

Class Notes @ 8:30AM : <https://tinyurl.com/COREJAVA830>

Workshop Link:

<https://attendee.gotowebinar.com/register/3184395022566665054>

Webinar ID: 736-975-283

``

Advanced JAVA Demo LINK:

Online :

<https://attendee.gotowebinar.com/register/2413803399191980634>

Webinar ID : 329-959-499

Spring Framework Demo Link:

Online :

<https://attendee.gotowebinar.com/register/3828952929465334879>

Webinar ID: 284-643-971

Object Orientation:

To prepare applications, there are four 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. USPLs are outdated programming languages, they were provided at the starting point of the computers, they are not suitable for the at present application requirements.

EX: BASIC, FORTRAN,....

SPLs are not outdated programming languages, they are useful for our present application requirements.

EX: C, PASCAL,....

2. USPLs do not have any standard structure to prepare the applications.

SPLs are having a standard Structure to prepare the applications.

3. USPLs are using mnemonic codes, they are low level codes, they are available in very less number and they will provide less number of features to the applications.

SPLs are using high level syntaxes, they are available in more numbers and they will provide more features to the application.

4. USPLA are using only goto statements to define flow of execution in the applications.

SPL is using more flow controllers to define flow of execution.

5. USPLs are not having functions, so code reusability is not possible.

SPLs have functions, so code reusability is possible.

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 prepare the applications.

Object oriented Programming languages are providing a simple approach to prepare the applications.

2. Structured Programming languages are providing less modularization.

Object oriented programming languages are providing a very good Modularization.

3. Structured Programming languages are providing less Abstraction.

Object oriented programming languages are providing a very good Abstraction.

4. Structured programming languages are providing less security.

Object oriented programming languages provide very good Security.

5. Structured Programming languages are providing less Shareability.

Object Oriented programming languages are providing more shareability.

6. Structured programming languages are providing less Reusability.

Object Oriented programming languages are providing more Reusability.

Aspect oriented Programming languages:

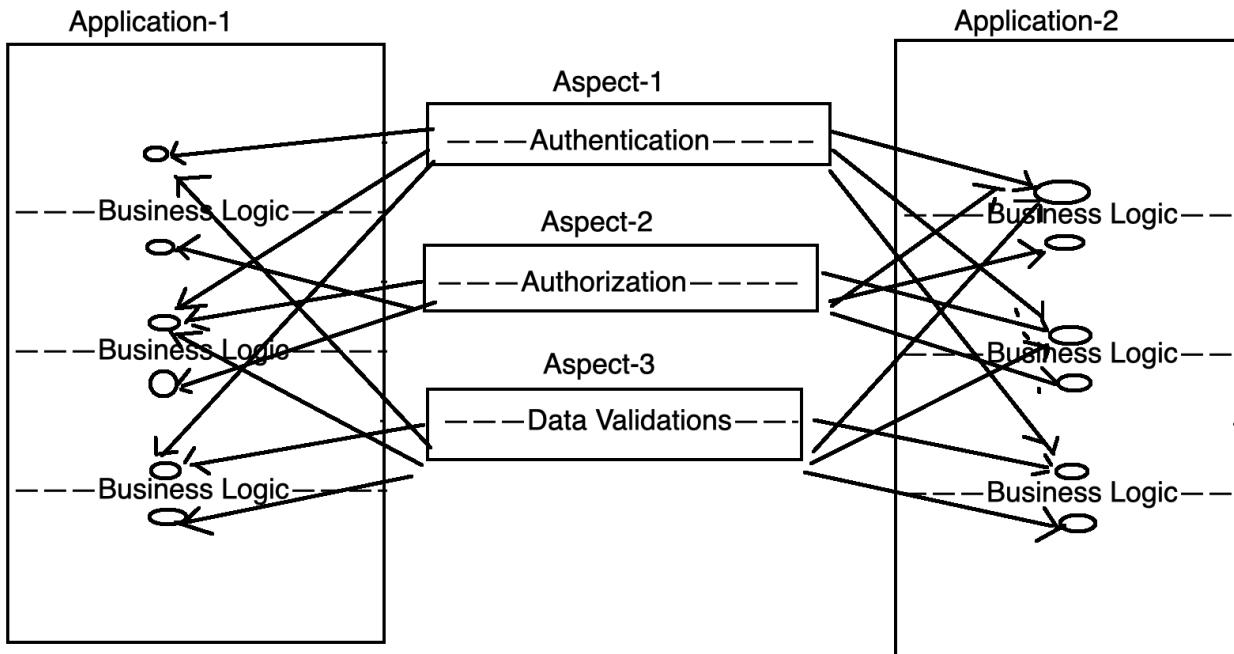
“Aspect Oriented programming languages” is a wrong terminology, because no Programming language exists on the basis of the Aspect orientation.

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

In general,in Object Orientation programming we will prepare the programs by combining both Services logics and Business logics, it will provide less shareability and less reusability.

IN the above Object Oriented programming, to improve shareability and Reusability we have to use the following Aspect orientation provided conventions.

1. Separate all the services logics from the Business Logic.
2. Decare each and every Service as an Aspect.
3. Find the Join points in the application to inject aspects.
4. Inject the aspects at runtime of the applications.



Object Oriented Features:

To describe the nature of the Object orientation, Object Orientation has provided the following 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: C++, Java, Python,....

Object Based Programming Languages are following almost all the Object Oriented Features excluding Inheritance.

EX: Java Script.

Q)What are the differences between Class and Object?

Ans:

1. Class is a group of elements having common properties and common behaviors.

Object is an individual element among the group of elements having real properties and behaviors.

2. Class is Virtual.
Object is real or Physical.
3. Class is a Generalization.

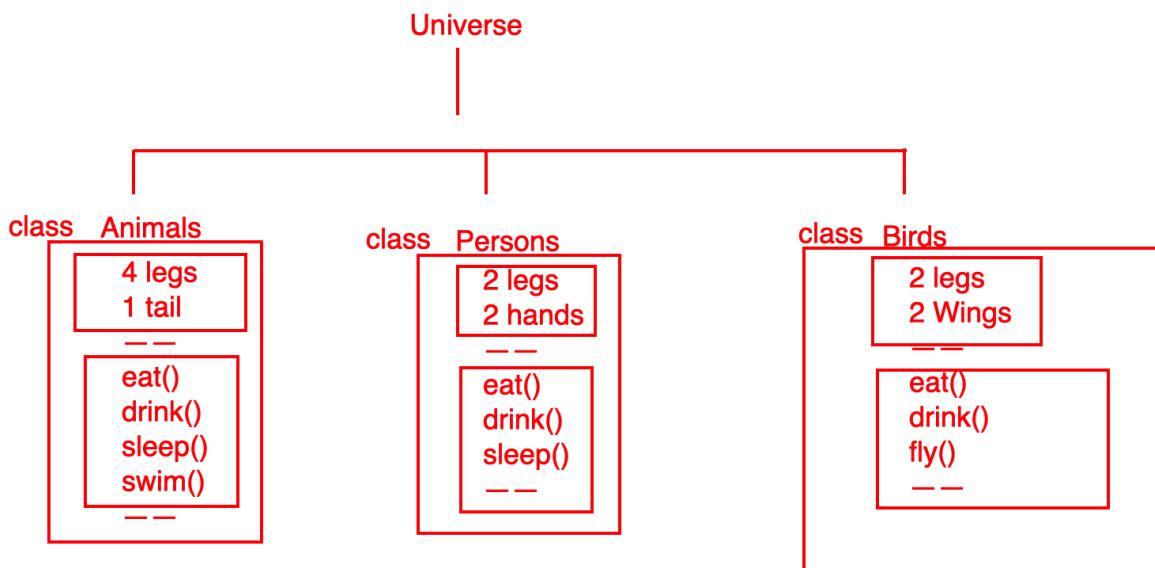
Object is a Specialization.

4. Class is the Virtual Encapsulation of the properties and behaviors.

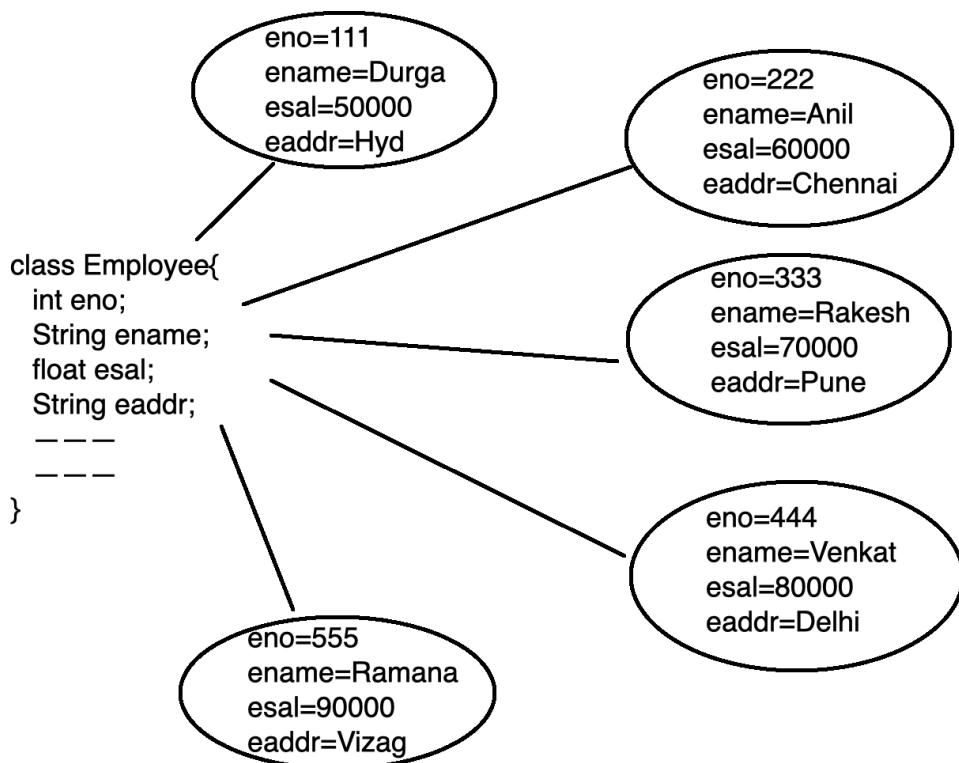
Object is the real or physical encapsulation of the Properties and Behaviors.

5. Class is a Model or blueprint for the Objects.

Object is an instance of the class.



In java applications, we are able to create more objects from a single class.



Q) What is the difference between Encapsulation and Abstraction?

Ans:

Encapsulation is the process of binding data and coding.

Abstraction is the process of hiding unnecessary implementations and showing necessary implementations.

In Java applications, both Encapsulation and Abstraction are Co-existed, both together will provide “Security” in the applications.

Encapsulation + Abstraction = Security

Inheritance:

Inheritance is a relation between classes, it will provide variables and methods from one class to the another class.

In Java applications, Inheritance is able to improve “Code Reusability”.

Polymorphism:

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

If one thing exists in more than one form then it is called Polymorphism.

The main advantage of the Polymorphism is “Flexibility” to design applications.

Message Passing:

The process of sending data along with flow of execution from one entity to the another entity is called Message Passing.

The main Advantage of the Message Passing is “Communication between entities” and the “Data Navigation between Entities”.

Containers in Java:

Container is a top element in java, it is able to manage some other programming elements like variables, methods, blocks, constructors,....

There are three types of containers in Java.

1. Class
2. Abstract Class
3. Interface

Class:

The main purpose of the class is to represent Entities in the Java programming language.

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

All the Entities attributes[Data] are represented in the form of Variables inside the classes.

EX: studentId, studentName, studentAddress, studentEmailId,... in Student class.

All the Entities behaviors[Activities] are represented in the form of methods inside the classes.

EX: deposit(), withdraw(), transferFunds(),... in Transaction class.

EX: createAccount(), updateAccount(), deleteAccount(),... in Account class.

IN Java Applications , if we want to prepare classes we have to use the following Syntax:

```
[Access Modifiers] class ClassName [extends  
SuperClassName][implements InterfaceList]{
```

-----Variables-----

-----Constructors-----

-----Methods-----

-----Blocks-----

-----Classes-----

-----Abstract Classes-----

-----Interfaces-----

-----Enums-----

}

Access Modifiers:

The main purpose of the Access modifiers is

1. To define scopes to the programming elements like variables, methods, classes,..... And
2. To define some extra nature to the programming elements like variables, methods, classes,....

To define scopes to the programming elements we will use the following access modifiers.

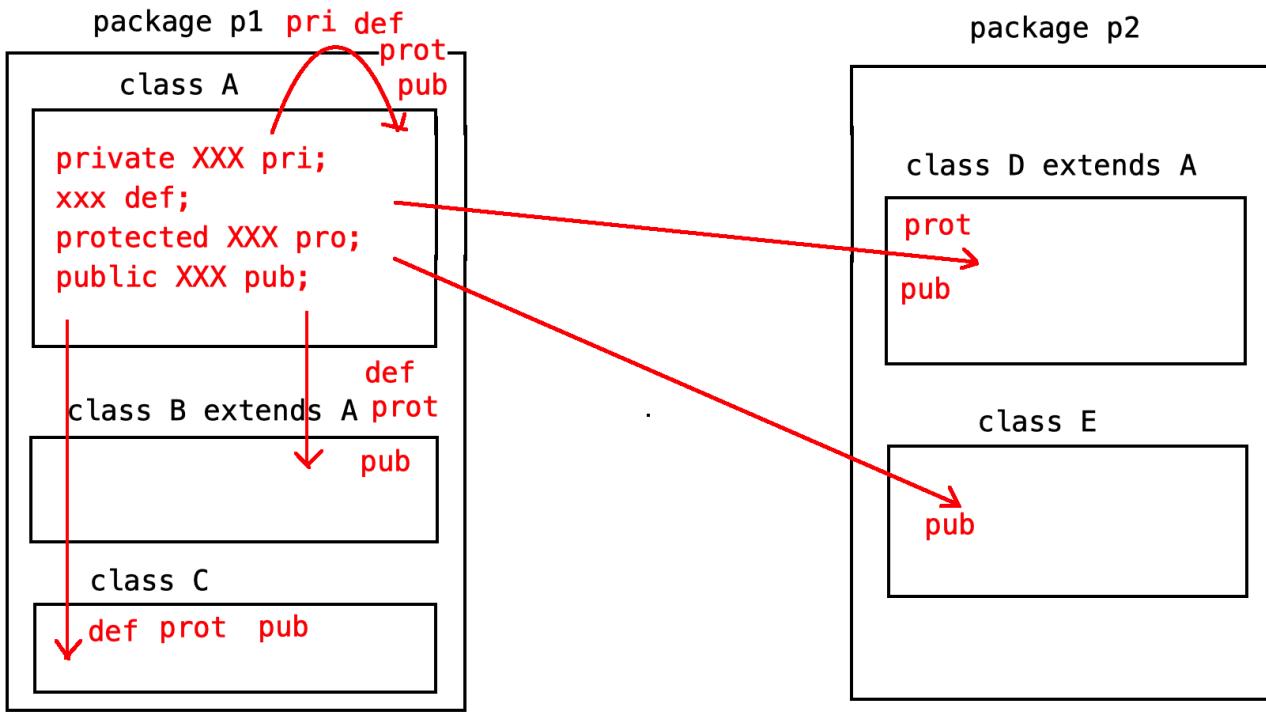
1. Private or Restricted
2. <default> or Package
3. protected
4. Public or Global

Private members have the scope up to the respective class where we declare that members.

<default> members have the scope throughout the present package.

Protected members have the scope throughout the present package and the subclass available in other packages.

Public members have the scope throughout the application.



In Java applications, classes are able to allow the access modifiers like public and <default>, not allowing the access modifiers like protected and private.

Note: Inner classes are able to allow the access modifiers like public, protected, <default> and private.

Note: If any access modifier is defined on the classes scopes 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 modifier “private” is defined on the boundaries of the class, so it is not applicable for the class and it is applicable for the members of the class.

To define some extra nature to the programming elements java has provided the following access modifiers.

static, final, abstract, native, volatile, transient, synchronized and strictfp.

In Java applications, classes are able to allow the access modifiers like final, abstract and strictfp.

Note: Inner classes are able to allow the access modifiers like static, final, abstract and strictfp.

Note:

```
class A{
    int i = 10;
    static int j = 20;
    static class B{
    }
}

A a = new A();
System.out.println(a.i);
System.out.println(A.i);----> Error
System.out.println(a.j);
System.out.println(A.j); ---> No Error
```

Where the “class” keyword is able to represent the Class Object oriented Feature.

Where the “ClassName” is an identifier to recognize and access the class independently.

Where the “extends” is able to represent the Inheritance object oriented feature, it is able to allow only one superclass name in the class syntax.

Where the “implements” keyword is able to provide one or more interfaces names in the class syntax in order to provide implementation for the methods of the interfaces.

In Class Syntax, both extends and implements keywords are optional, so

1. We can declare a class with extends keyword and without implements keyword.
2. We can declare a class without an extends keyword and with implements keyword.
3. We can declare a class without both extends and implements keywords.
4. We can declare a class with both extends and implements keywords, but we must provide extends keyword first then implements keyword.

Q) Find the valid syntaxes from the following list?

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

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

Steps to utilize classes in Java applications:

1. Declare a class by using the “class” keyword.
2. Inside the class declare variables and methods as per the application requirements.
3. In the Main class, in the main() method, create an object for the user defined class.
4. Access the members of the class by using the generated reference variable.

EX:

```
class Employee{
    int eno = 111;
    String ename = "Durga";
```

```
float esal = 50000.0f;
String eemailId = "durga123@durgasoft.com";
String emobileNo = "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 Email Id    : "+eemailId);
    System.out.println("Employee Mobile No   : "+emobileNo);
    System.out.println("Employee Address      : "+eaddr);

}

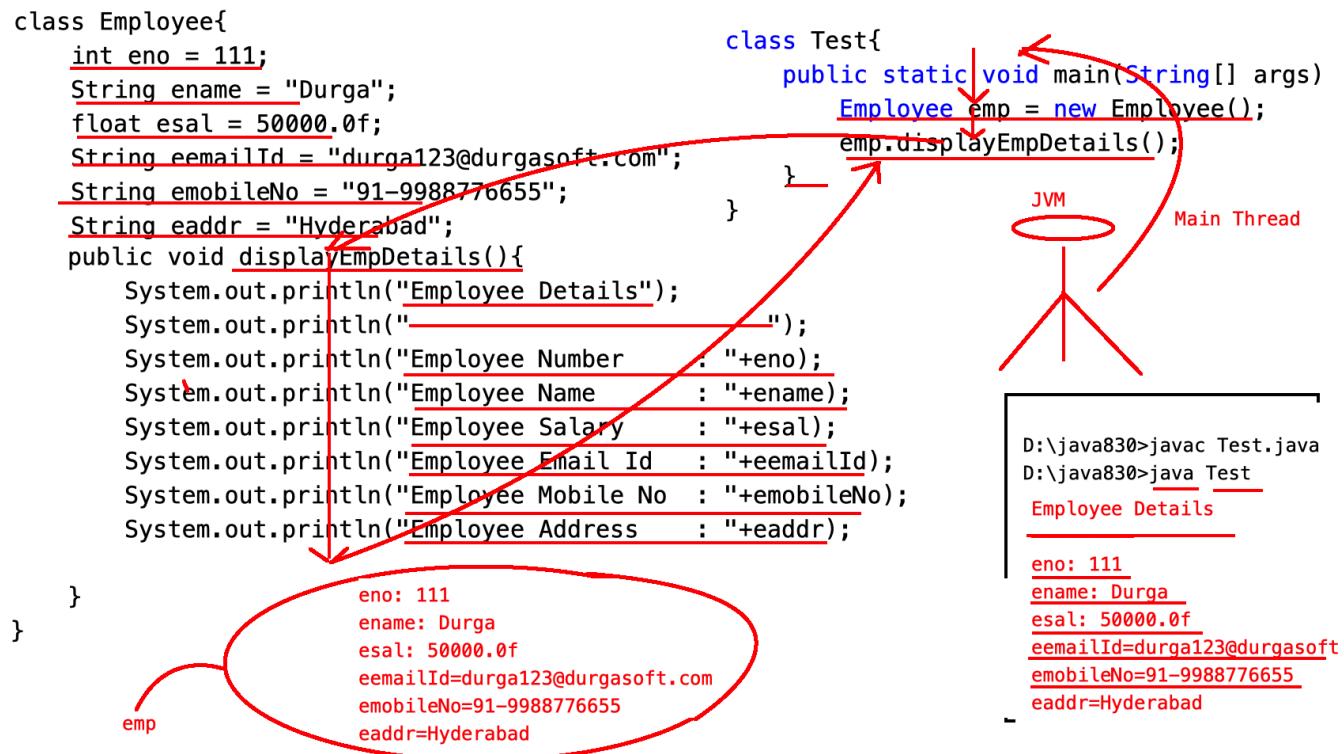
}

class Test{
    public static void main(String[] args) {
        Employee emp = new Employee();
        emp.displayEmpDetails();
    }
}

nagoorn@Nagoors-MacBook-Pro COREJAVA-7 % javac Test.java
```

```
nagoorn@Nagoors-MacBook-Pro COREJAVA-7 % java Test
Employee Details
```

```
-----
Employee Number      : 111
Employee Name        : Durga
Employee Salary       : 50000.0
Employee Email Id    : durga123@durgasoft.com
Employee Mobile No   : 91-9988776655
Employee Address     : Hyderabad
```



EX:

`Student.java`

```
public class Student {
```

```
String sid = "S-111";
String sname = "Durga";
String saddr = "Hyd";

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);
}

}
```

Customer.java

```
public class Customer {

    String cid = "C-111";
    String cname = "Anil";
    String caddr = "Chennai";

    public void getCustomerDetails(){
        System.out.println("Customer Details");
        System.out.println("-----");
        System.out.println("Customer Id      : "+cid);
        System.out.println("Customer Name     : "+cname);
        System.out.println("Customer Address  : "+caddr);
    }
}
```

```
 }
}
```

Test.java

```
public class Test {
    public static void main(String[] args) {
        Student student = new Student();
        student.getStudentDetails();
        System.out.println();

        Customer customer = new Customer();
        customer.getCustomerDetails();
    }
}
```

```
nagoorn@Nagoors-MacBook-Pro COREJAVA-830 % javac Test.java
```

```
nagoorn@Nagoors-MacBook-Pro COREJAVA-830 % java Test
```

Student Details

```
-----
Student Id      : S-111
Student Name    : Durga
Student Address : Hyd
```

Customer Details

```
-----
Customer Id     : C-111
Customer Name   : Anil
Customer Address: Chennai
```

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?

