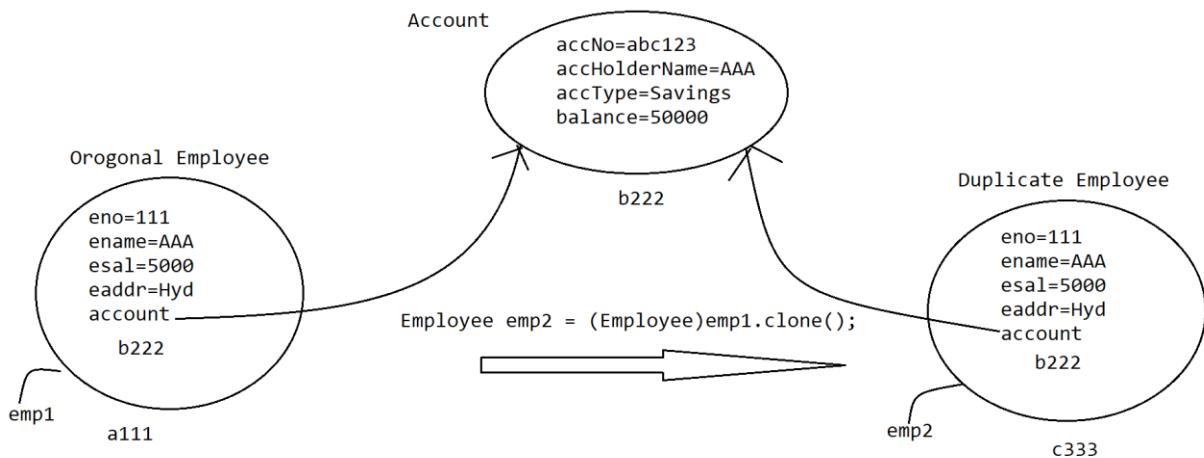
Note: Object class has `clone()` method, it was implemented as per shallow cloning.

Deep cloning is not the default cloning mechanism, where we must define cloning logic explicitly.

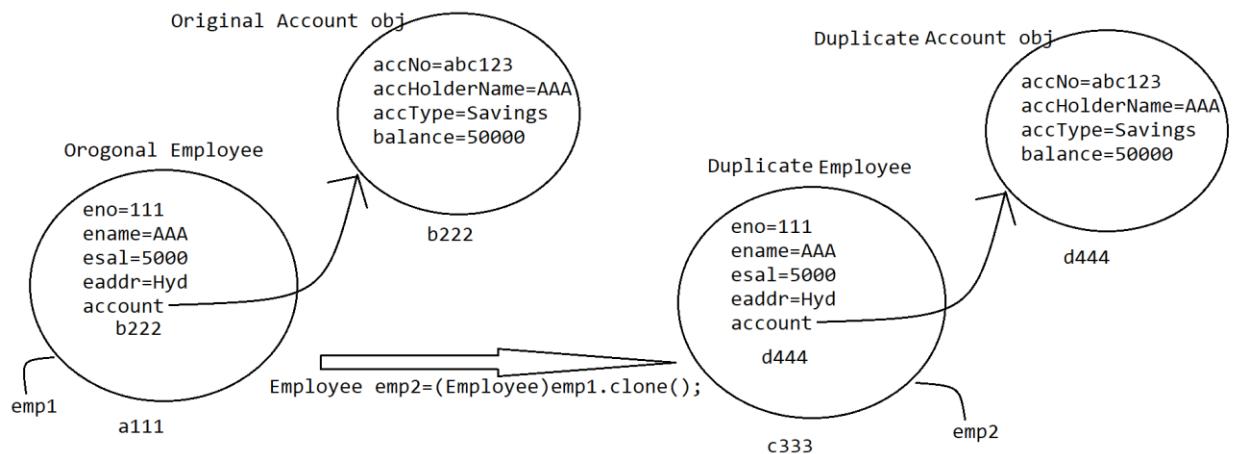
2. In the case of Shallow cloning, if we clone a container object then JVM will perform cloning over container object only, JVM will not perform cloning over contained object along with Container object cloning.

In the case of Deep cloning, if we perform clone operation over a container object then JVM will perform Cloning over Contained object automatically along with Container object cloning.