1. Concrete Method is a normal Java method, it must have both Method Declaration and Method Implementation.

Abstract Method is a Java method, it must have only Method Declaration without the Method implementation.

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

Abstract Methods are possible in the abstract classes and in the interfaces.

3. To declare concrete methods no special keyword is required.

To declare abstract methods we must use the “abstract” keyword.

4. Concrete methods will provide less shareability.

Abstract Methods will provide more shareability.

Concrete Method:

```
public void add(int l, int j)
{
    int result = l + j;
    System.out.println(result);
}
```

Method Declaration / Method Prototype / Method Header

Method Implementation / Method Body

Abstract Method:

```
public abstract void add(int l, int j);
```

Abstract Classes:

It is like a class, it is able to allow zero or more concrete methods and zero or more abstract methods.

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

In Java applications, Abstract classes allow us to prepare only reference variables, they are not allowed to create objects.

In Java applications, if we want to use abstract classes then we have to use the following steps.

1. Declare an abstract class by providing the “abstract” keyword along with the “class” keyword.
2. Declare variables and methods inside the abstract class as per the requirement.
3. Provide a subclass for the abstract class.
4. Provide implementations for the abstract methods of the Abstract class inside the subclass.
5. In the Main class, in the main() method, create an object for the subclass and declare a reference variable either for the abstract class or for the subclass.
6. Access the members of the abstract class.

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");
    }
}

public class Test {
    public static void main(String[] args) {
        //A a = new A(); ---> Error
        A a = new B();
        a.m1();
        a.m2();
        a.m3();

        B b = new B();
        b.m1();
    }
}
```

```
    b.m2();
    b.m3();
}

}
```

```
nagoorn@Nagoors-MacBook-Pro COREJAVA-830 % javac Test.java
nagoorn@Nagoors-MacBook-Pro COREJAVA-830 % java Test
```

```
m1-A
m2-B
m3-B
m1-A
m2-B
m3-B
```

Note: In the case of inheritance, if we declare a reference variable of superclass then we are able to access only superclass members, it is not possible to access subclass own members, but if we declare a reference variable of subclass then we are able to access both superclass members and the subclass own members.

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 Test {
    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();
    }
}
```

nagoorn@Nagoors-MacBook-Pro COREJAVA-830 % javac Test.java

```
nagoorn@Nagoors-MacBook-Pro COREJAVA-830 % java Test
```

```
m1-A
```

```
m2-B
```

```
m3-B
```

```
m1-A
```

```
m2-B
```

```
m3-B
```

```
m4-B
```

Q) What are the differences between Classes and Abstract Classes?

-

Ans:

1. Classes are able to allow only concrete methods.

Abstract classes are able to allow zero or more concrete methods and zero or more abstract methods.

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

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

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

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

4. Classes are able to provide less shareability.

Abstract classes are able to provide more shareability.

Interfaces:

Interface is able to allow only abstract methods.

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

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

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

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

In Java applications, to use interfaces we have to use the following steps.

1. Declare an interface by using the “interface” keyword.
2. Declare variables and methods inside the interface.
3. Declare an implementation class for the interface.
4. Provide implementations for all the abstract methods of the interface inside the implementation class.
5. In the main class, in the main() method, create an object for the implementation class and declare the reference variables either for interface or for the implementation class.
6. Access the members of the interface.

Note: If we declare a reference variable for the interface then we are able to access only interface members, it is not possible to access implementation class own members. If we declare a reference variable of implementation class type then we are able

to access both interface members and the implementation's 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");
    }
}

public class Test {
    public static void main(String[] args) {
        //I i = 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);

}

}

nagoorn@Nagoors-MacBook-Pro COREJAVA-830 % javac Test.java
nagoorn@Nagoors-MacBook-Pro COREJAVA-830 % java Test
m1-A
m2-A
m3-A
m1-A
m2-A
m3-A
```

m4-A

10
10
10
10

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

Ans:

1. Classes are able to provide only concrete methods.

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

Interfaces are able to provide only abstract methods.

2. To declare classes only the “class” keyword is sufficient.

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

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

3. For classes we are able to create 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. INside the interfaces, by default all variables are “public static final”.

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

5. Inside the interfaces, by default all methods are “public and abstract”.

No default cases exist for the methods inside 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.

7. In classes and abstract classes constructors are possible.

Interfaces do not allow constructors.

8. Classes and abstract classes are able to allow static blocks, instance blocks,..

INterfaces do not allow static blocks, instance blocks,....

9. Inside the interfaces, by default all inner classes are static inner classes.

No default cases exist from the inner classes inside the classes and abstract classes.

10. Interfaces are mainly for declaring the Services.

Classes are mainly for implementing Services which are defined by the interfaces.

Abstract classes are mediators between interface and classes , they are able to provide some services implementation and some other services declarations.

IDEs:

1. IDE: Integrated Development Environment
2. The main purpose of the IDE is to simplify and Speedup the application development.
3. IDEs are providing Auto Compilation, no need to compile java code explicitly.
4. Application execution is very simple , just click on the Run button.
5. IDEs are able to provide API Help to write code fastly.
6. IDEs are able to provide shortcuts for the code completion.
7. No need to set path and classpath environment variables explicitly.
8. IDEs are able to provide an integrated environment to get servers, Databases,
Debugging tools,.....
9. Some IDEs are able to provide Java software internally , no need to install
Explicitly.
10. IDEs are able to perform the tasks like Start Server, Deploy applications, Open Browser and access the application automatically when we click on the run button.

EX: Eclipse, IntelliJ Idea,....

Methods in Java:

Method is a set of instructions representing a particular action of an entity, it will be executed when we access it.

In Java applications, to prepare methods we have to use the following syntax.

```
[Access Modifiers] ReturnType methodName([ParamList])[throws  
ExceptionList]{
```

```
}
```

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

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

Where returnType is able to specify a particular data type in order to return a particular value from the method, where the value may be generated as the result of the action that method performs.

All Java methods are able to allow all primitive data types, all User defined data types and ‘void’ as return types.

Where ‘void’ is representing no data is available to return from the method.

Where methodName is an identity to the method in order to recognize and access that method.

Where the purpose of the method Parameters is to provide input data to the method in order to perform an action.

All Java methods are able to allow all primitive data types, user defined data types as parameters.

Where the ‘throws’ keyword is able to bypass the generated exception from the present method to the caller method in order to handle it.

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

1. public void m1(){ } -----> Valid
2. protected void m1(){ } -----> Valid
3. void m1(){ } -----> Valid
4. private void m1(){ } -----> Valid
5. static void m1(){ } -----> Valid
6. abstract void m1(){ } -----> Invalid
7. abstract void m1(); -----> Valid
8. native void m1(); -----> Valid
9. final void m1(){ } -----> Valid
10. final abstract void m1(){ } -----> Invalid
11. volatile void m1(){ } -----> Invalid
12. transient void m1(){ } -----> Invalid
13. synchronized void m1(){ } -----> Valid
14. synchronized final void m1(){ } -----> Valid
15. Strictfp void m1(){ } ----->
Valid
16. int m1(){ return 10; } ----->
Valid
17. float m1(){ return 22.22f; } ----->
Valid

18. long m1(){ return 10; } ----->
Valid

19. double m1(){ return 22.22f; } ----->
Valid

20. float m1(){ return 22.222; } ----->
Invalid

21. int m1(){ } ----->
Invalid

22. void m1(){ } ----->
Valid

23. void m1(){ return 10; } ----->
Invalid

24. A m1(){ A a = new A(); return a; } ----->
Valid

25. A m1(){ B b = new B(); return b; } ----->
Invalid

26. void m1(int i, float f){ } ----->
Valid

27. void m1(A a){ } ----->
Valid

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

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

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

31. void m1() throws Exception{ } ----->
Valid

32. void m1()throws Arithmeticexception, NullPointerException
{ } -----> Valid

EX:

```
class B{  
    String convertMessage(String message){
```

```
String newVal = message.toUpperCase();
System.out.println(newVal);
return newVal;
}
}
class A{
    void m1(String str){
        System.out.println(str);
    }
    String m2(String name){
        String newString = name+"@durgasoft.com";
        return newString;
    }
    int[] m3(int[] values){
        int[] newValues = new int[values.length];
        for(int i = 0; i < values.length; i++){
            newValues[i] = values[i]*2;
        }
        return newValues;
    }
    String m4(B b){
        String str = b.convertMessage("durga software solutions");
        return str;
    }
}
public class Main {
    public static void main(String[] args) {
        A a = new A();
        a.m1("Hello m1() Method");
        String str = a.m2("Durga");//String
        str="durga@durgasoft.com";
        System.out.println(str);
        int[] values = {10,20,30,40,50};
        int[] newValues = a.m3(values);
```

```
        for(int i = 0; i < newValues.length; i++){
            System.out.print(newValues[i]+ " ");
        }
        System.out.println();
        B b = new B();
        String str1 = a.m4(b);
        System.out.println(str1);
    }
}
```

Hello m1() Method

Durga@durgasoft.com

20 40 60 80 100

DURGA SOFTWARE SOLUTIONS

DURGA SOFTWARE SOLUTIONS

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 Prototype?

Ans:

Method Signature is the method description that includes Method Name, Method parameter list.

EX: forName(Class cls)

Method Prototype is the Method Description that includes Access Modifiers list, return type, method name, parameter List, throws exception List.

EX: `public static Class forName(Class cls) throws ClassNotFoundException`

In Java applications, there are two types of methods as per the Object state manipulation.

1. Mutator Methods
2. Accessor Methods

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

Ans:

Mutator Methods are the normal java methods, they are able to set / modify the data in the Object.

EX: All `setXXX()` methods in Java Bean classes are Mutator Methods.

Accessor Methods are the normal java methods, they are able to access / read / get data from the Object.

EX: All `getXXX()` methods in java Bean classes are Accessor Methods.

Java Bean is a normal Java class , it is able to represent an entity in java applications, in Java bean classes we will provide

all properties as private properties, we must provide a separate set of setXXX() and getXXX() methods for each and every property with public.

EX:

```
class User{

    private String uname;
    private String upwd;

    public void setUsername(String userName){
        uname = userName;
    }
    public void setPassword(String userPassword){
        upwd = userPassword;
    }

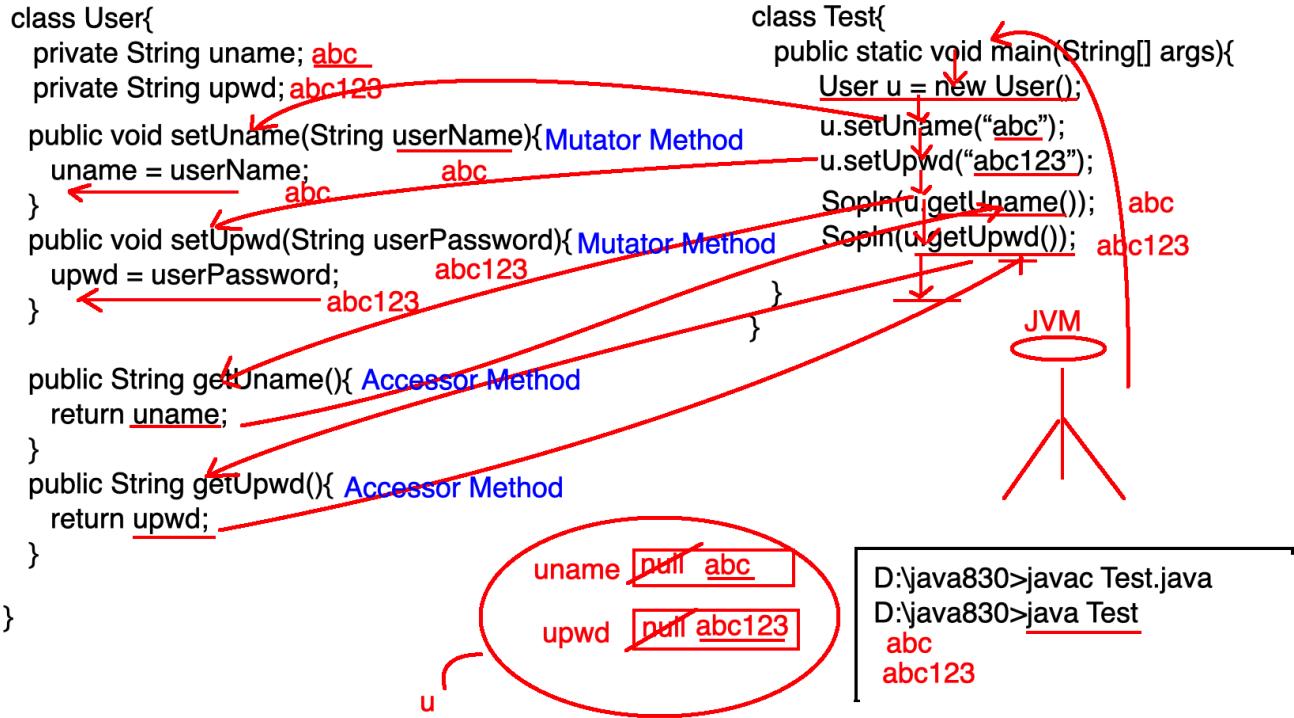
    public String getUsername(){
        return uname;
    }
    public String getPassword(){
        return upwd;
    }
}

public class Main {
    public static void main(String[] args) {
        User user = new User();
        user.setUsername("durga");
        user.setPassword("durga123");
        System.out.println("User Details");
        System.out.println("-----");
        System.out.println("User Name      : " + user.getUsername());
        System.out.println("User Password : " + user.getPassword());
    }
}
```

```

    }
}

```



In the above application, we have declared all properties are private and all methods are public, in this case to provide data to the variables we have to access `setXXX()` method and to get data from the properties we have to access `getXXX()` methods, here the data is flowing through the methods, here methods are able to bind both data and coding part, it will improve “Encapsulation”.

Note: In Java application, to achieve Encapsulation we have to use Java Bean components

EX:

```

class Employee {
    private int eno;

```

```
private String ename;
private float esal;
private String eaddr;

// Getter and Setter methods
public int getEno() {
    return eno;
}

public void setEno(int enoParam) {
    eno = enoParam;
}

public String getEname() {
    return ename;
}

public void setEname(String enameParam) {
    ename = enameParam;
}

public float getEsal() {
    return esal;
}

public void setEsal(float esalParam) {
    esal = esalParam;
}

public String getEaddr() {
    return eaddr;
}
```

```

public void setEaddr(String eaddrParam) {
    eaddr = eaddrParam;
}
}

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

        Employee emp = new Employee();
        emp.setEno(111);
        emp.setEname("Durga");
        emp.setEsal(5000.0f);
        emp.setEaddr("Hyd");
        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 Details

Employee Number	:	111
Employee Name	:	Durga
Employee Salary	:	5000.0
Employee Address	:	Hyd

Java Beans:

Java Bean is a reusable software component, it is able to manage the state of an entity.

In general, in enterprise applications we will use Java beans in the following situations.

1. To manage user data at Server side applications.
 2. To perform data Validations in java applications.
 3. To manage the Persistence data in order to store in the databases in the Database applications.
 4. To transfer the data from the controller layer to the View layer in MVC based applications.
-
-

To prepare Java Bean components we have to use the following guidelines.

1. All Java Bean classes must be declared with public, non abstract and non final.

Where the purpose of public is to make available java bean classes throughout the application.

Where the purpose of Non Abstract is to allow creating objects.

Where the Purpose of non final to allow inheritance relation between Java bean class to improve Reusability.

2. Every Java Bean class must implement `java.io.Serializable` interface in order to make javabean objects eligible to travel in the network.

3. In Java Bean classes, we have to provide variables w.r.t the entities, where the entities may be an user form in the web applications or a Table in the Databases.
4. In Java Bean classes, we have to provide a separate setter and getter method for each and every variable.
5. In Java Bean classes, we have to provide all variables or properties as private and all setters and getters as public in order to provide encapsulation.
6. In Java Bean classes, if we want to provide a constructor then provide a public and zero-argument constructor.
7. In Java Bean classes, if we want to provide our own comparison between the bean components then override equals() method.
8. In Java Bean classes, if we want to provide our own hashCode values then override hashCode() method.

Var-Ag Methods:

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, it is not possible to access that method by passing 'n-1' number of parameters values and 'n+1' number of parameter values.

EX:

```
class A{  
    void add(int i, int j){
```

```
        System.out.println(i+j);
    }
}

class Test{
    public static void main(String[] args) {
        A a = new A();
        //a.add(); -----> Error
        //a.add(10); -----> Error
        //a.add(10,20,30); -----> Error
        a.add(10,20);
    }
}
```

```
nagoorn@Nagoors-MacBook-Pro COREJAVA-7 % javac Test.java
nagoorn@Nagoors-MacBook-Pro COREJAVA-7 % java Test
30
```

In Java applications, as per the requirement we want to access a method by passing a variable number of parameter values , for this requirement we have to use Variable-Argument Methods, in short Var-Arg method.

If any java method has a Var-Arg parameter then that java method is called Var-Arg method.

Var-Arg Parameter Syntax:

DataType ... VarName

EX:

```
void m1(int ... i){// int[] i = {10,20,30}
-----
}
```

We are able to access the m1() method by passing any number of integer parameter values.

```
m1(); -----> Valid
m1(10); -----> Valid
m1(10,20);-----> Valid
m1(10,20,30);-----> Valid
m1(22.22f); -----> Invalid
```

When we access the Var-Arc method with some parameter values , JVM will convert the Var-Arc Parameter into an array of the same Var-ARc parameter data type in order to store all the provided parameter values.

EX:

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

```
        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);
    }
}
```

```
nagoorn@Nagoors-MacBook-Pro COREJAVA-7 % javac Test.java
nagoorn@Nagoors-MacBook-Pro COREJAVA-7 % java Test
```

```
No Of Arguments : 0
```

```
Argument Values :
```

```
Arguments SUM : 0
```

```
-----
```

```
No Of Arguments : 1
```

```
Argument Values : 10
```

```
Arguments SUM : 10
```

```
-----
```

```
No Of Arguments : 2
```

```
Argument Values : 10 20
```

```
Arguments SUM : 30
```

```
-----
```

```
No Of Arguments : 3
```

```
Argument Values : 10 20 30
```

```
Arguments SUM : 60
```

```
-----
```

Q) Is it possible to provide normal parameters in the Var-Arg method?

Ans:

Yes, It is possible to provide normal parameters along with the Var-Arg parameter in the Var-Arg method, but the normal parameters must be provided before the Var-Arg parameter only, we must not provide normal parameters after the Var-Arg Parameter in the Var-Arg method , because in Var-Arg Method the var-Arg parameter must be the last parameter.

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 var-Arg method the Var-ARg parameter must be the last parameter, if we provide more than one Var-Arg parameter in a single var-Arg method then only one Var-Arg parameter is possible to provide as last parameter, all remaining Var-arg parameters are not possible to provide last Parameters.

EX:

```
class A{  
    void m1(int ... i, float f){  
        System.out.println("Var-Arg Method");  
    }  
}
```

```

}

public class Test {
    public static void main(String[] args) {
        A a = new A();
        a.m1(22.22f);
        a.m1(10, 22.22f);
        a.m1(10,20, 22.22f);
        a.m1(10,20,30, 22.22f);
    }
}

```

```

nagoorn@Nagoors-MacBook-Pro COREJAVA-830 % javac Test.java
Test.java:2: error: varargs parameter must be the last parameter
    void m1(int ... i, float f){

```

EX:

```

class A{
    void m1(float f,int ... i){
        System.out.println("Var-Ag Method");
    }
}

public class Test {
    public static void main(String[] args) {
        A a = new A();
        a.m1(22.22f);
        a.m1(22.22f,10);
        a.m1(22.22f,10,20);
        a.m1(22.22f,10,20,30);
    }
}

```

```
}

}

nagoorn@Nagoors-MacBook-Pro COREJAVA-830 % javac Test.java
nagoorn@Nagoors-MacBook-Pro COREJAVA-830 % java Test
Var-Ar Method
Var-Ar Method
Var-Ar Method
Var-Ar Method
```

EX:

```
class A{
    void m1(float ... f,int ... i){
        System.out.println("Var-Ar Method");
    }
}

public class Test {
    public static void main(String[] args) {
        A a = new A();
        a.m1(22.22f);
        a.m1(22.22f,10);
        a.m1(22.22f,10,20);
        a.m1(22.22f,10,20,30);
    }
}
```

```
nagoorn@Nagoors-MacBook-Pro COREJAVA-830 % javac Test.java
Test.java:2: error: varargs parameter must be the last parameter
void m1(float ... f,int ... i){
```

Object Creation Process in Java:

Q)What is the requirement of Creating Objects in Java applications?

Ans:

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 in java applications.
2. To Store Entities data Temporarily we need Objects.
3. To access the members of any class we need an object.

To create Objects in Java applications we have to use the following Syntax.

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

ClassName([ParamValues]) is a Constructor call.

EX:

```
class A{
```

```
}
```

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

If we execute the above statement[new keyword], JVM will perform the following actions.

1. JVM will find the class to which JVM is creating the object that is the constructor name.

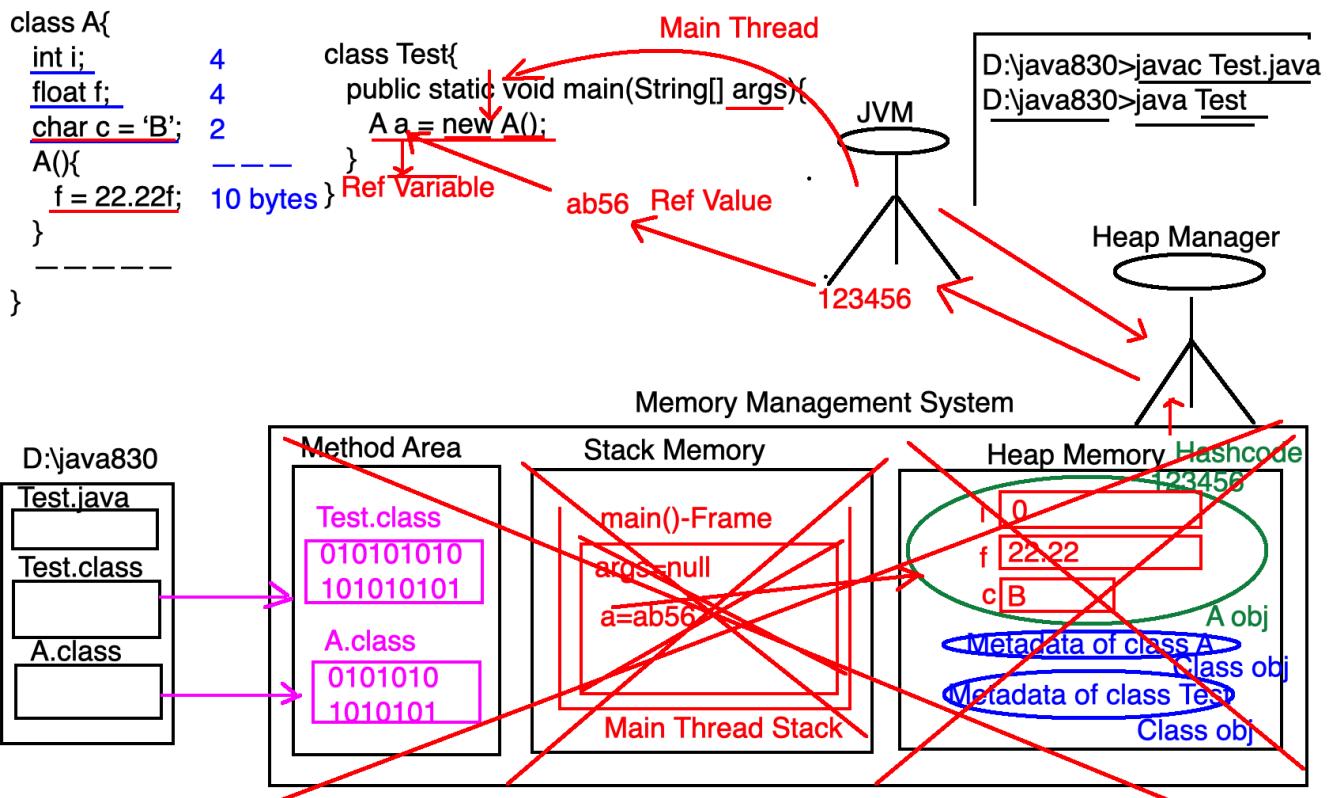
Note: IN Java , constructor is the same as the respective class name.

2. JVM will search for its .class file at the current location, at the Java predefined library and at the locations provided by the “classpath” environment variable.
3. If the required .class file is identified at either of the above locations then JVM will load the respective class bytecode to the memory that is in Method Area.

Note: When a class bytecode is loaded, automatically JVM will create java.lang.Class object in heap memory with the metadata of the loaded class, here the metadata includes class name, access modifiers of the class, superclass details, implemented interfaces details, variables details, methods details,.....

4. JVM will find the minimal object size on the basis of the instance variables and their data types of the loaded class and JVM will request the Heap manager to create an object of the provided size.
5. Heap manager will create the required block of memory for the object and the Heap Manager create an unique identity in the form of an integer value called HashCode, here the Heap Manager will return the generated HashCode value to JVM.
6. JVM will convert the provided hashCode value to its Hexa-Decimal form called Reference value of the Object and JVM will assign the generated reference value to a variable called Reference Variable.
7. JVM will provide memory allocation for all the instance variables of the respective class.
8. JVM will provide initializations for the instance variables inside the object by checking at class level declarations and inside the constructor.

9. If any variable is not having initialization at both the locations then JVM will provide default value for that variable inside the object on the basis of its data type.



In Java applications, 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 a non Java programming language, `hashCode()` method is a native method declared in Java and implemented in a H/W language to return Hashcode value.

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

```
public String toString()
```

EX:

```
class A{  
}  
  
public class Main {  
    public static void main(String[] args) {  
        A a = new A();  
        int hc = a.hashCode();  
        System.out.println("Hashcode      : "+hc);  
        String ref = a.toString();  
        System.out.println("Ref Value     : "+ref);  
    }  
}  
HashCode      : 2055281021  
Ref Value     : A@7a81197d// ClassName@RefValue
```

In Java, for all the classes there is a common and default superclass in the form of `java.lang.Object` class, here the `java.lang.Object` class contains the following 11 methods in order to provide to all the java classes.

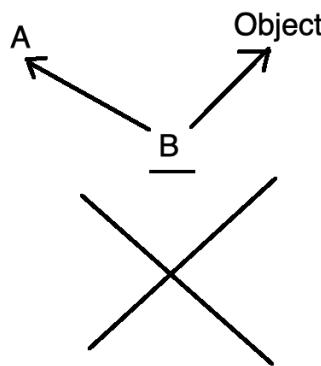
1. `hashCode()`
2. `toString()`
3. `equals()`
4. `getClass()`
5. `clone()`
6. `finalize()`
7. `wait()`
8. `wait(int)`
9. `wait(int, long)`
10. `notify()`
11. `notifyAll()`

Q) In Java applications, if we provide a superclass for a subclass explicitly then the respective subclass has two superclasses in the form java.lang.Object class and the provided explicit superclass, it represents Multiple Inheritance, how can we say that the Multiple inheritance is not possible in Java?

Ans:

In Java, java.lang.Object class is a common and default superclass for any class when the classes are not extended from any superclass explicitly, if a class is extending from a superclass explicitly then the java.lang.Object class is indirectly a superclass to the respective subclass that is through MultiLevel Inheritance but not through Multiple INheritance.

```
class A{  
}  
class B extends A{  
}
```



In Java applications, if we pass an object reference variable as a parameter to the `System.out.println()` method then JVM will access `toString()` method over the provided reference variable internally , here the return value of the `toString()` method will be provided as parameter to `System.out.println()` method and it will display the return value on console.

EX:

```
class A{  
}  
  
public class Main {  
    public static void main(String[] args) {  
        A a = new A();  
        String refVal1 = a.toString();  
        System.out.println(refVal1);  
  
        System.out.println(a.toString());  
  
        System.out.println(a); // System.out.println(a.toString());  
    }  
}
```

A@7a81197d

A@7a81197d

A@7a81197d

In java applications, when we access `toString()` method or when JVM access `toString()` method internally, JVM will perform the following actions.

1. JVM will goto the class on which reference variable that `toString()` was accessed.
2. IN the respective class, JVM will search for the `toString()` method.
3. If the `toString()` method was defined in the respective class directly then JVM will execute that `toString()` method.
4. In this case, if the `toString()` method was not defined in the respective class then JVM will search for the `toString()` method in the superclass.

5. If no superclass exists for the class explicitly then JVM will search for the `toString()` method in the default superclass `java.lang.Object` class.
6. In `java.lang.Object` class, the `toString()` method was implemented in such a way that to return a String that contains “`ClassName@RefValue`”.

In the above Context, if we don't want to display the object reference value then we must not execute the `java.lang.Object` class provided `toString()` method, instead, we have to provide and execute our own `toString()` method in the respective class.

EX:

```
class Account{
    String accNo = "abc123";
    String accHolderName = "Durga";
    String accType = "Savings";
    long balance = 50000L;

    public String toString(){
        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);
        return "-----";
    }
}

public class Main {
    public static void main(String[] args) {
        Account account = new Account();
        System.out.println(account); // SopLn(account.toString());
    }
}
```

```
 }  
}
```

Account Details

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

In Java, some of the predefined classes like String, StringBuffer, Exception, Thread, Wrapper classes, Collection classes,.... Are not depending on the java.lang.Object class provided `toString()` method, they have their own `toString()` method to display their own details when we pass the respective class reference variable as parameter to `System.out.println();`

EX:

```
import java.util.ArrayList;  
  
public class Main {  
    public static void main(String[] args) {  
        String str = new String("Welcome To String  
Manipulations");  
        System.out.println(str);  
  
        ArithmeticException exception = new  
ArithmeticeException("My Arithmetic Exception");  
        System.out.println(exception);  
  
        Thread thread = new Thread();  
        System.out.println(thread);
```

```
ArrayList list = new ArrayList();
list.add(10);
list.add(20);
list.add(30);
list.add(40);
System.out.println(list);
}
}

Welcome To String Manipulations
java.lang.ArithmetricException: My Arithmetic Exception
Thread[Thread-0,5,main]
[10, 20, 30, 40]
```

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?

Ans:

Immutable Objects are the Java objects , they are not allowing modifications on their content, if we are trying to perform modifications on the immutable object's data then the data is allowed for modifications , but the resultant modified data will not be stored back in the original object, here the modified resultant data will be stored by creating another new object.

EX: String class objects are Immutable.

EX: All Wrapper classes objects are immutable.

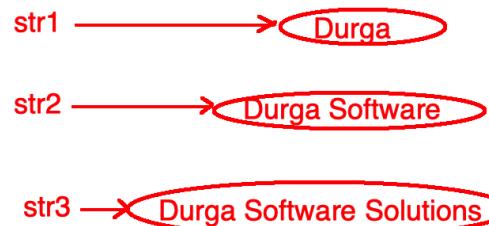
Mutable Objects are the Java Objects, they are able to allow the modifications directly on their content.

EX: In Java, by default all Objects are mutable.

EX: StringBuffer class objects are mutable

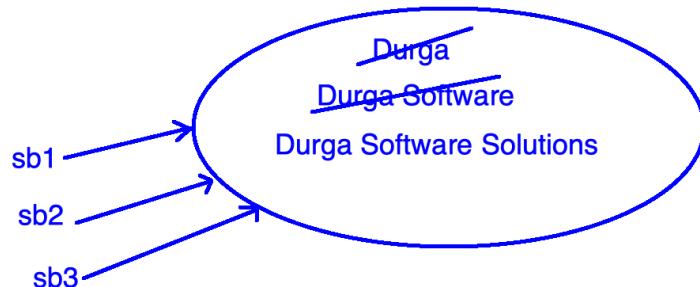
```
String str1 = new String("Durga ");  
String str2 = str1.concat("Software ");  
String str3 = str2.concat("Solutions");
```

```
System.out.println(str1); Durga  
System.out.println(str2); Durga Software  
System.out.println(str3); Durga Software Solutions
```



```
StringBuffer sb1 = new StringBuffer("Durga ");  
StringBuffer sb2 = sb1.append("Software ");  
StringBuffer sb3 = sb2.append("Solutions");
```

```
System.out.println(sb1);  
OP: Durga Software Solutions  
System.out.println(sb2);  
OP: Durga Software Solutions  
System.out.println(sb3);  
OP: Durga Software Solutions
```



EX:

```
public class Main {  
    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);  
        System.out.println(str2);  
        System.out.println(str3);  
        System.out.println(str1 == str2);  
        System.out.println(str2 == str3);
```

```
System.out.println(str3 == str1);
System.out.println();

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

}

}
```

```
Durga
Durga Software
Durga Software Solutions
false
false
false
```

```
Durga Software Solutions
Durga Software Solutions
Durga Software Solutions
true
true
true
```

Q) What is the difference between Object and instance?

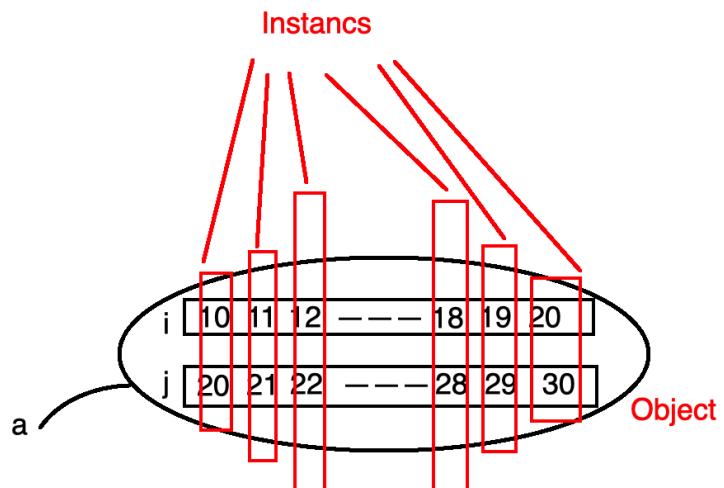
Ans:

Object is a Block memory to store data.

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

IN Java applications, an Object can have multiple instances, but we are always able to get the latest instance , because all the instances are provided in an overriding manner.

```
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  
        }  
    }  
}
```



Constructors:

1. Constructor is a Java feature, it can be used to create Objects.
2. The role of the constructors in the Object Creation Process is to provide initial values inside the Objects.
3. In Java applications, Constructors are executed exactly at the time of creating objects, 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. In Java, constructors' names must be the same as the respective class name.
6. Constructors are not having return types.
7. Constructors do not allow the access modifiers like Static, final, abstract,....
8. Constructors are able to allow the access modifiers like public, protected, <default>, private.
9. Constructors are able to allow the throws keyword to throw out the generated Exception from the present constructor to the caller.

Syntax:

```
[Access Modifier] className([ParamList])[throws ExceptionList]{  
    ---instructions---  
}
```

EX:

```
class A{  
    A(){  
        System.out.println("A-Con");  
    }  
}  
public class Main {
```

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

In Java applications, we must provide the same class name to the constructors, in the case if we provide a different name to the constructor which is not the same as the respective class name then the compiler will raise an error like “invalid method declaration; return type required”, because if we provide different name[Not same as the class name] to the constructor then the compiler will treat the provided constructor as a normal java method without the return type, in normal Java methods return type is mandatory.

EX:

```
class A{
    B(){
        System.out.println("A-Con");
    }
}

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

nagoorn@Nagoors-MacBook-Pro COREJAVA-830 % javac Test.java
Test.java:2: error: invalid method declaration; return type
required
    B(){
```

In Java , Constructors are not having return types, in the case if we provide a return type to the constructor then the compiler will not raise any error, because the compiler has treated the provided constructor as a normal java method and its syntax is matched with the java method methods syntax, in this case if we access the provided constructor as like a normal java method then JVM is able to execute the provided constructor.