3. Shallow Cloning.



Deep Cloning



EX For Shallow Cloning:

```
Account.java
package com.durgasoft.entities;

public class Account {
```

```
private String accNo;
private String accHolderName;
private String accType;
private int balance;

public String getAccNo() {
    return accNo;
}

public void setAccNo(String accNo) {
    this.accNo = accNo;
}

public String getAccHolderName() {
    return accHolderName;
}

public void setAccHolderName(String accHolderName)
{
    this.accHolderName = accHolderName;
}

public String getAccType() {
    return accType;
}

public void setAccType(String accType) {
    this.accType = accType;
}

public int getBalance() {
    return balance;
}

public void setBalance(int balance) {
    this.balance = balance;
}
```

```
        }
    }

Employee.java
package com.durgasoft.entities;

public class Employee implements Cloneable {
    private int eno;
    private String ename;
    private float esal;
    private String eaddr;
    private Account account;

    @Override
    public Object clone() throws
CloneNotSupportedException {
        Account duplicateAccount = new Account();
        duplicateAccount.setAccNo(account.getAccNo());

        duplicateAccount.setAccHolderName(account.getAccHolde
rName());
        duplicateAccount.setAccType(account.getAccType());
        duplicateAccount.setBalance(account.getBalance());

        Employee duplicateEmployee = new
Employee(this.eno, this.ename, this.esal,
this.eaddr,duplicateAccount);
        return duplicateEmployee;
    }

    public Employee(int eno, String ename, float esal,
String eaddr, Account account) {
        this.eno = eno;
        this.ename = ename;
```

```

        this.esal = esal;
        this.eaddr = eaddr;
        this.account = account;
    }

    public Account getAccount() {
        return account;
    }

    public void getEmployeeDetails() {
        System.out.println("Employee Details");
        System.out.println("-----");
        System.out.println("Employee Number : " + eno);
        System.out.println("Employee Name : " + ename);
        System.out.println("Employee Salary : " + esal);
        System.out.println("Employee Address : " + eaddr);
        System.out.println("Account Details");
        System.out.println("-----");
        System.out.println("Account Number : " + account.getAccNo());
        System.out.println("Account Holder Name : " + account.getAccHolderName());
        System.out.println("Account Type : " + account.getAccType());
        System.out.println("Account Balance : " + account.getBalance());
    }
}

```

Main.java

```
-----  
import com.durgasoft.entities.Account;  
import com.durgasoft.entities.Employee;  
  
public class Main {  
    public static void main(String[] args) throws  
CloneNotSupportedException {  
        Account account = new Account();  
        account.setAccNo("abc123");  
        account.setAccHolderName("Durga");  
        account.setAccType("Savings");  
        account.setBalance(50000);  
  
        Employee employee1 = new Employee(111,  
"Durga", 5000, "Hyd", account);  
        System.out.println("Original Employee  
Details");  
        employee1.getEmployeeDetails();  
        System.out.println("Original Employee Object  
Ref : "+employee1);  
        System.out.println("Original Account Object  
Ref : "+employee1.getAccount());  
        System.out.println();  
  
        Employee employee2 = (Employee)  
employee1.clone();  
        System.out.println("Duplicate Employee  
Details");  
        employee2.getEmployeeDetails();  
        System.out.println("Duplicate Employee Object  
Ref : "+employee2);  
        System.out.println("Duplicate Account Object  
Ref : "+employee2.getAccount());  
  
    }  
}
```

Output:

Original Employee Details

Employee Details

Employee Number : 111
Employee Name : Durga
Employee Salary : 5000.0
Employee Address : Hyd
Account Details

Account Number : abc123
Account Holder Name : Durga
Account Type : Savings
Account Balance : 50000
Original Employee Object Ref :
com.durgasoft.entities.Employee@7ef20235
Original Account Object Ref :
com.durgasoft.entities.Account@27d6c5e0

Duplicate Employee Details

Employee Details

Employee Number : 111
Employee Name : Durga
Employee Salary : 5000.0
Employee Address : Hyd
Account Details

Account Number : abc123
Account Holder Name : Durga
Account Type : Savings
Account Balance : 50000
Duplicate Employee Object Ref :
com.durgasoft.entities.Employee@4f3f5b24

```
Duplicate Account Object Ref      :  
com.durgasoft.entities.Account@27d6c5e0
```

EX for Deep Cloning:

```
-----  
Account.java  
package com.durgasoft.entities;  
  
public class Account {  
    private String accNo;  
    private String accHolderName;  
    private String accType;  
    private int balance;  
  
    public String getAccNo() {  
        return accNo;  
    }  
  
    public void setAccNo(String accNo) {  
        this.accNo = accNo;  
    }  
  
    public String getAccHolderName() {  
        return accHolderName;  
    }  
  
    public void setAccHolderName(String accHolderName)  
    {  
        this.accHolderName = accHolderName;  
    }  
  
    public String getAccType() {  
        return accType;  
    }
```

```
public void setAccType(String accType) {
    this.accType = accType;
}

public int getBalance() {
    return balance;
}

public void setBalance(int balance) {
    this.balance = balance;
}
}

Employee.java
package com.durgasoft.entities;

public class Employee implements Cloneable {
    private int eno;
    private String ename;
    private float esal;
    private String eaddr;
    private Account account;

    @Override
    public Object clone() throws
CloneNotSupportedException {
        Account duplicateAccount = new Account();
        duplicateAccount.setAccNo(account.getAccNo());

        duplicateAccount.setAccHolderName(account.getAccHolde
rName());

        duplicateAccount.setAccType(account.getAccType());
        duplicateAccount.setBalance(account.getBalance());
    }
}
```

```

        Employee duplicateEmployee = new
Employee(this.eno, this.ename, this.esal,
this.eaddr,duplicateAccount);
        return duplicateEmployee;
    }

    public Employee(int eno, String ename, float esal,
String eaddr, Account account) {
        this.eno = eno;
        this.ename = ename;
        this.esal = esal;
        this.eaddr = eaddr;
        this.account = account;
    }

    public Account getAccount() {
        return account;
    }

    public void getEmployeeDetails() {
        System.out.println("Employee Details");
        System.out.println("-----");
        System.out.println("Employee Number      : "+eno);
        System.out.println("Employee Name       : "+ename);
        System.out.println("Employee Salary     : "+esal);
        System.out.println("Employee Address    : "+eaddr);
        System.out.println("Account Details");
        System.out.println("-----");
    }
}