EX:

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

```
nagoorn@Nagoors-MacBook-Pro COREJAVA-830 % javac Test.java  
nagoorn@Nagoors-MacBook-Pro COREJAVA-830 % java Test  
A-Con
```

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

EX:

```
class A{
    static A(){
        System.out.println("A-Con");
    }
}

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

```
nagoorn@Nagoors-MacBook-Pro COREJAVA-830 % javac Test.java
Test.java:2: error: modifier static not allowed here
    static A(){
```

In Java applications, it is possible to provide the access modifiers like public, protected, <default> and private to the constructors, if we declare a constructor with private access modifier then that constructor is having scope up to the respective class only, it is not available outside of the respective class, so we must access the private constructor with in the same class, not possible to access the private constructor outside of the respective class.

EX:

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

```

}

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

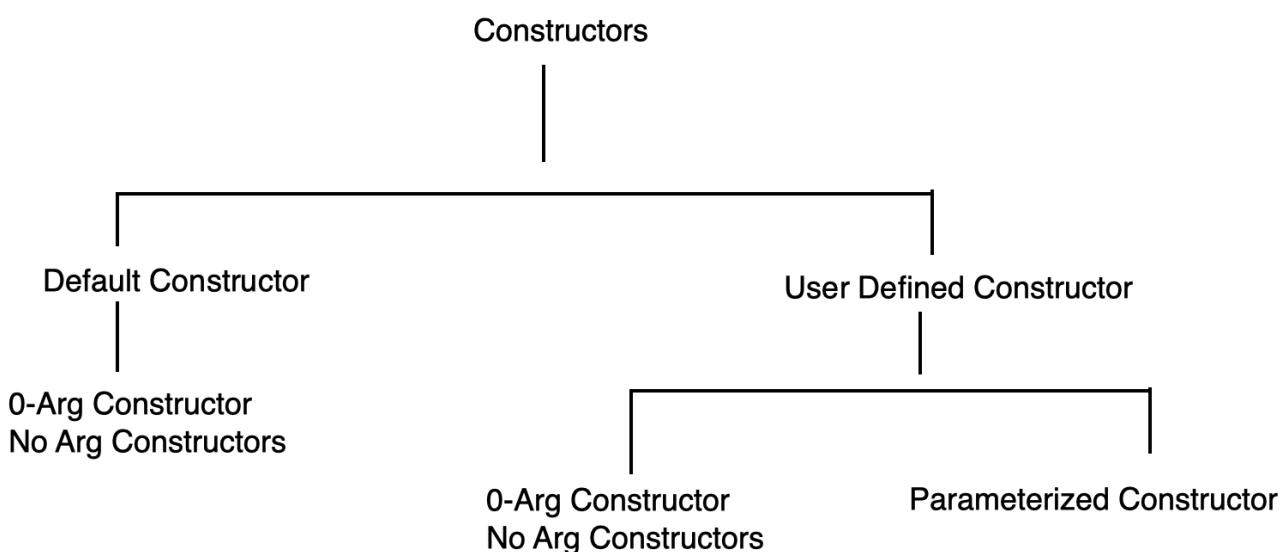
```

nagoorn@Nagoors-MacBook-Pro COREJAVA-830 % javac Test.java
Test.java:8: error: A() has private access in A

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

In Java, there are two types of constructors.

1. Default constructors
2. User Defined Constructors



Default Constructor:

If we declare a class without providing any constructor explicitly then the Compiler will add a 0-arg constructor to the class, here the compiler provided 0-arg constructor is called “Default Constructor”, if we provide at least one constructor explicitly in the class then the compiler will not provide default constructor.

Note: INnJava, by default, the default constructor is a 0-arg constructor, but all 0-arg constructors need not be default constructors, here the 0-arg constructors which are provided by the compiler are the default constructors and the 0-arg constructors which are provided by the developers are called user defined constructors.

EX:

Test.java

```
public class Test {
```

```
}
```

```
nagoorn@Nagoors-MacBook-Pro COREJAVA-830 % javac Test.java
```

```
nagoorn@Nagoors-MacBook-Pro COREJAVA-830 % javap Test
```

```
Compiled from "Test.java"
```

```
public class Test {
```

```
    public Test();-----> Default Constructor
```

```
}
```

The scope of the default constructor is the same as the scope of the respective class.

EX:

Test.java

```
class Test {
```

```
}
```

```
nagoorn@Nagoors-MacBook-Pro COREJAVA-830 % javac Test.java
```

```
nagoorn@Nagoors-MacBook-Pro COREJAVA-830 % javap Test
```

```
Compiled from "Test.java"
```

```
class Test {
```

```
    Test(); --> Default Constructor
```

```
}
```

User Defined Constructor:

These Constructors are provided by the developers as per their application requirements.

There are two types of User defined constructors.

1. 0-Ang Constructor: If we declare a constructor without the parameters then that constructor is a 0-arg constructor.
2. Parameterized Constructor: If we declare a constructor with at least one parameter then that constructor is a Parameterized constructor.

EX:

```
class Account{
```

```
    String accNo;
```

```
    String accHolderName;
```

```
    String accType;
```

```
    long accBalance;
```

```
    public void setAccountDetails(String acc_No, String
```

```
acc_Holder_Name, String acc_Type, long acc_Balance){
```

```
        accNo = acc_No;
```

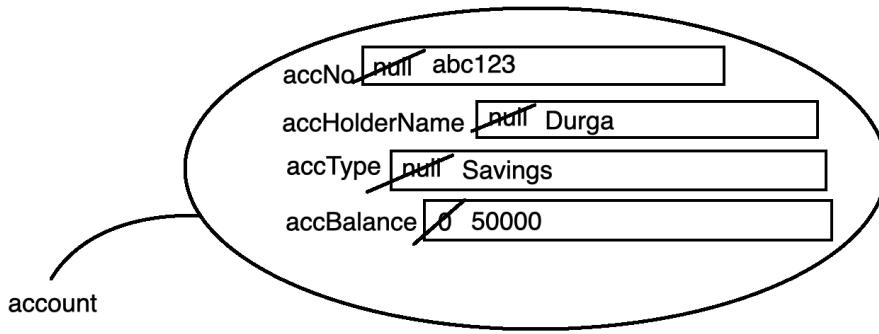
```
        accHolderName = acc_Holder_Name;
        accType = acc_Type;
        accBalance = acc_Balance;
    }

    public void getAccountDetails(){
        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);
    }

}

public class Main {
    public static void main(String[] args) {
        Account account = new Account();
        account.setAccountDetails("abc123", "Durga", "Savings",
50000);
        account.getAccountDetails();
    }
}
```

```
Account Details
-----
Account Number      : abc123
Account Holder Name : Durga
Account Type       : Savings
Account Balance    : 50000
```



In the above program, when we create `Account` class object , `Account` class Object is created with the default data like null values and 0, when we access `setAccountDetails()` method only all the account details are provided inside the object, this approach is against to the bank Application requirement, because in bank applications Account Details must be provided at the time of creating Account that is in Java applications All the Account Details must be provided at the time of creating Account Object.

In the above context, to provide all the Account details at the time of creating an `Account` class object we have to use Constructors, because in Java applications Constructors are executed exactly at the time of creating Objects.

EX:

```
class Account{

    String accNo;
    String accHolderName;
    String accType;
    long accBalance;

    Account(){
        accNo = "abc123";
        accHolderName = "Durga";
    }
}
```

```

        accType = "Savings";
        accBalance = 50000L;
    }

    public void getAccountDetails(){
        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);
    }

}

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

```

Account Details

```

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

```

In the above application, if we create more than one object then all the objects will have the same data, because we have provided hard coded values to the instance variables inside the constructor, it will provide the same data in all the objects.

```
class Account{

    String accNo;
    String accHolderName;
    String accType;
    long accBalance;

    Account(){
        accNo = "abc123";
        accHolderName = "Durga";
        accType = "Savings";
        accBalance = 50000L;
    }

    public void getAccountDetails(){
        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);
    }
}

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

        Account account2 = new Account();
        account2.getAccountDetails();
        System.out.println();
```

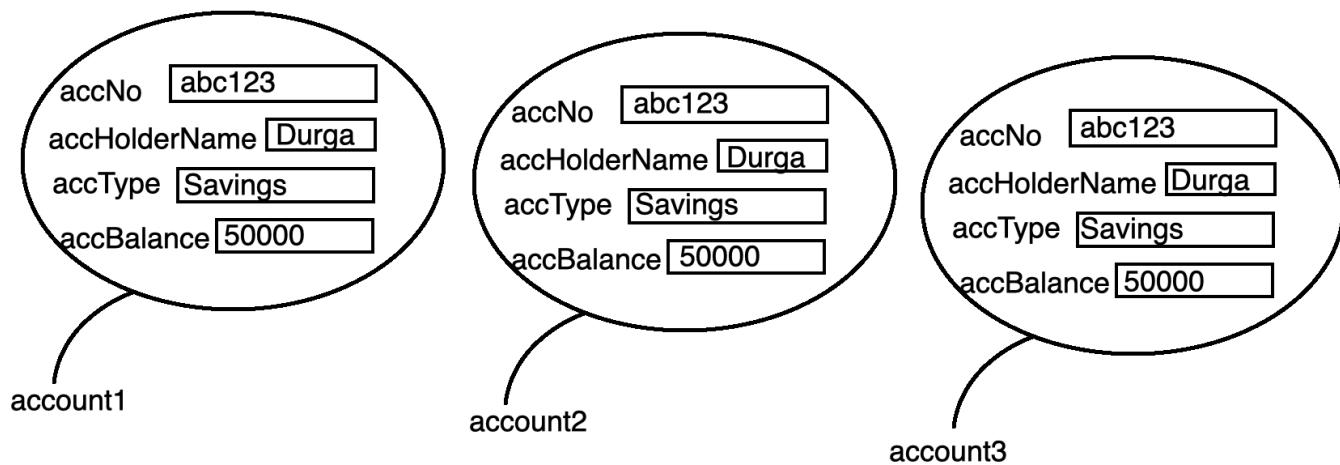
```

        Account account3 = new Account();
        account3.getAccountDetails();

    }

}

```



In the above application we have to create multiple `Account` objects with the different data, not the same data, in this context if we want to provide our own data at the time of creating objects then we have to use Parameterized constructors.

```

class Account{

    String accNo;
    String accHolderName;
    String accType;
    long accBalance;

    Account(String acc_No, String acc_Holder_Name, String
    acc_Type, long acc_Balance){
        accNo = acc_No;
    }
}

```

```
    accHolderName = acc_Holder_Name;
    accType = acc_Type;
    accBalance = acc_Balance;
}

public void getAccountDetails(){
    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);
}

}

public class Main {
    public static void main(String[] args) {
        Account account1 = new Account("a111", "Durga", "Savings",
50000L);
        account1.getAccountDetails();
        System.out.println();

        Account account2 = new Account("a222", "Venkat",
"Savings", 60000L);
        account2.getAccountDetails();
        System.out.println();

        Account account3 = new Account("a333", "Ramana", "Savings",
40000L);
        account3.getAccountDetails();
    }
}
```

Account Details

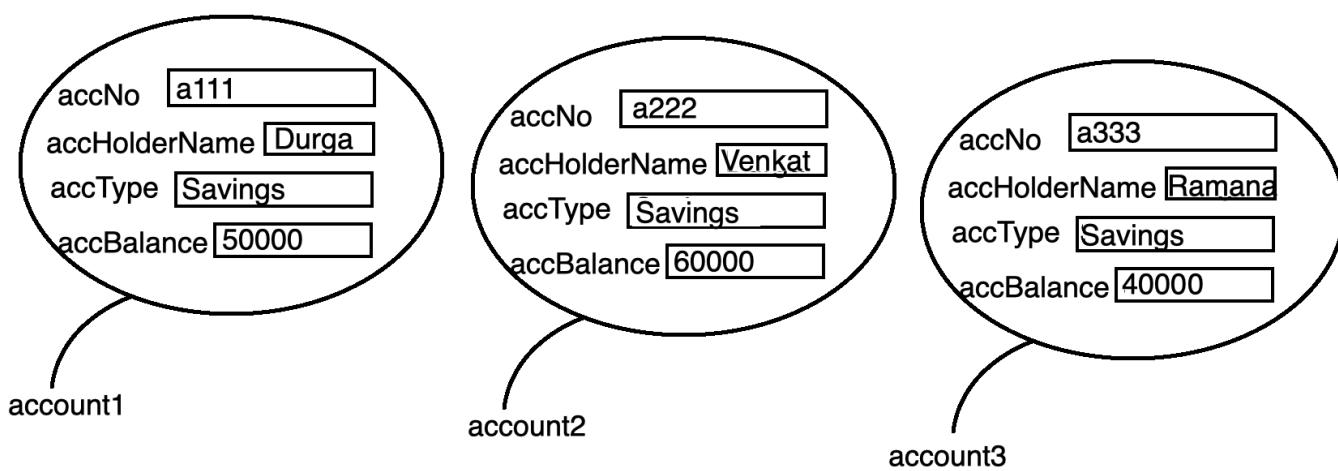
Account Number : a111
Account Holder Name : Durga
Account Type : Savings
Account Balance : 50000

Account Details

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

Account Details

Account Number : a333
Account Holder Name : Ramana
Account Type : Savings
Account Balance : 40000



Q)What is Constructor Overloading?

Ans:

If we provide more than one constructor with the same name and with the different parameter list then it is called Constructor Overloading.

The main advantage of the Constructor Overloading is flexibility to design applications.

EX:

```
class Student{
    String sid;
    String sname;
    String semail;
    String smobile;

    Student(String studentId){
        sid = studentId;
    }
    Student(String studentId, String studentName){
        sid = studentId;
        sname = studentName;
    }
    Student(String studentId, String studentName, String
studentEmail){
        sid = studentId;
        sname = studentName;
        semail = studentEmail;
    }
    Student(String studentId, String studentName, String
studentEmail, String studentMobile){
        sid = studentId;
```

```
sname = studentName;
semail = studentEmail;
smobile = studentMobile;
}
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 Email   : "+semail);
    System.out.println("Student Mobile No : "+smobile);
}
}
public class Main {
    public static void main(String[] args) {
        Student s1 = new Student("S-111");
        s1.getStudentDetails();
        System.out.println();

        Student s2 = new Student("S-222","Durga");
        s2.getStudentDetails();
        System.out.println();

        Student s3 = new Student("S-333","Anil", "anil@dss.com");
        s3.getStudentDetails();
        System.out.println();

        Student s4 = new Student("S-444","Ramesh",
"ramesh@dss.com", "91-9988776655");
        s4.getStudentDetails();
    }
}
```

Student Details

```
Student Id      : S-111
Student Name    : null
Student Email   : null
Student Mobile No : null
```

Student Details

```
Student Id      : S-222
Student Name    : Durga
Student Email   : null
Student Mobile No : null
```

Student Details

```
Student Id      : S-333
Student Name    : Anil
Student Email   : anil@dss.com
Student Mobile No : null
```

Student Details

```
Student Id      : S-444
Student Name    : Ramesh
Student Email   : ramesh@dss.com
Student Mobile No : 91-9988776655
```

EX:

```
class A{
    int i, j , k;
    A(){
    }
}
```

```
A(int x){  
    i = x;  
}  
A(int x, int y){  
    i = x;  
    j = y;  
}  
A(int x, int y, int z){  
    i = x;  
    j = y;  
    k = z;  
}  
public void add(){  
    System.out.println("ADD      : "+(i+j+k));  
}  
}  
public class Main {  
    public static void main(String[] args) {  
        A a1 = new A();  
        a1.add();  
        A a2 = new A(10);  
        a2.add();  
        A a3 = new A(10,20);  
        a3.add();  
        A a4 = new A(10,20,30);  
        a4.add();  
    }  
}  
ADD      : 0  
ADD      : 10  
ADD      : 30  
ADD      : 60
```

Instance Context:

In Java applications, when we create an object, first we have to load the respective class bytecode to the memory, at the time of loading class bytecode to the memory an environment will be created to perform the activities at load time called Static Context.

In Java applications, in the object creation process, after loading class bytecode to the memory, when the object is created automatically an environment will be created to perform the activities while creating an object called Instance Context.

In Java, instance content is represented by the following three elements.

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

Instance Variables:

Instance variable is a normal java variable whose values will vary from one instance to another instance of an object.

Instance variables are the normal java variables, they will be recognized and initialized just before executing the respective class constructors.

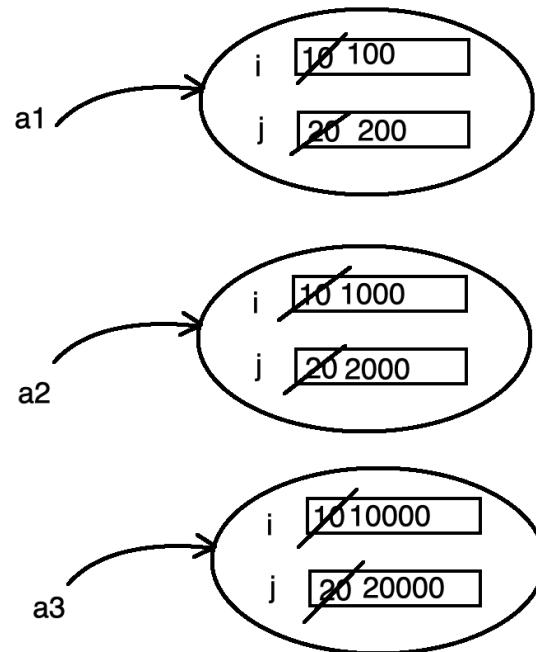
In Java applications, a separate copy of the instance variables are maintained for each and every object of the same class.

```
class A{  
    int i = 10;  
    int j = 20;  
}
```

```
A a1 = new A();  
a1.i=100;  
a1.j=200
```

```
A a2 = new A();  
a2.i=1000  
a2.j=2000
```

```
A a3 = new A();  
a3.i=10000  
a3.j=20000
```



In Java applications, to access the instance variables in the same class no need to use any reference variable or any keyword, directly we can access the current class instance variables without using reference variables and keywords, but if we want to access the instance variables outside of the respective class then we must create an object and we must use the generated reference variable of the respective class.

In Java applications, if we access any instance variable by using a reference variable that contains null value then JVM will raise an exception like `java.lang.NullPointerException`, because if the reference variable value is null then there is no object for the respective class, here without creating Object for the respective class there is no chance storing instance variable values.

In Java applications, always instance variables must be declared at class level only without using static keyword, not possible to declare as local variables.

In Java applications, instance variables data will be stored inside the Heap memory in the form of Objects.

EX:

```
class A{
    int i = 10;
    int j = 20;
}

class Test {
    public static void main(String[] args) {
        A a1 = new A();
        System.out.println(a1.i+" "+a1.j);
        a1.i=a1.i+10;
        a1.j=a1.j+10;
        System.out.println(a1.i+" "+a1.j);

        A a2 = new A();
        System.out.println(a2.i+" "+a2.j);
        a2.i=a2.i+100;
        a2.j=a2.j+100;
        System.out.println(a2.i+" "+a2.j);
    }
}

nagoorn@Nagoors-MacBook-Pro COREJAVA-830 % javac Test.java
nagoorn@Nagoors-MacBook-Pro COREJAVA-830 % java Test
10    20
20    30
```

```
10    20
110   120
```

EX:

```
class A{
    int i = 10;
    void m1(){
        System.out.println(i);
    }
}

class Test {
    public static void main(String[] args) {
        A a1 = new A();
        a1.m1();
        System.out.println(a1.i);

        A a2 = null;
        System.out.println(a2.i);
    }
}
```

```
nagoorn@Nagoors-MacBook-Pro COREJAVA-830 % javac Test.java
nagoorn@Nagoors-MacBook-Pro COREJAVA-830 % java Test
10
10
Exception in thread "main" java.lang.NullPointerException: Cannot
read field "i" because "<local2>" is null
at Test.main(Test.java:14)
```

Instance Method:

Instance method is a normal java method without the static keyword.

Instance method is a set of instructions to represent a particular action of an entity.

In Java applications, instance variables are recognized and initialized automatically just before executing the respective class constructor, but instance methods are recognized and executed the moment when we access it.

Instance Methods are able to operate on both instance variables and static variables.

In Java applications, in the current class we can access the instance methods directly without using any reference variable and without using any special keyword, but if we want to access the instance methods outside of the respective class then we must create an object for the respective class and we must use the generated reference variable.

In Java applications, if we access an instance method by using a reference variable that contains null value then JVM will raise an exception like `java.lang.NullPointerException`.

EX:

```
class A{
    int i = 10;
    static int j = 20;
    void m1(){
        System.out.println("m1-A");
```

```

        System.out.println(i);
        System.out.println(j);
    }
}

public class Main {
    public static void main(String[] args) {
        A a = new A();
        a.m1();
        A a1 = null;
        a1.m1();
    }
}

```

m1-A
10
20
Exception in thread "main" java.lang.NullPointerException: Cannot
invoke "A.m1()" because "a1" is null
at Main.main(Main.java:15)

EX:

```

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

```

```
    }  
}
```

m1-A
A-Con

Q)What are the differences between Constructor and Method?

Ans:

1. In Java, the main purpose of the constructor is to provide initializations to the objects.

In Java, the main purpose of the method is to represent and perform a particular task.

2. In Java, Constructors are executed automatically when we create an object, here in the object creation statement we will access the constructor along with a new keyword.

In Java, Methods are executed when we access them, without providing method calls methods won't be executed.

3. The Constructor's name must be the same as the respective class name.

In Java, method names may or may not be the same as the respective class name.

4. Constructors do not have return types.

Methods must have return types.

5. Constructors do not have the access modifiers like static, final,....

Methods may have access modifiers like static, final,....

6. If we provide initializations to the instance variables through a constructor then the provided initializations will be stored inside the objects as the first data at the time of creating objects, not after creating the objects.

If we provide initializations to the instance variables through a method then the provided initializations will be stored inside the objects as the second data after creating the objects, not at the time of creating the objects.

7. In a class, if we provide initializations to a variable through constructor and a method then JVM will provide constructor provided initializations first at the time of creating object, after creating the object if we access the method then JVM will override they constructor provided initializations with the method provided initializations, finally Object will have the method provided initializations.

EX:

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

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

A-Con

A-Method

EX:

```
class Employee{

    int eno = 111;
    String ename = "Durga";
    float esal = 5000.0f;
    String eaddr = "Hyd";

    Employee(){
        eno = 222;
        ename = "Anil";
        esal = 6000.0f;
        eaddr = "Chennai";
    }

    public void setEmployeeDetails(){
        eno = 333;
        ename = "Venkat";
        esal = 7000.0f;
        eaddr = "Pune";
    }

    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);
    }

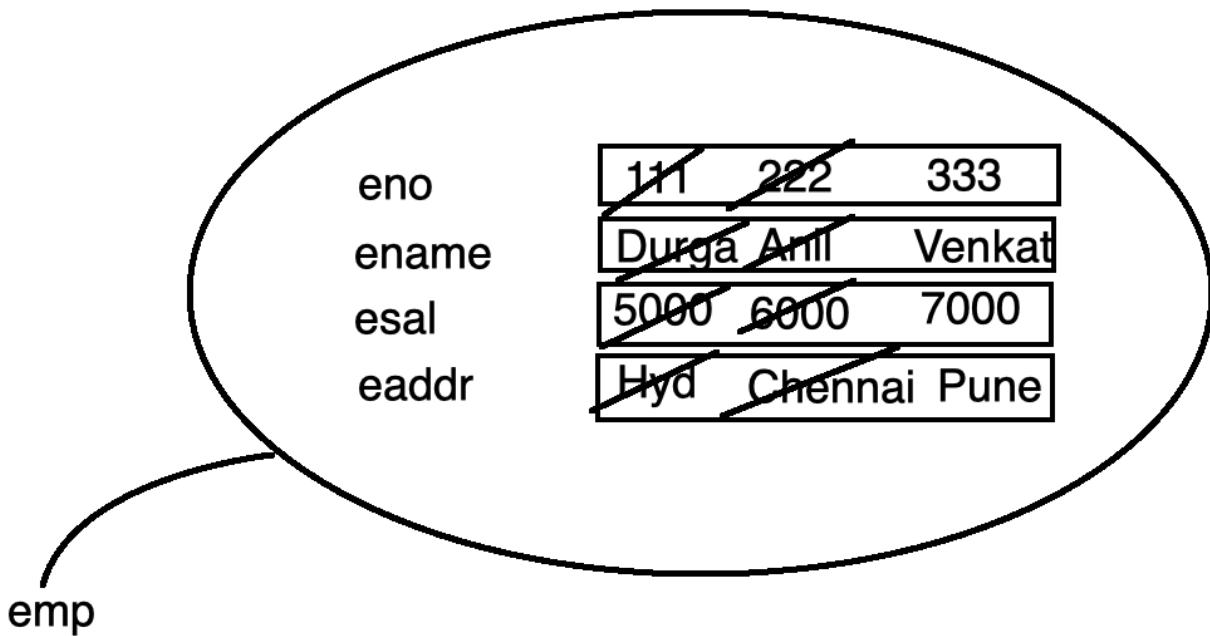
}

public class Main {
    public static void main(String[] args) {
        Employee emp = new Employee();
        emp.setEmployeeDetails();

        emp.getEmployeeDetails();
    }
}
```

Employee Details

```
Employee Number      : 333
Employee Name        : Venkat
Employee Salary      : 7000.0
Employee Address    : Pune
```



Instance Blocks:

It is a set of instructions, it will be recognized and executed automatically just before executing the respective class constructor.

Note: Instance blocks are having the same power of the constructors, but we are able to use constructors mainly in java applications.

Syntax:

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

EX:

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

```
}

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

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

IB-A

A-Con

Instance Flow of Execution:

In Java applications, instance context is represented by Instance variables, instance methods and instance blocks, where the instance variables and the instance blocks are recognized and executed automatically just before executing the respective class constructor and the instance method will be executed the moment when we access it.

Note: In Java classes, JVM will execute the instance context before the constructor execution in top to bottom order from starting point of the class to the ending point of the class.

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;
}
}

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

```

IB-A

m1-A

A-Con

EX:

```

class A{
    int i = m1();
    {
        System.out.println("IB-A");
    }
    int m1(){
        System.out.println("m1-A");
        return 10;
    }
    A(){
        System.out.println("A-Con");
    }
}

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

```

```
    }
}

m1-A
IB-A
A-Con
```

EX:

```
class A{
{
    System.out.println("IB1-A");
}
int i = m1();
int m1(){
    System.out.println("m1-A");
    return 10;
}
A(){
    System.out.println("A-Con");
}
{
    System.out.println("IB2-A");
}
int j = m2();
int m2(){
    System.out.println("m2-A");
    return 20;
}
}

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

IB1-A

m1-A

IB2-A

m2-A

A-Con

In Java applications, JVM will recognize and execute the instance context for each and every Object or every time executing the respective class constructor.

EX:

```
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");
    }
}
public class Main {
    public static void main(String[] args) {
        A a1 = new A();
        System.out.println();

        A a2 = new A();
    }
}
```

IB-A

m1-A
A-Con

IB-A
m1-A
A-Con

‘this’ keyword:

‘this’ is a keyword in Java, it is able to represent the current class object.

In Java applications, there are four ways to utilize the ‘this’ keyword.

1. To refer to the current class variables.
2. To refer to the current class methods.
3. To refer to the current class constructors
4. To return the current class object

To refer to the current class variables:

IN Java applications, to refer to the current class variables by using the ‘this’ keyword then we have to use the following syntax.

`this.varName`

Note: In Java applications, if we have the same variables at local and at the class level, where to refer to the current class level variables over the local variables there we have to use ‘this’ keyword.

EX:

```
class A{
    int i = 10;
    int j = 20;
    A(int i, int j){// i=50, j = 60
        System.out.println("Local Vars      : "+i+"      "+j);// 50
        60
        System.out.println("Class Vars      : "+this.i+
"+this.j); // 10    20
    }
}
public class Main {
    public static void main(String[] args) {
        A a = new A(50, 60);
    }
}
```

```
Local Vars      : 50      60
Class Vars      : 10      20
```

In Java applications, we will use Java Bean classes frequently, in java bean classes we have to declare a separate set of `setXXX()` and `getXXX()` methods for each and every variable, in this context we may use the same class level variable's name to the local parameter variable of the `setXXX()` methods, in the `setXXX()` methods we have to assign local parameter variable value to the class level variable, so we have to assign local parameter variable to the class level variable, in this case we have to refer the class level variable over the local variable, for this we have to use 'this' keyword.

EX:

```
class Employee{
```

```
private int eno;
private String ename;
private float esal;
private String eaddr;

public void setEno(int eno){
    this.eno = eno;
}
public void setEname(String ename){
    this.ename = ename;
}
public void setEsal(float esal){
    this.esal = esal;
}
public void setEaddr(String eaddr){
    this.eaddr = eaddr;
}

public int getEno(){
    return eno;
}
public String getEname(){
    return ename;
}
public float getEsal(){
    return esal;
}
public String getEaddr(){
    return eaddr;
}

}

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

```

Employee emp = new Employee();
emp.setEno(111);
emp.setEname("Durga");
emp.setEsal(60000.0f);
emp.setEaddr("Hyd");

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 Details

Employee Number	:	111
Employee Name	:	Durga
Employee Salary	:	60000.0
Employee Address	:	Hyd

2. To refer to the current class methods:

In Java applications, to access the current class methods by using 'this' keyword then we have to use the following syntax.

`this.methodName([ParamValues]);`

Note: In Java applications, we are able to access the current class methods without using any reference variable, we can access it directly. We can use this keyword also to access the current class method.

EX:

```
class A{
    void m1(){
        System.out.println("m1-A");
        m2();
        this.m2();
    }
    void m2(){
        System.out.println("m2-A");
    }
}
public class Main {
    public static void main(String[] args) {
        A a = new A();
        a.m1();
    }
}
```

m1-A

m2-A

m2-A

To refer to the current class constructors:

In Java applications, to access the current class constructor by using this keyword then we have to use the following syntax.

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

```
this();      : to access 0- arg constructor.  
this(10);    : to access int parameterized constructor  
this(10,20) : to access two int parameterized constructor  
-----  
-----
```

Note: If we access the constructor along with the “new” keyword then the constructor execution will be the part of the object creation process, where it will provide initializations inside the object. If we access the constructor with the “this” keyword then it will not be the part of the Object creation process, it will be the part of accessing the current class constructor individually.

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(33.3333);  
        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();
    }
}

```

nagoorn@Nagoors-MacBook-Pro COREJAVA-830 % javac Test.java

nagoorn@Nagoors-MacBook-Pro COREJAVA-830 % java Test

A-double-param-con

A-float-param-con

A-int-param-con

A-Con

If we want to access the current class constructor by using the “this” keyword then we must follow the below conditions.

1. The respective “this” statement must be the first statement.
2. We must use the “this” statement in the constructor only, not in the normal java method.

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(33.333);
}
A(double d){
    System.out.println("A-double-param-con");
}
}
class Test {
    public static void main(String[] args) {
        A a = new A();
    }
}
nagoorn@Nagoors-MacBook-Pro COREJAVA-830 % javac Test.java
Test.java:5: error: call to this must be first statement in
constructor
    this(10);
           ^
Test.java:10: error: call to this must be first statement in
constructor
    this(22.22f);
           ^
Test.java:15: error: call to this must be first statement in
constructor
```

```
this(33.3333);
```

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();  
    }  
}
```

nagoorn@Nagoors-MacBook-Pro COREJAVA-830 % javac Test.java
Test.java:9: error: call to this must be first statement in
constructor
 this(10);

Q) Is it possible to access more than one current class constructor by using the “this” keywords from a single current class constructor?

Ans:

No, it is not possible to access more than one current class constructor by using the “this” keyword, because in the constructors “this” statement must be the first statement, if we want to access more than one current class constructors by using this statements then we have to provide more than one this statements, where only one statement is valid and all the remaining statements are invalid.

EX:

```
class A{
    A(){}
        this(10);
        this(22.22f);
        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 {
```

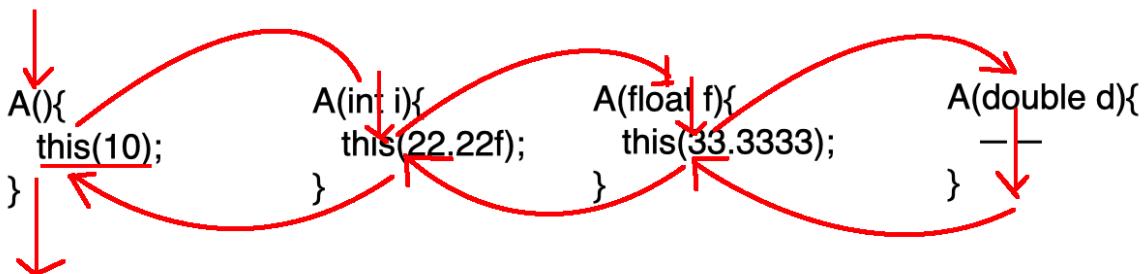
```
public static void main(String[] args) {  
    A a = new A();  
}  
}
```

nagoorn@Nagoors-MacBook-Pro COREJAVA-830 % javac Test.java
Test.java:4: error: call to this must be first statement in
constructor
 this(22.22f);

Q)What is Constructor Chaining?

Ans:

The process of accessing the current class constructors by using “this” keyword in a chain fashion is called Constructor Chaining.



To return the current class Object:

In Java applications, if we want to return the current class objects by using this keyword then we have to use the following syntax.

```
return this;
```

EX:

```
class A{
    A getRef1(){
        A a = new A();
        return a;
    }
    A getRef2(){
        return this;
    }
}
public class Main {
    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
    }
}
```

A@7a81197d

A@5ca881b5

A@24d46ca6

A@4517d9a3

A@7a81197d

A@7a81197d

A@7a81197d

In the above program, for every call of getRef1() method, JVM executes a new keyword, JVM creates a new Object and returns the new object reference value every time, this approach will increase duplicate objects.

In the above program, for every call of getRef2() method , JVM returns this keyword, it will return the Object reference value on which we have accessed getRef2() method, every time it returns the same object reference value , it will reduce the duplicate objects.

Q)What is method chaining?

The process of accessing methods by returning Object reference values is called Method Chaining.

EX:

```
class A{
    A m1(){
        System.out.println("m1-A");
        return this;
    }
    A m2(){
```

```
        System.out.println("m2-A");
        return this;
    }
    A m3(){
        System.out.println("m3-A");
        return this;
    }
}
public class Main {
    public static void main(String[] args) {
        A a = new A();
        a.m1().m2().m3();
    }
}
```

m1-A

m2-A

m3-A

static keyword:

‘static’ is a Java keyword, it is able to improve Shareability in Java applications.

In Java applications, there are four ways to utilize the static keyword.

1. Static variables
2. Static Methods
3. Static Blocks
4. Static import

Static variables:

Static Variable is a java variable, it will be recognized and initialized at the time of loading the respective class bytecode to the memory.

In Java applications, a single copy of the Static Variables data will be provided to all the objects of the same class.

Static Variables are able to provide more shareability when compared with the instance variables.

If we perform modifications on the static variables data then the provided modifications will be applied to all the objects of the respective class.

In Java applications, the last modified values of the static variables are shared to the past objects which we have created already and to the future objects which we are going to create.

In Java applications, static variables must be declared as the class level variables only, not possible to declare as local variables, if we declare any static variable as local variable then the compiler will raise an error.

In Java applications, we are able to access the static variables either by creating objects or by using the class name directly.

Note: In Java applications, it is always suggestible to access the static variables by using the respective class name.

In Java applications, if we access any instance variable by using a reference variable that contains null value then JVM will raise an exception like `java.lang.NullPointerException`, but if we

access any static variable by using a reference variable that contains null value then JVM will not raise any exception like `java.lang.NullPointerException`.

In Java applications, static variables data will be stored inside the Method Area.

EX:

```
class User{
    String uname;
    String uaddr;
    int uage;
    String uemail;
    String umobile;
    public static final int MIN_AGE = 18;
    public static final int MAX_AGE = 25;

    public User(String uname, String uaddr, int uage, String
uemail, String umobile) {
        this.uname = uname;
        this.uaddr = uaddr;
        this.uage = uage;
        this.uemail = uemail;
        this.umobile = umobile;
    }
    public void getUserDetails(){
        System.out.println("User Details");
        System.out.println("-----");
        System.out.println("User Name : "+uname);
        System.out.println("User Address : "+uaddr);
        System.out.println("User Age : "+uage);
        System.out.println("User Email : "+uemail);
        System.out.println("User Mobile Number : "+umobile);
        System.out.println("User Min Age : "+MIN_AGE);
    }
}
```

```
        System.out.println("User Max Age      : "+MAX_AGE);  
    }  
  
}  
public class Main {  
    public static void main(String[] args) {  
        User user1 = new User("Durga", "Hyd", 23,  
"durga@gmail.com", "91-9988776655");  
        user1.getUserDetails();  
        System.out.println();  
  
        User user2 = new User("Venkat", "Chennai", 22,  
"venkat@gmail.com", "91-5566778899");  
        user2.getUserDetails();  
  
    }  
}
```

User Details

```
User Name      : Durga  
User Address   : Hyd  
User Age       : 23  
User Email     : durga@gmail.com  
User Mobile Number : 91-9988776655  
User Min Age   : 18  
User Max Age   : 25
```

User Details

```
User Name      : Venkat  
User Address   : Chennai  
User Age       : 22  
User Email     : venkat@gmail.com
```

User Mobile Number : 91-5566778899

User Min Age : 18

User Max Age : 25

EX:

```
class A{
    int i = 10;
    static int j = 20;
    void m1(){
        //static int k = 30; -----> Error
        System.out.println("m1-A");
    }
}
public class Main {
    public static void main(String[] args) {
        A a = new A();
        System.out.println(a.i);
        System.out.println(a.j);
        //System.out.println(A.i); ---> Error
        System.out.println(A.j);
        a.m1();

        A a1 = null;
        //System.out.println(a1.i); -->
        java.lang.NullPointerException
        System.out.println(a1.j);
    }
}
```

10

20

20

m1-A

20

EX:

```
class A{
    int i = 10;
    static int j = 10;
}
public class Main {
    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(a2.i+" "+a2.j);
        System.out.println(a1.i+" "+a1.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(a3.i+" "+a3.j);
        System.out.println(a2.i+" "+a2.j);
        System.out.println(a1.i+" "+a1.j);
    }
}
```

10	10
11	11
10	11
11	12

```

11      12
10      12
11      13
11      13
11      13

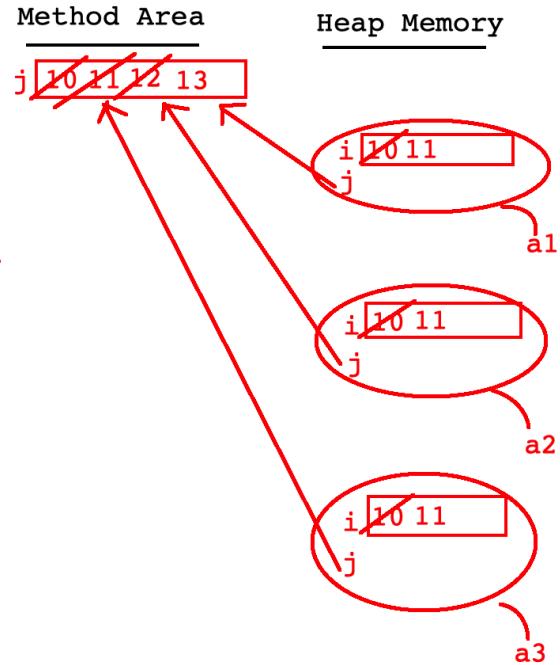
```

```

class A{
    int i = 10;
    static int j = 10;
}

public class Main {
    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(a2.i+" "+a2.j); 11 12
        System.out.println(a1.i+" "+a1.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(a3.i+" "+a3.j); 11 13
        System.out.println(a2.i+" "+a2.j); 11 13
        System.out.println(a1.i+" "+a1.j); 11 13
    }
}

```



Note: In Java applications, for a particular class bytecode will be loaded only once but we can create any number of objects. In this context, static variables are created only once but instance variables are created multiple times individually at each and every object.

Q)What are the differences between static variables and instance variables?

Ans:

1. Static variable is a normal java variable, it will be recognized and initialized at the time of loading the respective class bytecode to the memory.

Instance variable is a normal java variable, it will be recognized and initialized just before accessing the respective class constructor and after loading the respective class bytecode to the memory.

2. To declare static variables we have to use the static keyword explicitly.

To declare instance variables no need to use the static keyword.

3. A single copy of the static variable's data will be shared to all the objects of the same class.

For each and every Object a separate copy of the instance variable's data will be created.

4. If we perform modification on the static variable then that modification is available to all the objects of the same class.

If we perform modification on the instance variable then that modification is available to a particular object, not to all the objects.

5. Static Variables are able to provide more shareability.

Instance variables are able to provide less shareability.

6. Static variables are allowed in both static methods and instance methods.

Instance variables are allowed in only instance methods.

7. In Java applications, we are able to access the static variables either by creating objects or by using the respective class reference variable.

In Java applications, we are able to access instance variables only by creating objects only, not by using the respective class name.

8. In Java applications, it is not possible to get `java.lang.NullPointerException` in the case of accessing a static variable by using a reference variable that contains a null value.

In Java applications, if we access an instance variable by using a reference variable that contains null value then JVM will raise an exception like `java.lang.NullPointerException`.

9. In Java applications, static variables data will be stored inside the method area.

In Java applications, instance variables data will be stored inside the heap memory in the form of Objects.

Static Methods:

Static method is a normal java method with static keyword, it will be recognized and executed the moment when we access it, it may be at load time and it may be at runtime.

Static Methods are able to allow only static members of the current class directly, it will not allow instance members of the current class directly.

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

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

In Java applications, to access static methods either we have to use the class reference variable or we have to use the respective class name.

In Java applications, if we access an instance method by using a reference variable that contains null value then JVM will raise an exception like `java.lang.NullPointerException`, but if we access any static method by using a reference variable that contains null value then JVM will not raise an exception like `java.lang.NullPointerException`.

EX:

```
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);
    A a = new A();
    System.out.println(a.i);
    //System.out.println(this.j); -----> Error
}
void m2(){
    System.out.println("m2-A");
    m1();
    this.m1();
}
public class Main {
    public static void main(String[] args) {
        A a1 = new A();
        a1.m1();
        A.m1();
        a1.m2();

        A a2 = null;
        //a2.m2(); -----> java.lang.NullPointerException
        a2.m1();
    }
}

```

m1-A
20
10
m1-A
20
10

m2-A

m1-A

20

10

m1-A

20

10

m1-A

20

10

m1-A

20

10

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

Ans:

Yes, it is possible to display a line of text on the command prompt without using the main() method, but we have to use a static variable and static method combination.

EX:

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

D:\FullstackJava830\JAVA830>java6.bat

D:\FullstackJava830\JAVA830>set path=C:\Java\jdk1.6.0_45\bin;

```
D:\FullstackJava830\JAVA830>javac -version  
javac 1.6.0_45
```

```
D:\FullstackJava830\JAVA830>javac Test.java
```

```
D:\FullstackJava830\JAVA830>java Test  
Welcome to Durasoft!
```

```
D:\FullstackJava830\JAVA830>java7.bat
```

```
D:\FullstackJava830\JAVA830>set path=C:\Java\jdk1.7.0_80\bin;
```

```
D:\FullstackJava830\JAVA830>javac Test.java
```

```
D:\FullstackJava830\JAVA830>java Test  
Error: Main method not found in class Test, please define the  
main method as:
```

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

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

In JAVA 6 version JVM will load main class bytecode to the memory without checking main() method availability in the main class, here at the time of loading main class bytecode to the memory JVM will recognize and initialize the static variable, as part of the static variable initialization JVM will execute static method, where the static method is able to display the required message on the command prompt, here when JVM encounters System.exit(0) instruction JVM will terminate the program execution.

In the JAVA 7 version, JVM will check whether the main() method is available or not in the main class before loading the main class bytecode to the memory. If the main() method does not exist

in the main class then JVM will not load main class bytecode to the memory and it will provide an error message like below

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

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

If the main() method exists in the main class then JVM will load the main class bytecode to the memory.

Static blocks:

Static Block is a set of instructions which are recognized and executed at the time of loading the respective class bytecode to the memory.

Static Blocks are not allowing instance members of the current class.

In the static blocks if we want to access the instance members then we have to create an object for the respective class and we have to use the generated reference variable.

Inside the static block we are unable to use 'this' keyword.

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

```
        System.out.println(a.i);
        //System.out.println(this.j); -----> Error
    }
}

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

SB-A

20

10

Q) Is it possible to display a line of data on the console without using the `main()` method, static variable and static method?