```

```

        System.out.println("Account Number      : "
"+account.getAccNo());
        System.out.println("Account Holder Name : "
"+account.getAccHolderName());
        System.out.println("Account Type       : "
"+account.getAccType());
        System.out.println("Account Balance    : "
"+account.getBalance());
    }
}

Main.java
import com.durgasoft.entities.Account;
import com.durgasoft.entities.Employee;

public class Main {
    public static void main(String[] args) throws
CloneNotSupportedException {
        Account account = new Account();
        account.setAccNo("abc123");
        account.setAccHolderName("Durga");
        account.setAccType("Savings");
        account.setBalance(50000);

        Employee employee1 = new Employee(111,
"Durga", 5000, "Hyd", account);
        System.out.println("Original Employee
Details");
        employee1.getEmployeeDetails();
        System.out.println("Original Employee Object
Ref   : "+employee1);
        System.out.println("Original Account Object
Ref   : "+employee1.getAccount());
        System.out.println();

```

```

        Employee employee2 = (Employee)
employee1.clone();
        System.out.println("Duplicate Employee
Details");
        employee2.getEmployeeDetails();
        System.out.println("Duplicate Employee Object
Ref : "+employee2);
        System.out.println("Duplicate Account Object
Ref : "+employee2.getAccount());
    }

}

```

Output

```

Original Employee Details
Employee Details
-----
Employee Number      : 111
Employee Name       : Durga
Employee Salary     : 5000.0
Employee Address    : Hyd
Account Details
-----
Account Number      : abc123
Account Holder Name : Durga
Account Type        : Savings
Account Balance     : 50000
Original Employee Object Ref   :
com.durgasoft.entities.Employee@7ef20235
Original Account Object Ref   :
com.durgasoft.entities.Account@27d6c5e0

```

Duplicate Employee Details

```

Employee Details
-----
Employee Number      : 111

```

```
Employee Name      : Durga
Employee Salary    : 5000.0
Employee Address   : Hyd
Account Details
-----
Account Number     : abc123
Account Holder Name: Durga
Account Type       : Savings
Account Balance    : 50000
Duplicate Employee Object Ref   :
com.durgasoft.entities.Employee@4f3f5b24
Duplicate Account Object Ref    :
com.durgasoft.entities.Account@15aeb7ab
```

instanceof operator:

```
-----
'instanceof' operator is a boolean operator, it can
be used to check whether a reference variable is
representing an instance of a particular entity or
not.
```

Syntax:

```
refVar instanceof ClassName
```

Case#1: If the ref Variable type is the same as the specified className then the instanceof operator will return true value.

Case#2: If the ref variable type is subclass to the specified className then instanceof operator will return true value.

Case#3: If the ref variable type is superclass to the specified className then instanceof operator will return false value.

Case#4: If the ref variable type and the specified className are not related then the compiler will raise an error.

Case#5: If the ref variable type is an interface and it is not related to the specified className then instanceof operator will return false value, but not any compilation error.

EX:

```
class A{  
  
}  
class B extends A{  
  
}  
class C{  
  
}  
public class Main {  
    public static void main(String[] args) {  
        A a = new A();  
        System.out.println(a instanceof A);  
        B b = new B();  
        System.out.println(b instanceof B);  
        System.out.println(b instanceof A);  
        System.out.println(a instanceof B);  
        C c = new C();  
        System.out.println(c instanceof C);  
        //System.out.println(c instanceof A); --->  
Error  
  
    }  
}
```

EX:

```
class A{  
  
}  
class B extends A{  
  
}  
  
public class Main {  
    public static void main(String[] args) {  
        A a = new B();  
        System.out.println(a instanceof A);  
        System.out.println(a instanceof B);  
    }  
}  
True  
True
```

EX:

```
interface I{  
  
}  
class A{  
  
}  
class B extends A{  
  
}  
class C implements I{  
  
}  
  
public class Main {  
    public static void main(String[] args) {  
        A a = new B();  
        System.out.println(a instanceof A);  
    }  
}
```

```
System.out.println(a instanceof B);
System.out.println(a instanceof Object);

I i = new C();
System.out.println(i instanceof C);
System.out.println(i instanceof A);
}

Op:
true
true
true
true
false
```