Ans:

Yes, it is possible to display a line of text on the console without using the `main()` method, static variable and static method, but we have to use a static block.

EX:

```
class Test {
    static{
        System.out.println("Welcome to Durgasoft!");
        System.exit(0);
    }
}
```

D:\FullstackJava830\JAVA830>java6.bat

D:\FullstackJava830\JAVA830>set path=C:\Java\jdk1.6.0_45\bin;

D:\FullstackJava830\JAVA830>javac Test.java

```
D:\FullstackJava830\JAVA830>java Test  
Welcome to Durgasoft!
```

```
D:\FullstackJava830\JAVA830>java7.bat
```

```
D:\FullstackJava830\JAVA830>set path=C:\Java\jdk1.7.0_80\bin;
```

```
D:\FullstackJava830\JAVA830>javac Test.java
```

```
D:\FullstackJava830\JAVA830>java Test  
Error: Main method not found in class Test, please define the  
main method as:
```

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

Note: The above question and Answer are valid up to JAVA 1.6 version, they are invalid from JAVA 1.7 version onwards, because in JAVA 1.6 version JVM will load main class bytecode to the memory without checking main() method availability, but from JAVA 1.7 version onwards , first JVM will check whether main() method exists or not, if the main() method does not exist then JVM will not load main class bytecode to the memory, instead JVM will provide the following error message.

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

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

If the main() method exists in the main class then only JVM will load the main class bytecode to the memory.

If we execute the above program in JAVA 1.6 version, JVM will load main class bytecode to the memory, at the time of loading main class bytecode to the memory JVM will recognize and execute

the provided static block, as part of executing the static block JVM will display the required message on the command prompt, here when JVM encounters `System.exit(0)` JVM will terminate the program execution immediately.

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

Ans:

Yes, it is possible to display a line of text on the command prompt without using the `main()` method, static variable-static method combination and the static block, but we have to use “Static anonymous inner classes of the Object class”.

EX:

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

D:\FullstackJava830\JAVA830>java6.bat

D:\FullstackJava830\JAVA830>set path=C:\Java\jdk1.6.0_45\bin;

D:\FullstackJava830\JAVA830>javac Test.java

D:\FullstackJava830\JAVA830>java Test

Welcome To Durgasoftware!

```
D:\FullstackJava830\JAVA830>java7.bat
```

```
D:\FullstackJava830\JAVA830>set path=C:\Java\jdk1.7.0_80\bin;
```

```
D:\FullstackJava830\JAVA830>javac Test.java
```

```
D:\FullstackJava830\JAVA830>java Test
```

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

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

Note: The above question and Answer are valid up to JAVA 1.6 version, they are invalid from JAVA 1.7 version onwards, because in JAVA 1.6 version JVM will load main class bytecode to the memory without checking main() method availability, but from JAVA 1.7 version onwards , first JVM will check whether main() method exists or not, if the main() method does not exist then JVM will not load main class bytecode to the memory, instead JVM will provide the following error message.

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

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

If the main() method exists in the main class then only JVM will load the main class bytecode to the memory.

If we execute the above program in JAVA 1.6 version, JVM will load main class bytecode to the memory, at the time of loading main class bytecode to the memory JVM will recognize and execute the provided static anonymous inner class of the Object class, as part of executing the static anonymous inner class of the Object

class JVM will display the required message on the command prompt, when JVM encounters System.exit(0) JVM will terminate the program execution immediately.

Static Import:

In Java applications, if we want to use classes and interfaces of a particular package in the present java file , first, we have to make available those classes and interfaces to the present java file, after we will decide whether we want to use them or not.

In Java applications, to make available the classes and interfaces of a particular package to the present java file we have to import the respective packages in the present java file.

EX:

```
import java.io.*;  
import java.util.*;
```

In Java applications, to make available static members of a particular class or interface to the present java file we have to use the “static import”.

Syntax:

```
import static packageName.ClassName.*;
```

It is able to import all static members of the specified class to the present java file.

```
import static packageName.ClassName.memberName;
```

It is able to import only the specified static member of the specified class to the present java file.

In general, in java applications, to access the static members of a particular class we must use either reference variable of the respective class or directly the class name, but if we import any static members of a particular class then we are able to access them directly as like local members without using class name and reference variables.

EX: in `java.lang.Thread` class, there are three static variables to represent the thread priority values like below.

```
public class Thread{
    public static final int MIN_PRIORITY = 1;
    public static final int NORM_PRIORITY = 5;
    public static final int MAX_PRIORITY = 10;
    -----
    -----
}
```

If we import all the above constants through static import then we are able to access them directly without using class name or reference variable.

```
import static java.lang.Thread.*;
-----
-----
System.out.println(MIN_PRIORITY);
System.out.println(NORM_PRIORITY);
System.out.println(MAX_PRIORITY);
```

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);  
}  
}  
  
nagoorn@Nagoors-MacBook-Pro COREJAVA-830 % javac Test.java  
nagoorn@Nagoors-MacBook-Pro COREJAVA-830 % java Test  
1  
5  
10
```

Static Context / Static Flow of Execution:

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

In Java applications, Static Context is represented by the following three elements.

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

Where the Static variables and the Static Blocks are recognized and executed at the time of loading the respective class bytecode to the memory, but the static methods are recognized and executed when we access them.

EX:

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

SB-A

m1-A

EX:

```
class A{
    static int m1(){
        System.out.println("m1-A");
        return 10;
    }
    static{
        System.out.println("SB1-A");
    }
    static int i = m1();
    static int m2(){
        System.out.println("m2-A");
        return 20;
    }
}
```

```
static{
    System.out.println("SB2-A");
}
static int j = m2();
}
public class Main {
    public static void main(String[] args) {
        A a = new A();
    }
}
```

SB1-A

m1-A

SB2-A

m2-A

EX:

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

SB-A

m1-A

EX:

```
class A{
{
    System.out.println("IB-A");
}
static int m1(){
    System.out.println("m1-A");
    return 10;
}
int i = m2();
A(){
    System.out.println("A-Con");
}
static int j = m1();
int m2(){
    System.out.println("m2-A");
    return 20;
}
static{
    System.out.println("SB-A");
}
}

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

m1-A

SB-A

IB-A

m2-A

A-Con

EX:

```
class A{
    static {
        System.out.println("SB-A");
    }
    static int m1(){
        System.out.println("m1-A");
        return 10;
    }
    static int i = m1();

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

    {
        System.out.println("IB-A");
    }
    int m2(){
        System.out.println("m2-A");
        return 20;
    }
    int j = m2();
}

public class Main {
    public static void main(String[] args) {
        A a1 = new A();
        System.out.println();
        A a2 = new A();
    }
}
```

SB-A
m1-A
IB-A
m2-A
A-Con

IB-A
m2-A
A-Con

Class.forName() Method:

Consider the following program,

```
class A{
    static{
        System.out.println("Class Loading....");
    }
    A(){
        System.out.println("Object Creating....");
    }
}
public class Main {
    public static void main(String[] args) {
        A a = new A();
    }
}
```

Class Loading....
Object Creating....

If we execute the above program, JVM will execute the Object Creation statement in the main() method, it will load class bytecode to the memory and it will create an object for the respective class automatically.

As per the application requirements we want to load class bytecode to the memory only, we don't want to create an object, 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");
```

When JVM executes the above instruction, JVM will perform the following actions.

1. JVM will take the provided class from the forName() method.
2. JVM will search for the provided class's .class file at current location, at Java predefined library and at the locations provided by the "classpath" environment variable.
3. If the required .class file is not identified at all the above locations then JVM will raise an exception like java.lang.ClassNotFoundException.
4. If the required .class file exists at either of the above locations then JVM will load its bytecode to the memory.
5. When the class bytecode is loaded automatically JVM will create java.lang.Class object in Heap memory with the metadata of the loaded class, where the class metadata includes name of the class, access modifiers of the class, superclass metadata, implemented interfaces metadata, all

declared variables metadata, all declared methods metadata,
all declared constructors metadata,....

6. After creating java.lang.Class object JVM will return
java.lang.Class object reference value from forName()
method.

EX: Main.java

```
public class Main {  
    public static void main(String[] args) throws Exception {  
        Class.forName("Employee");  
    }  
}
```

Exception in thread "main" java.lang.ClassNotFoundException:
Employee

EX: Main.java

```
class Employee{  
    static{  
        System.out.println("Employee class loading.....");  
    }  
    Employee(){  
        System.out.println("Employee class Object Creating.....");  
    }  
}  
public class Main {  
    public static void main(String[] args) throws Exception {  
        Class cls = Class.forName("Employee");  
        System.out.println("Class Name : " + cls.getName());  
        System.out.println("Super Class Name : " +  
cls.getSuperclass().getName());  
    }  
}
```

Employee class loading.....

Class Name : Employee

Super Class Name : java.lang.Object

EX: In JDBC[Java Database Connectivity], to map Java representations to the Database representations and to map Database representations to the Java representations we will use Drivers. In JDBC, Drivers are provided in the form of predefined classes, in JDBC to activate driver we have to load the driver class bytecode to the memory without creating object, so here to load driver class bytecode to the memory without creating object we have to use Class.forName() method.

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

Class.forName("com.mysql.cj.jdbc.Driver");

newInstance() Method :

In Java applications, after loading class bytecode to the memory by using Class.forName() method, if we want to create an object for a loaded class explicitly then we have to use the following method from java.lang.Class.

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

EX:

```
Class cls = Class.forName("Employee");  
Object obj = cls.newInstance();
```

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

1. JVM will goto the loaded class and search for a constructor which is a 0-arg constructor and a non private constructor.
2. If the required constructor exists in the loaded class then JVM will execute that constructor and return the generated object in the form of java.lang.Object type.
3. If there is no 0-arg constructor, if we have only parameterized constructor in the loaded class then JVM will raise an exception like java.lang.InstantiationException.
4. If we have only private constructors then JVM will raise an exception like java.lang.IllegalAccessException.
5. If we have a constructor which is parameterized and private then JVM will raise an exception like java.lang.InstantiationException, not java.lang.IllegalAccessException.

EX:

```
class Employee{
    static{
        System.out.println("Employee class loading.....");
    }
    Employee(){
        System.out.println("Employee class Object Creating.....");
    }
}
public class Main {
    public static void main(String[] args) throws Exception {
        Class cls = Class.forName("Employee");
        Object obj = cls.newInstance();
    }
}
```

```
Employee class loading.....  
Employee class Object Creating.....
```

EX:

```
class Employee{  
    static{  
        System.out.println("Employee class loading.....");  
    }  
    Employee(int i){  
        System.out.println("Employee class Object Creating.....");  
    }  
}  
  
public class Main {  
    public static void main(String[] args) throws Exception {  
        Class cls = Class.forName("Employee");  
        Object obj = cls.newInstance();  
    }  
}
```

Employee class loading.....

Exception in thread "main" java.lang.InstantiationException:
Employee

EX:

```
class Employee{  
    static{  
        System.out.println("Employee class loading.....");  
    }  
    private Employee(){  
        System.out.println("Employee class Object Creating.....");  
    }  
}  
  
public class Main {  
    public static void main(String[] args) throws Exception {  
        Class cls = Class.forName("Employee");  
        Object obj = cls.newInstance();  
    }  
}
```

```
}
```

```
Employee class loading.....  
Exception in thread "main" java.lang.IllegalAccessException:  
class Main cannot access a member of class Employee with  
modifiers "private"
```

EX:

```
class Employee{  
    static{  
        System.out.println("Employee class loading.....");  
    }  
    private Employee(int i){  
        System.out.println("Employee class Object Creating.....");  
    }  
}  
public class Main {  
    public static void main(String[] args) throws Exception {  
        Class cls = Class.forName("Employee");  
        Object obj = cls.newInstance();  
    }  
}
```

```
Employee class loading.....  
Exception in thread "main" java.lang.InstantiationException:  
Employee
```

EX: In Server side applications, all the server side components will be executed by the servers or containers by following their lifecycle actions, where all the Server side component's lifecycle are having “Loading” and “Instantiation” are two states, here to perform Server side component loading Server or container will use the “Class.forName()” method and to perform

Server side component's Instantiation Server or container will use newInstance() method.

Factory Method:

Factory Method is a Java method, it has to return an object reference value, where the Object reference value may be of the same class or of the different class.

Factory Method is an idea provided by the Factory Method Design pattern.

Note: Design pattern is a system which defines a problem and its solution.

When Developers find a design problem in their application development they have to select the right design pattern and they have to get the solution from the right design pattern.

EX:

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

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

A-Con
m1-A

There are two types of Factory methods.

1. Static Factory Methods:

If any static method returns an Object reference value then that static method is called a Static Factory method.

EX: Class cls = Class.forName("----");
NumberFormat nf = NumberFormat.getInstance();
DateFormat df = DateFormat.getDateInstance();
Connection con = DriverManager.getConnection("----");

2. Instance Factory Methods:

If any instance method returns an object reference value then that instance method is called an Instance Factory Method.

EX: Majority of the String class methods are instance factory methods.

```
String str = new String("Durga");
String str1 = str.concat("soft");
String str2 = str.trim();
String str3 = str.toLowerCase();
String str4 = str.toUpperCase();
-----
-----
```

Singleton class:

If any class allows us to create only one object for it then that class is called a singleton class.

If we want to make a class as a singleton class then we have to use the following steps.

1. Declare an user defined class.
2. Declare a private constructor inside the class.
3. Declare a static factory method with the following implementation.
 - a. Check whether any object exists or not for the current class.
 - b. If no Object exists for the current class then create a new object and return it.
 - c. If an object exists for the current class already then return that existing object without creating a new object.
4. In the Main class, in the main() method access the static factory method multiple times and check the return values.
5. If we get the same reference value every time then the class is Singleton class otherwise the class is not a singleton class.

EX:

```
class A{  
    private static A a = null;// a111  
    private A(){  
        System.out.println("A-Con");  
    }  
    static A getInstance(){  
        /*  
         * 1. Check whether an Object is created already or not  
        */  
    }  
}
```

2. If an Object created already then return that object without creating new Object
3. If no Object is created so far then create new Object and return it.

```

        */
/*if(a == null){
    a = new A();
    return a;
}else{
    return a;
}*/
if(a == null){
    a = new A();
}
return a;
}

}

public class Main {
    public static void main(String[] args){
        A a1 = A.getInstance(); // a1=A@a111
        A a2 = A.getInstance(); // a2=A@a111
        A a3 = A.getInstance(); // a3=A@a111

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

```

A-Con

A@7a81197d

A@7a81197d

A@7a81197d

Optimized Code:

```
class A{
    private static A a = null;
    static{
        a = new A();
    }
    private A(){
        System.out.println("A-Con");
    }
    static A getInstance(){
        return a;
    }
}
public class Main {
    public static void main(String[] args){
        A a1 = A.getInstance(); // a1=A@a111
        A a2 = A.getInstance(); // a2=A@a111
        A a3 = A.getInstance(); // a3=A@a111

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

A-Con

A@7a81197d

A@7a81197d

A@7a81197d

More optimized Code:

```
class A{
    private static A a = new A();

    private A(){
        System.out.println("A-Con");
    }
    public static A getInstance(){
        return a;
    }
}

public class Main {
    public static void main(String[] args){
        A a1 = A.getInstance(); // a1=A@a111
        A a2 = A.getInstance(); // a2=A@a111
        A a3 = A.getInstance(); // a3=A@a111

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

A-Con

A@7a81197d

A@7a81197d

A@7a81197d

In general, in MVC based web applications, we have to use a class as a controller, as per the MVC rules and regulations we must provide only one controller component per application, that is we must create only one object for the controller class, it implies the controller class must be a Singleton class.

Note: In MVC based applications, we will use classes for Controller, Service, Repository , here all those classes must be Singleton classes, they must allow only one object.

Note: Singleton class is an idea provided by Singleton class Design pattern.

'final' keyword:

'final' is a Java keyword, it can be used to declare the constant expressions in Java applications.

In Java applications, there are three ways to utilize the final keyword.

1. final variables
2. final Methods
3. final classes

final variable:

final variable is a Java variable, it will not allow modifications on its value.

In Java applications, final variables do not allow the re-assigments.

EX:

```
final int i = 10;  
i = i + 10; -----> Error  
i = 20; -----> Error
```

In Java applications, we must not declare loop variables as final as they need increment / decrement operations over the loop variables in order to perform iterations.

EX:

```
class Test{  
    public static void main(String[] args) {  
  
        for(final int i = 0; i < 10; i++){  
            System.out.println(i);  
        }  
    }  
}
```

```
nagoorn@Nagoors-MacBook-Pro COREJAVA-7 % javac Test.java  
Test.java:4: error: cannot assign a value to final variable i  
    for(final int i = 0; i < 10; i++){
```

EX: In general, in bank applications, when we create an account with the account number then it is not possible to modify the account number, but we can modify the account details like account holder name, account holder address details,... other than account number, in fix a particular account number for a particular customer throughout the bank application we have to declare the account number variable as final.

final Methods:

These are the java methods, they will not allow changes in their functionality.

In Java applications, we are able to change the method functionalities by performing method overriding. If we declare a method as a final then it will not allow method overriding.

In Method Overriding, superclass methods must not be declared as final.

EX:

```
class A{
    final void m1(){
        System.out.println("X-Functionality");
    }
}
class B extends A{
    void m1(){
        System.out.println("Y-Functionality");
    }
}
public class Main {
    public static void main(String[] args){
        A a = new B();
        a.m1();
    }
}
```

Status: Compilation Error.

Note: In method Overriding Superclass method must not be declared as final, but we can declare subclass method as final.

final classes:

Final class is a java class, it will not allow subclasses.

In inheritance, superclasses never declared as final.

EX:

```
final class A{
    void m1(){
        System.out.println("X-Functionality");
    }
    void m2(){
        System.out.println("Y-Functionality");
    }
}
class B extends A{
```

}

Status: Compilation Error.

In Inheritance, super class never be final, but subclass may be or may not be final.

In Java applications, to declare constant variables Java has provided a convention like to declare constant variables with “public static final”.

Where the purpose of the “public” is to make available constant variables throughout the application in order to access.

Where the purpose of the “static” is to access the constant variables by using class names without creating objects and to share its value to multiple objects of the same class.

Where the purpose of the “final” is to fix a particular value to the variable throughout the application, that is not to allow modifications on its value.

EX: In Thread class, to represent Thread priority values Java has provided the following constants.

```
class Thread{  
    public static final int MIN_PRIORITY = 1;  
    public static final int NORM_PRIORITY = 5;  
    public static final int MAX_PRIORITY = 10;  
}
```

EX:

```
public class Main {  
    public static void main(String[] args){  
        System.out.println(Thread.MIN_PRIORITY);  
        System.out.println(Thread.NORM_PRIORITY);  
        System.out.println(Thread.MAX_PRIORITY);  
    }  
}
```

1
5
10

EX:

```
class WeekDays{  
    public static final String DAY1_OF_WEEK = "MONDAY";  
    public static final String DAY2_OF_WEEK = "TUESDAY";  
    public static final String DAY3_OF_WEEK = "WEDNESDAY";  
    public static final String DAY4_OF_WEEK = "THURSDAY";  
    public static final String DAY5_OF_WEEK = "FRIDAY";  
    public static final String DAY6_OF_WEEK = "SATURDAY";
```

```

    public static final String DAY7_OF_WEEK = "SUNDAY";
}
public class Main {
    public static void main(String[] args){
        System.out.println(WeekDays.DAY1_OF_WEEK);
        System.out.println(WeekDays.DAY2_OF_WEEK);
        System.out.println(WeekDays.DAY3_OF_WEEK);
        System.out.println(WeekDays.DAY4_OF_WEEK);
        System.out.println(WeekDays.DAY5_OF_WEEK);
        System.out.println(WeekDays.DAY6_OF_WEEK);
        System.out.println(WeekDays.DAY7_OF_WEEK);
    }
}

```

MONDAY
 TUESDAY
 WEDNESDAY
 THURSDAY
 FRIDAY
 SATURDAY
 SUNDAY

EX:

```

class UserStatus{
    public static final String GREEN = "User Available";
    public static final String RED = "User Busy";
    public static final String ORANGE = "User Idle";
}
public class Main {
    public static void main(String[] args){
        System.out.println(UserStatus.GREEN);
        System.out.println(UserStatus.RED);
        System.out.println(UserStatus.ORANGE);
    }
}

```

```
 }  
}
```

User Available

User Busy

User Idle

If we use the above convention to declare the constant variables then we are able to get the following problems.

1. For each and every constant variable we have to provide the “public static final” explicitly.
2. This approach allows different data types to declare the constant variables , it will reduce the typedness in java applications and it will provide type unsafe operations in java applications.
3. In this approach, if we access the constant variable then we are able to get the value of the constant variable, here the value of the constant variable may or may not represent its actual intention of the constant variable.

To overcome all the above problems JAVA has provided an alternative in the form of “enums”.

Inside the enums,

1. All the constant variables are by default “public static final”, no need to declare explicitly.
2. All the constant variables are by default the same enum type, no need to provide data types explicitly, this approach will increase typedness in java applications and it is able to allow type safe operations in java applications.

3. All the constant variables are “Named Constants”, when we access a constant variable then it will provide its name instead of its value, where the name of the constant variable will represent the intention of the constant variable.

Syntax:

```
[Access Modifier] enum Name{  
    ----List of constant variables----  
}
```

EX:

```
enum UserStatus{  
    AVAILABLE, BUSY , IDLE;  
}  
public class Main {  
    public static void main(String[] args){  
        System.out.println(UserStatus.AVAILABLE);  
        System.out.println(UserStatus.BUSY);  
        System.out.println(UserStatus.IDLE);  
  
    }  
}
```

```
AVAILABLE  
BUSY  
IDLE
```

When we compile an enum then the compiler will convert the enum as a final class like below.

```
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 {};  
}
```

From the above translations,

1. Every enum is a final class, so inheritance is not possible between enums.
2. Every enum must be a subclass of java.lang.Enum.
3. Every constant variable is converted to “public static final”.
4. Every constant variable must have the same enum type.

Q) In general, in Java applications we will use enums to declare constant variables. In this context, is it possible to provide normal variables , normal methods, constructors inside the enum like normal classes?

Ans:

Yes, it is possible to provide normal variables, methods, constructors along with the constant variables inside the enum like classes.

EX:

```
enum Apple{  
    A(500),B(300),C(100);  
    private int price;  
    Apple(int price){  
        this.price = price;
```

```

    }
    public int getPrice(){
        return price;
    }
}
public class Main {
    public static void main(String[] args) {
        System.out.println("A-Grade Apple : " +
Apple.A.getPrice());
        System.out.println("B-Grade Apple : " +
Apple.B.getPrice());
        System.out.println("C-Grade Apple : " +
Apple.C.getPrice());
    }
}

```

If we compile the Apple enum then the compiler will provide the following translated class.

```

final class Apple extends 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);
    private int price;
    Apple(int price){
        this.price = price;
    }

    public int getPrice(){
        return price;
    }
}

```

Q) Write a Java program to represent Notebook and its details like number of pages and price value by using enum?

```
enum NoteBook{
    A(300, 150),B(200, 100),C(100, 50);
    private int pages;
    private int price;
    NoteBook(int pages, int price){
        this.pages = pages;
        this.price = price;
    }

    public int getPages() {
        return pages;
    }

    public int getPrice() {
        return price;
    }
}

public class Main {
    public static void main(String[] args) {
        System.out.println("A-Grade Notebook : Pages :" +
NoteBook.A.getPages()+" Price : " + NoteBook.A.getPrice());
        System.out.println("B-Grade Notebook : Pages :" +
NoteBook.B.getPages()+" Price : " + NoteBook.B.getPrice());
        System.out.println("C-Grade Notebook : Pages :" +
NoteBook.C.getPages()+" Price : " + NoteBook.C.getPrice());
    }
}
```

```
A-Grade Note Book : Pages :300      Price : 150
B-Grade Note Book : Pages :200      Price : 100
C-Grade Note book : Pages :100      Price : 50
```

Importance of the main() method:

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

Ans:

The main purpose of the main() method in java applications is

1. main() method is an entry point for the JVM to enter into the application and to execute the application.
2. In the java applications we need a method to hold the application logic which must be executed by the JVM automatically, so we need the main() method.
3. To define starting point and ending point to the application execution we need the main() method.

Syntax:

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

Note: main() method is not a predefined method, it is not an user defined method, it is a conventional method with the fixed prototype and with the user defined implementation.

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

Ans:

The main purpose of declaring main() method with public is to bring main() method scope to the JVM in order to access it.

If we declare the main() method as private then the private main() method will have scope up to the present class only; it is not available to the JVM as JVM is available in Java software under C drive.

If we declare the main() method as <default> then the default main() method will have scope up to the present package, but not to the JVM as JVM is available in Java software under C drive.

If we declare the main() method as protected then the protected main() method will have scope up to the present package and the child classes existed in other packages, but not to the JVM as JVM is available in Java software under C drive.

If we declare the main() method as public then the public main() method will have scope throughout the application or throughout the System including JVM.

```
C:\Java\jdk17\bin
package com.sun.jvm;
public class JVM{
    public static void main(String[] args){
        Test.main(--);
    }
}
```

```
D:\Fullstackjava830\corejava830\Test.java
package com.durgasoft.test;
public class Test{
    public static void main(String[] args){
        -----
    }
}
```

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

Ans:

In Java applications, if we declare main() method without public then the compiler will not raise any error, because the compiler has treated the main() method as a normal java method, in java applications we can declare normal java methods without public, but JVM will provide the following messages.

JAVA 6: main method not public.

JAVA 7: Error: main method not found in class Test, please define the main method as :

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

EX:

```
class Test {  
    private static void main(String[] args) {  
        System.out.println("Hello World!");  
    }  
}
```

D:\FullstackJava830\JAVA830>set path=C:\Java\jdk1.6.0_45\bin;

D:\FullstackJava830\JAVA830>javac Test.java

D:\FullstackJava830\JAVA830>java Test
Main method not public.

D:\FullstackJava830\JAVA830>java7.bat

D:\FullstackJava830\JAVA830>set path=C:\Java\jdk1.7.0_80\bin;

```
D:\FullstackJava830\JAVA830>javac Test.java
```

```
D:\FullstackJava830\JAVA830>java Test
```

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

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

```
D:\FullstackJava830\JAVA830>
```

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

Ans:

In Java applications, to start application execution JVM must access main() method, here to access main() method JVM must use class name , because JVM was implemented by SUN Microsystems in such a way that to access main() method by using class name only.

In JAVA / J2EE applications, only static methods are possible to access by using class names.

As per the internal implementation of JVM we must declare the main() method as “static”.

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

Ans:

In Java applications, if we declare main() method without the static keyword then the compiler will not raise any error,

because compiler will treat the `main()` method as normal java method,in java applications it is possible to declare methods without the `static` keyword, but JVM will provide the following messages.

JAVA 6 : `java.lang.NoSuchMethodError: main`
JAVA 7 : Error: Main method is not static in class Test, please define the main method as :
 `public static void main(String[] args)`

EX:

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

D:\FullstackJava830\JAVA830>`java6.bat`

D:\FullstackJava830\JAVA830>`set path=C:\Java\jdk1.6.0_45\bin;`

D:\FullstackJava830\JAVA830>`javac Test.java`

D:\FullstackJava830\JAVA830>`java Test`
Exception in thread "main" `java.lang.NoSuchMethodError: main`

D:\FullstackJava830\JAVA830>`java7.bat`

D:\FullstackJava830\JAVA830>`set path=C:\Java\jdk1.7.0_80\bin;`

D:\FullstackJava830\JAVA830>`javac Test.java`

D:\FullstackJava830\JAVA830>`java Test`

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

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

D:\FullstackJava830\JAVA830>

Q)What is the requirement to declare the main() method with the “void” return type?

Ans:

In Java applications, as per the java conventions, we have to provide the complete application logic inside the main() method, that is we must start application logic at starting point of the main() method and we must terminate the application logic at ending point of the main() method, we must not extend the application logic from main() method, in this context, 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 then we must provide the “void” as return type to the main() method.

Q)In java applications, if we declare main() method without the “void” return type then what will be the result in Java application?

Ans:

If we declare the main() method without the “void” return type then the compiler will not raise any error , because the compiler will treat the main() method as a normal java method, in java applications it is possible to declare methods without void return type, but JVM will provide the following messages.

```
JAVA 6: java.lang.NoSuchMethodError:main  
JAVA 7: Error: main method must return a value of type void,  
please define the main method as : public static void  
main(String[] args)
```

EX:

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

```
D:\FullstackJava830\JAVA830>java6.bat
```

```
D:\FullstackJava830\JAVA830>set path=C:\Java\jdk1.6.0_45\bin;
```

```
D:\FullstackJava830\JAVA830>javac Test.java
```

```
D:\FullstackJava830\JAVA830>java Test  
Exception in thread "main" java.lang.NoSuchMethodError: main
```

```
D:\FullstackJava830\JAVA830>java7.bat
```

```
D:\FullstackJava830\JAVA830>set path=C:\Java\jdk1.7.0_80\bin;
```

```
D:\FullstackJava830\JAVA830>javac Test.java
```

```
D:\FullstackJava830\JAVA830>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:\FullstackJava830\JAVA830>

Note: the name “main” is provided by the Java Creators to reflect the importance of the main() method.

main() method parameters:

Q) What is the requirement of the main() method parameters?

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 program at the time of writing the program then the provided input is called static input.

EX:

```
class A{  
    int fval = 10;  
    int sval = 20;  
    class add(){  
        int result = fval + sval;  
    }  
}
```

Dynamic Input:

If we provide input data to the program at runtime then the provided input data is called Dynamic input.

EX:

```
D:\java830>javac Add.java
```

```
D:\java830>java Add
```

```
First Value : 10
```

```
Second Value : 20
```

```
Add result : 30
```

Command Line Input / Command Line Arguments:

If we provide input data to the program by providing values along with the “java” command on command prompt then the provided input is called command line input.

EX:

```
D:\java830>javac Add.java
```

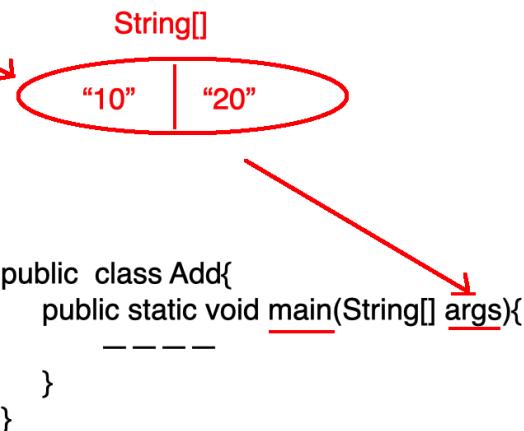
```
D:\java830>java Add 10 20
```

```
Add Result : 30
```

In java applications, if we provide command line arguments then JVM will read the command line arguments, JVM will store them in the form of `String[]` and JVM will access the `main()` method by passing the generated `String[]` parameter.

As per the above implementation, the purpose of the `main()` method parameter is to hold the command line arguments which are provided along with the java command.

```
D:\java830>javac Add.java  
D:\java830>java Add 10 20
```



EX:

```
class Test {  
    public static void main(String[] args) {  
        for(String str: args){  
            System.out.println(str);  
        }  
    }  
}  
nagoorn@Nagoors-MacBook-Pro COREJAVA-830 % javac Test.java  
nagoorn@Nagoors-MacBook-Pro COREJAVA-830 % java Test  
nagoorn@Nagoors-MacBook-Pro COREJAVA-830 % java Test 10 "abc" 'A'  
50000L false  
10  
abc  
A  
50000L  
false
```

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

Ans:

IN general, command line input may be varied from one application to another application or from one developer to another developer, in this context, even if we provided different types of command line input still the main() method parameter must store all the types of command line inputs. In JAVA / J2EE applications only String data type is capable of representing all the types of data, so we need String data type as a parameter to the main() method.

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

Ans:

In general, from application to application, Developer to developer we may provide a variable number of command line inputs, even if we provide a variable number of command line input still the main() method parameter must store all the number of command line input.

Note: IN JAVA/J2EE technologies arrays are used to represent more than one element.

Due to the above reason, the main() method must have an array as a parameter.

Q) In Java applications, if we provide the main() method without the String[] parameter then what will result in Java applications?

Ans:

If we provide main() method without the String[] parameters then the compiler will not raise any error, because the compiler will treat main() method as a normal java method, in java applications is it possible to declare methods without the String[] parameters, but JVM will provide the following messages.

JAVA 6: java.lang.NoSuchMethodError: main
JAVA 7: Error: main method is not found in class Test, please define the main method as: public static void main(String[] args)

EX:

```
class Test {  
    public static void main(int[] args) {  
        System.out.println("Hello World!");  
        return 10;  
    }  
}
```

D:\FullstackJava830\JAVA830>java6.bat

D:\FullstackJava830\JAVA830>set path=C:\Java\jdk1.6.0_45\bin;

D:\FullstackJava830\JAVA830>javac Test.java

D:\FullstackJava830\JAVA830>java Test
Exception in thread "main" java.lang.NoSuchMethodError: main

D:\FullstackJava830\JAVA830>java7.bat

```
D:\FullstackJava830\JAVA830>set path=C:\Java\jdk1.7.0_80\bin;
```

```
D:\FullstackJava830\JAVA830>javac Test.java
```

```
D:\FullstackJava830\JAVA830>java Test
```

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

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

Q) Find the valid syntaxes of 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(String args[]) ----->Valid
4. public static void main(String []args)----->Valid
5. public static void main(String ... args)----->Valid
6. public static void main(string[] args)----->Invalid
7. public static void main(String[][] args)-----> Invalid
8. public static void main(String args1, String args2)->Invalid
9. public static void Main(String[] args)-----> Invalid
10. public static int main(String[] args)-----> Invalid
11. public static final void main(String[] args)-->Valid
12. public void main(String[] args) -----> Invalid
13. static void main(String[] args) -----> Invalid
14. static public void main(String[] args) ----> valid

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

Ans:

Yes, it is possible to provide more than one `main()` method in a single application, but not in a single class, we can provide more than one `main()` method in multiple classes. In this context, the class whose name we provided along with the java command that class `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:\JAVA7>javac Test.java

D:\JAVA7>java A
main()-A

D:\JAVA7>java B
main()-B

D:\JAVA7>java C

`main()-C`

In the above context, it is possible to access one class `main()` method from another class `main()` method just by providing a `String[]` parameter explicitly and by using class name.

EX:

`Test.java`

```
class A {
    public static void main(String[] args) {
        System.out.println("main()-A");
        String[] str = {};
        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:\JAVA7>`javac Test.java`

D:\JAVA7>`java A`

```
main()-A
main()-B
```

main() - C

Q) Is it possible to overload the main() method?

Ans

Yes, it is possible to overload the main() method, we can provide more than one main() method with different parameter lists, in this context JVM will execute only the main() method that has a String[] parameter, but it is not possible to override the main() method, because in java static method overloading is possible, static methods overriding is not possible.

EX:

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

D:\JAVA7>javac Test.java

D:\JAVA7>java Test
String[]-Param-main()

Relationships in java:

In java application development, to provide effective implementation in order to reduce execution time and memory then we have to define relationships between classes.

In general, in java applications, we may provide the application logic over multiple classes, here to execute the applications we need communication between classes, for this we have to use Relationships between classes.

There are three types of relationships between the classes.

1. HAS-A Relationship
2. IS-A Relationship
3. USES-A Relationship

Q)What is the difference between HAS-A relation and IS-A relation?

Ans:

HAS-A Relationship able to define associations between entities, where association is able to provide code reusability, communication between classes and the data navigation between classes.

IS-A relationship is able to provide inheritance between entities in order to provide code reusability, communication between classes, Data navigation between classes.

Associations in Java:

It is a relation between entities where one or multiple instances of an entity should be matched with the one or multiple instances of another entity.

There are four types of associations in Java.

1. One-To-One Association.
2. One-To-Many Association.
3. Many-To-One Association.
4. Many-To-Many Association.

In Java applications, associations are achieved by using dependency injection design pattern.

Dependency Injection: Injecting Dependent objects in an entity is called Dependency Injection.

There are two types of Dependency Injection.

1. Constructor Dependency Injection.
2. Setter Method Dependency Injection.

Constructor Dependency Injection:

Injecting a Dependent Object through a constructor is called Constructor Dependency Injection.

Note: In java applications, Constructors are executed at the time of creating objects, so the Constructor Dependency injection will be performed at the time of creating Container class object only.

EX:

```
class Account{
    ----
}

class Employee{
    private Account account;
    Employee(Account account){
        this.account = account;
    }
    ----

}

class Test{
    public static void main(String[] args){
        Account account = new Account();
        Employee emp = new Employee(account);
        ----
    }
}
```

Setter Method Dependency Injection:

Injecting the dependent Object in an entity class through a
setter method is called Setter method Dependency injection.

EX:

```
class ReviewAndRating{
    ----
}

class Movie{
    ----
    private ReviewAndRating reviewAndRating;
    ----
```

```

public void setReviewAndRating(ReviewAndRating
                               reviewAndRating){
    this.reviewAndRating = reviewAndRating;
}
}

class Test{
    public static void main(String[] args){
        ReviewAndRating rr = new ReviewAndRating();
        Movie m = new Movie();
        m.setReviewAndRating(rr);
    }
}

```

One-To-One Association:

It is a relation between entities, where one instance of an entity should be mapped with exactly one instance of another entity.

Example on Constructor Dependency Injection

Every Employee must have exactly one individual Bank Account.

EX:

Account.java

```

public class Account {

    private String accountNumber;
    private String accountHolderName;
    private String accountType;
    private long accountBalance;

    public Account(String accountNumber, String accountHolderName,
String accountType, long accountBalance) {

```

```
        this.accountNumber = accountNumber;
        this.accountHolderName = accountHolderName;
        this.accountType = accountType;
        this.accountBalance = accountBalance;
    }

    public String getAccountNumber() {
        return accountNumber;
    }

    public String getAccountHolderName() {
        return accountHolderName;
    }

    public String getAccountType() {
        return accountType;
    }

    public long getAccountBalance() {
        return accountBalance;
    }
}

Employee.java
public class Employee {
    private int eno;
    private String ename;
    private float esal;
    private String eaddr;
    private 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.getAccountNumber());
        System.out.println("Account Holder Name : " +
account.getAccountHolderName());
        System.out.println("Account Type        : " +
account.getAccountType());
        System.out.println("Account Balance     : " +
account.getAccountBalance());
    }

}

Main.java
```

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

        Account account = new Account("abc123", "Durga",
"Savings", 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

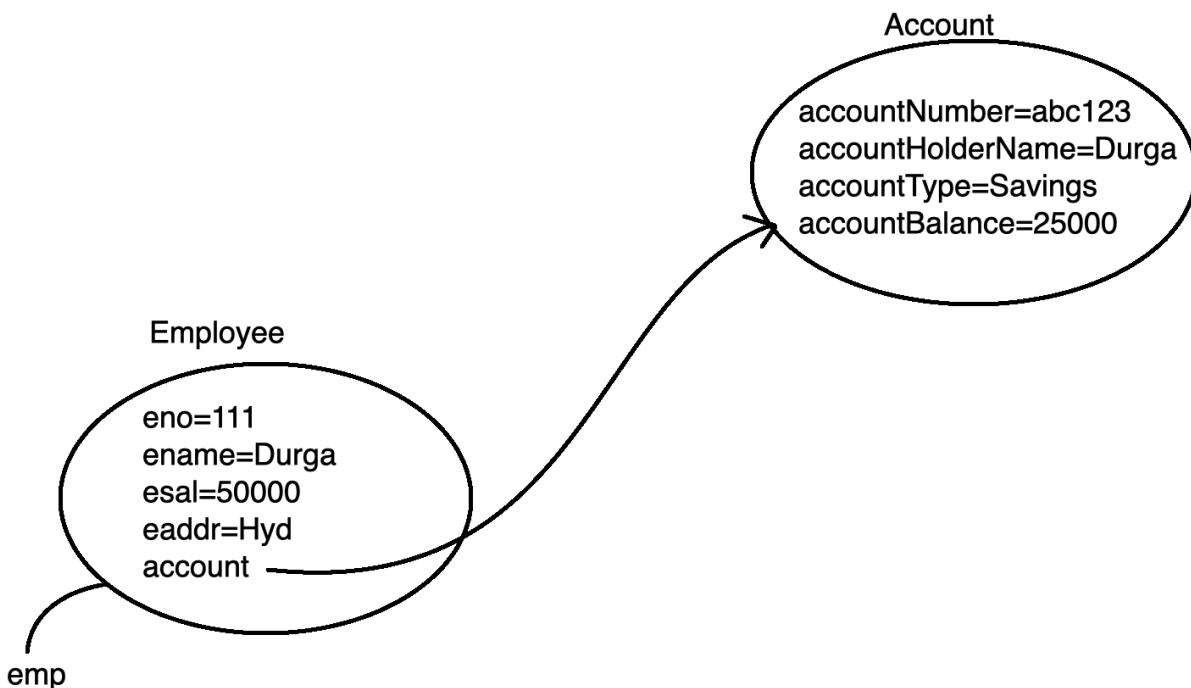
```

Account Details

```

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

```



Example On Setter Method Dependency Injection:

Every person has an individual DrivingLicence.

DrivingLicence.java

```
public class Person {

    private String aadharNo;
    private String personName;
    private String personDob;
    private String personEmailId;
    private String personMobileNo;
    private DrivingLicence drivingLicence;

    public String getAadharNo() {
        return aadharNo;
    }

    public void setAadharNo(String aadharNo) {
        this.aadharNo = aadharNo;
    }

    public String getPersonName() {
        return personName;
    }

    public void setPersonName(String personName) {
        this.personName = personName;
    }

    public String getPersonDob() {
        return personDob;
    }
}
```

```
public void setPersonDob(String personDob) {
    this.personDob = personDob;
}

public String getPersonEmailId() {
    return personEmailId;
}

public void setPersonEmailId(String personEmailId) {
    this.personEmailId = personEmailId;
}

public String getPersonMobileNo() {
    return personMobileNo;
}

public void setPersonMobileNo(String personMobileNo) {
    this.personMobileNo = personMobileNo;
}

public DrivingLicence getDrivingLicence() {
    return drivingLicence;
}

public void setDrivingLicence(DrivingLicence drivingLicence) {
    this.drivingLicence = drivingLicence;
}

public void getPersonDetails() {
    System.out.println("Person Details");
    System.out.println("-----");
    System.out.println("Aadhar Number : " + aadharNo);
    System.out.println("Person Name : " +
personName);
```

```

        System.out.println("Person DOB : " + personDob);
        System.out.println("Person Email ID : " +
personEmailId);
        System.out.println("Person Mobile No : " +
personMobileNo);

        System.out.println("Driving Licence Details");
        System.out.println("-----");
        System.out.println("Licence Number : "+drivingLicence.getLicenceNo());
        System.out.println("Licence Holder Name : "+drivingLicence.getLicenceHolderName());
        System.out.println("Licence Holder Address : "+drivingLicence.getLicenceHolderAddress());
        System.out.println("Licence Type : "+drivingLicence.getLicenceType());
        System.out.println("Licence Date : "+drivingLicence.getLicenceDate());
        System.out.println("Licence Expr Date : "+drivingLicence.getLicenceExprDate());

    }

}

```

Person.java

```

public class Person {

    private String aadharNo;
    private String personName;
    private String personDob;
    private String personEmailId;
    private String personMobileNo;
    private DrivingLicence drivingLicence;

```

```
public String getAadharNo() {
    return aadharNo;
}

public void setAadharNo(String aadharNo) {
    this.aadharNo = aadharNo;
}

public String getPersonName() {
    return personName;
}

public void setPersonName(String personName) {
    this.personName = personName;
}

public String getPersonDob() {
    return personDob;
}

public void setPersonDob(String personDob) {
    this.personDob = personDob;
}

public String getPersonEmailId() {
    return personEmailId;
}

public void setPersonEmailId(String personEmailId) {
    this.personEmailId = personEmailId;
}

public String getPersonMobileNo() {
```

```
        return personMobileNo;
    }

    public void setPersonMobileNo(String personMobileNo) {
        this.personMobileNo = personMobileNo;
    }

    public DrivingLicence getDrivingLicence() {
        return drivingLicence;
    }

    public void setDrivingLicence(DrivingLicence drivingLicence) {
        this.drivingLicence = drivingLicence;
    }

    public void getPersonDetails() {
        System.out.println("Person Details");
        System.out.println("-----");
        System.out.println("Aadhar Number : " + aadharNo);
        System.out.println("Person Name : " +
personName);
        System.out.println("Person DOB : " + personDob);
        System.out.println("Person Email ID : " +
personEmailId);
        System.out.println("Person Mobile No : " +
personMobileNo);

        System.out.println("Driving Licence Details");
        System.out.println("-----");
        System.out.println("Licence Number : " +
+drivingLicence.getLicenceNo());
        System.out.println("Licence Holder Name : " +
+drivingLicence.getLicenceHolderName());
    }
}
```

```
        System.out.println("Licence Type      : "+drivingLicence.getLicenceType());
        System.out.println("Licence Date       : "+drivingLicence.getLicenceDate());
        System.out.println("Licence Expr Date : "+drivingLicence.getLicenceExprDate());

    }
}
```

Main.java

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

        DrivingLicence drivingLicence = new DrivingLicence();
        drivingLicence.setLicenceNo("abc123");
        drivingLicence.setLicenceHolderName("Durga");
        drivingLicence.setLicenceHolderAddress("S R Nagar,
Hyderabad");
        drivingLicence.setLicenceType("2-wheeler");
        drivingLicence.setLicenceDate("13-11-2024");
        drivingLicence.setLicenceExprDate("13-11-2034");

        Person person = new Person();
        person.setAadharNo("chyuts12345defg");
        person.setPersonName("Durga");
        person.setPersonDob("10-10-1996");
        person.setPersonEmailId("durga123@gmail.com");
        person.setPersonMobileNo("91-9988776655");
        person.setDrivingLicence(drivingLicence);

        person.getPersonDetails();
    }
}
```

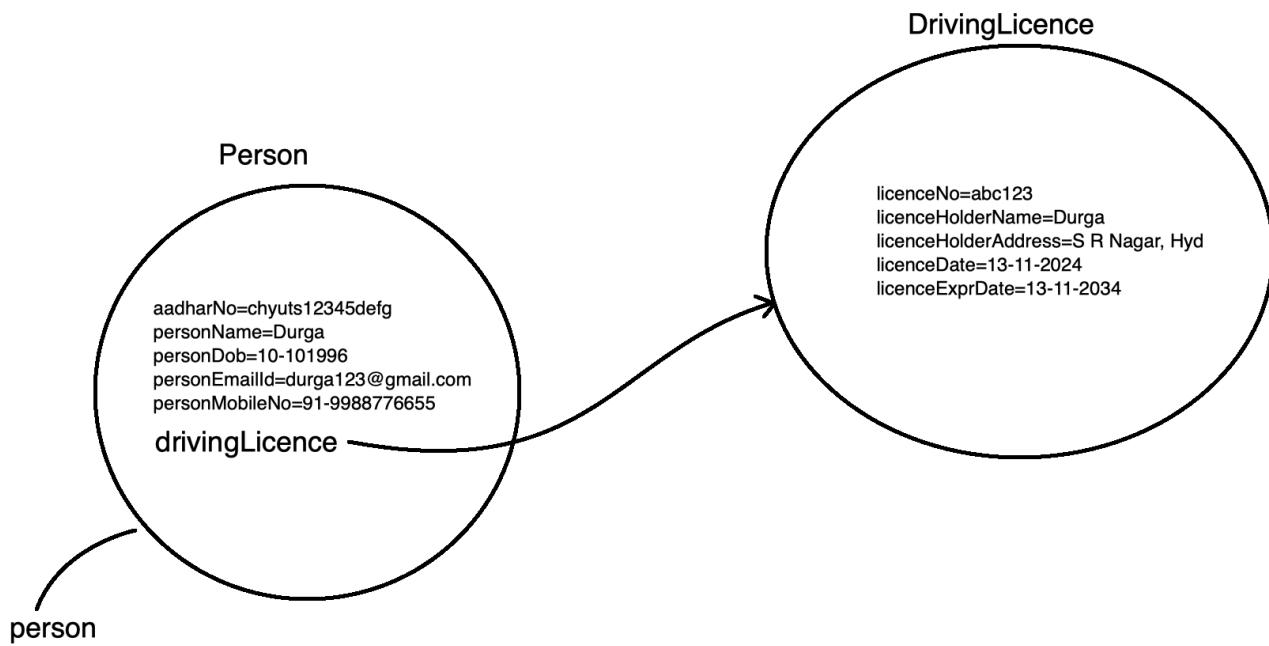
}

Person Details

Aadhar Number : chyuts12345defg
Person Name : Durga
Person DOB : 10-10-1996
Person Email ID : durga123@gmail.com
Person Mobile No : 91-9988776655

Driving Licence Details

Licence Number : abc123
Licence Holder Name : Durga
Licence Holder Address : S R Nagar, Hyderabad
Licence Type : 2-wheeler
Licence Date : 13-11-2024
Licence Expr Date : 13-11-2034



One-To-Many Association:

It is a relation between entities, where one instance of an entity should be mapped with multiple instances of another entity.

Example on Constructor Dependency Injection:

A movie can have multiple Actors.

EX:

`Actor.java`

```

public class Actor {

    private String name;
    private String role;
    private String address;
    private int noOfMovies;
  
```

```
public Actor(String name, String role, String address, int
noOfMovies) {
    this.name = name;
    this.role = role;
    this.address = address;
    this.noOfMovies = noOfMovies;
}

public String getName() {
    return name;
}

public String getRole() {
    return role;
}

public String getAddress() {
    return address;
}

public int getNoOfMovies() {
    return noOfMovies;
}
}
```

```
Movie.java
public class Movie {
    private String movieName;
    private String releaseDate;
    private String directorName;
    private String producerName;
    private Actor[] actors;
```

```

public Movie(String movieName, String releaseDate, String
directorName, String producerName, Actor[] actors) {
    this.movieName = movieName;
    this.releaseDate = releaseDate;
    this.directorName = directorName;
    this.producerName = producerName;
    this.actors = actors;
}

public void getMovieDetails(){
    System.out.println("Movie Details");
    System.out.println("-----");
    System.out.println("Movie Name : " + movieName);
    System.out.println("Release Date : " +
releaseDate);
    System.out.println("Director Name : " +
directorName);
    System.out.println("Producer Name : " +
producerName);
    System.out.println("Name\tRole\tAddress\tMovies");
    System.out.println("-----");
    for(Actor actor : actors){
        System.out.print(actor.getName()+"\t");
        System.out.print(actor.getRole()+"\t");
        System.out.print(actor.getAddress()+"\t\t");
        System.out.print(actor.getNoOfMovies()+"\n");
    }
}
}

```

Main.java

```
public class Main {
```

```

public static void main(String[] args){

    Actor actor1 = new Actor("Prabhas", "Hero", "Hyd", 15);
    Actor actor2 = new Actor("Anushka", "Heroin", "Hyd", 25);
    Actor actor3 = new Actor("Rana", "Villain", "Hyd", 13);
    Actor[] actors = {actor1, actor2, actor3};

    Movie movie = new Movie("Bahubali", "10-July-2015", "S S
Rajamouli", "Shobu Yarlagadda", actors);
    movie.getMovieDetails();

}

}

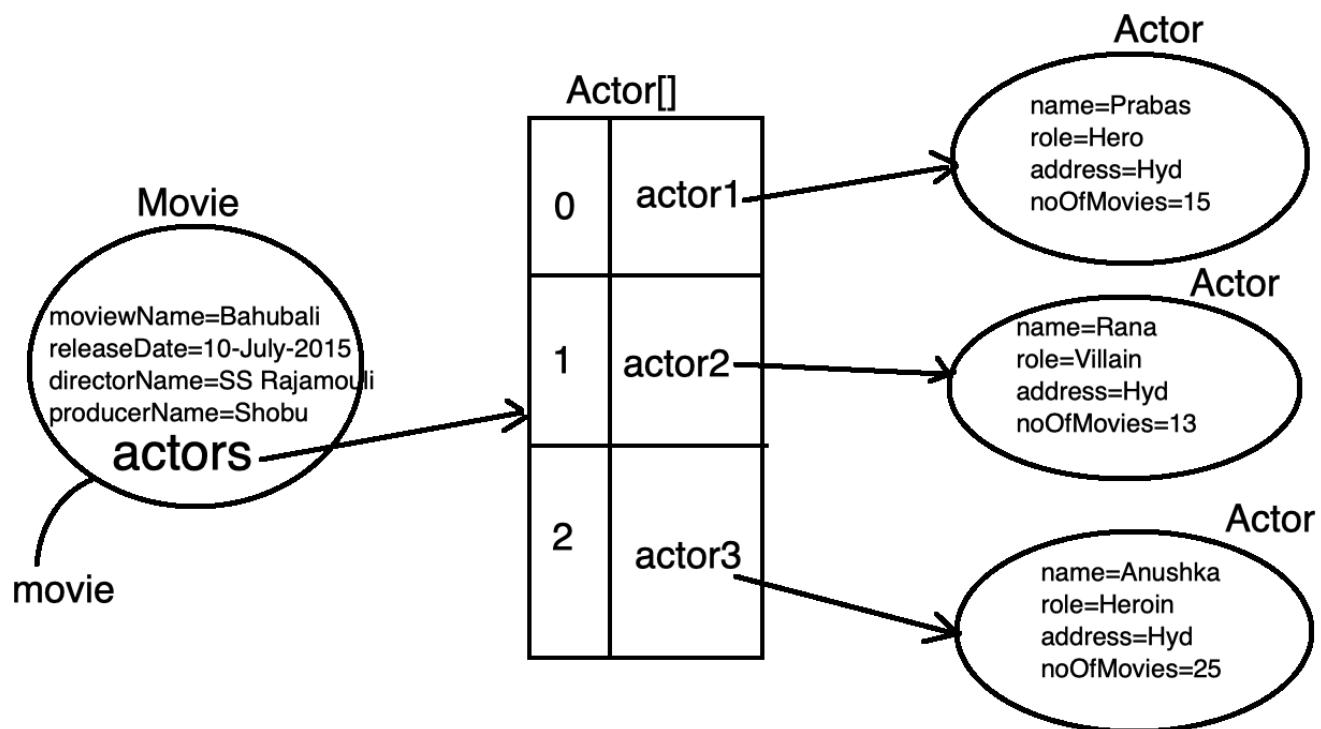
```

Movie Details

Movie Name	:	Bahubali
Release Date	:	10-July-2015
Director Name	:	S S Rajamouli
Producer Name	:	Shobu Yarlagadda

Name Role Address Movies

Prabhas	Hero	Hyd	15
Anushka	Heroin	Hyd	25
Rana	Villain	Hyd	13



Example on Setter Method Dependency Injection:

A single Movie can have multiple Reviews and Ratings:

ReviewAndRating.java

```

package com.durgasoftware.entities;
public class ReviewAndRating {

    private String reviewId;
    private String reviewerName;
    private String reviewDescription;
    private String reviewDate;
    private float rating;
}

```

```
public String getReviewId() {
    return reviewId;
}
public void setReviewId(String reviewId) {
    this.reviewId = reviewId;
}
public String getReviewerName() {
    return reviewerName;
}
public void setReviewerName(String reviewerName) {
    this.reviewerName = reviewerName;
}
public String getReviewDescription() {
    return reviewDescription;
}
public void setReviewDescription(String reviewDescription) {
    this.reviewDescription = reviewDescription;
}
public String getReviewDate() {
    return reviewDate;
}
public void setReviewDate(String reviewDate) {
    this.reviewDate = reviewDate;
}
public float getRating() {
    return rating;
}
public void setRating(float rating) {
    this.rating = rating;
}
}
```

```
Movie.java
package com.durgasoft.entities;
public class Movie {

    private String movieName;
    private String releaseDate;
    private String directorName;
    private String producerName;

    private ReviewAndRating[] reviewAndRatings;
    public String getMovieName() {
        return movieName;
    }
    public void setMovieName(String movieName) {
        this.movieName = movieName;
    }
    public String getReleaseDate() {
        return releaseDate;
    }
    public void setReleaseDate(String releaseDate) {
        this.releaseDate = releaseDate;
    }
    public String getDirectorName() {
        return directorName;
    }
    public void setDirectorName(String directorName) {
        this.directorName = directorName;
    }
    public String getProducerName() {
        return producerName;
    }
    public void setProducerName(String producerName) {
        this.producerName = producerName;
    }
}
```

```
}

public ReviewAndRating[] getReviewAndRatings() {
    return reviewAndRatings;
}

public void setReviewAndRatings(ReviewAndRating[]
reviewAndRatings) {
    this.reviewAndRatings = reviewAndRatings;
}

public void getMovieDetails() {
    System.out.println("Movie Details");
    System.out.println("-----");
    System.out.println("Movie Name : "+movieName);
    System.out.println("Release Date : "+releaseDate);
    System.out.println("Director Name : "+directorName);
    System.out.println("Producer Name : "+producerName);
    System.out.println();

    System.out.println("Review And Ratings");
    int reviewCount = 0;
    for(ReviewAndRating reviewAndRating: reviewAndRatings) {
        reviewCount = reviewCount + 1;
        System.out.println("Review#"+reviewCount);
        System.out.println("\tReview ID : "+reviewAndRating.getReviewId());
        System.out.println("\tReviewer Name : "+reviewAndRating.getReviewerName());
        System.out.println("\tReview Description : "+reviewAndRating.getReviewDescription());
        System.out.println("\tReview Date : "+reviewAndRating.getReviewDate());
        System.out.println("\tRating : "+reviewAndRating.getRating());
        System.out.println();
    }
}
```

```
    }  
}  
  
}
```

Test.java

```
package com.durgasoft.test;  
import com.durgasoft.entities.Movie;  
import com.durgasoft.entities.ReviewAndRating;  
public class Test {  
    public static void main(String[] args) {  
  
        ReviewAndRating reviewAndRating1 = new  
ReviewAndRating();  
        reviewAndRating1.setReviewId("R111");  
        reviewAndRating1.setReviewerName("Suresh Kondeti");  
        reviewAndRating1.setReviewDescription("Block Buster");  
        reviewAndRating1.setReviewDate("10-11-2024");  
        reviewAndRating1.setRating(3.5f);  
  
        ReviewAndRating reviewAndRating2 = new  
ReviewAndRating();  
        reviewAndRating2.setReviewId("R222");  
        reviewAndRating2.setReviewerName("Sairaj");  
        reviewAndRating2.setReviewDescription("Superhit");  
        reviewAndRating2.setReviewDate("10-11-2024");  
        reviewAndRating2.setRating(3.0f);  
  
        ReviewAndRating reviewAndRating3 = new  
ReviewAndRating();  
        reviewAndRating3.setReviewId("R333");  
        reviewAndRating3.setReviewerName("Venkat N");  
        reviewAndRating3.setReviewDescription("Average");
```

```

        reviewAndRating3.setReviewDate("10-11-2024");
        reviewAndRating3.setRating(2.5f);

        ReviewAndRating[] reviewAndRatings = {reviewAndRating1,
reviewAndRating2, reviewAndRating3};

        Movie movie = new Movie();
        movie.setMovieName("Amaran");
        movie.setReleaseDate("09-11-2024");
        movie.setDirectorName("Rajkumar");
        movie.setProducerName("Kamalhasan");
        movie.setReviewAndRatings(reviewAndRatings);

        movie.getMovieDetails();

    }

}


```

Movie Details

```

Movie Name      : Amaran
Release Date   : 09-11-2024
Director Name  : Rajkumar
Producer Name  : Kamalhasan

```

Review And Ratings

Review#1

```

    Review ID       : R111
    Reviewer Name  : Suresh Kondeti
    Review Description : Block Buster
    Review Date    : 10-11-2024
    Rating         : 3.5

```

Review#2

```

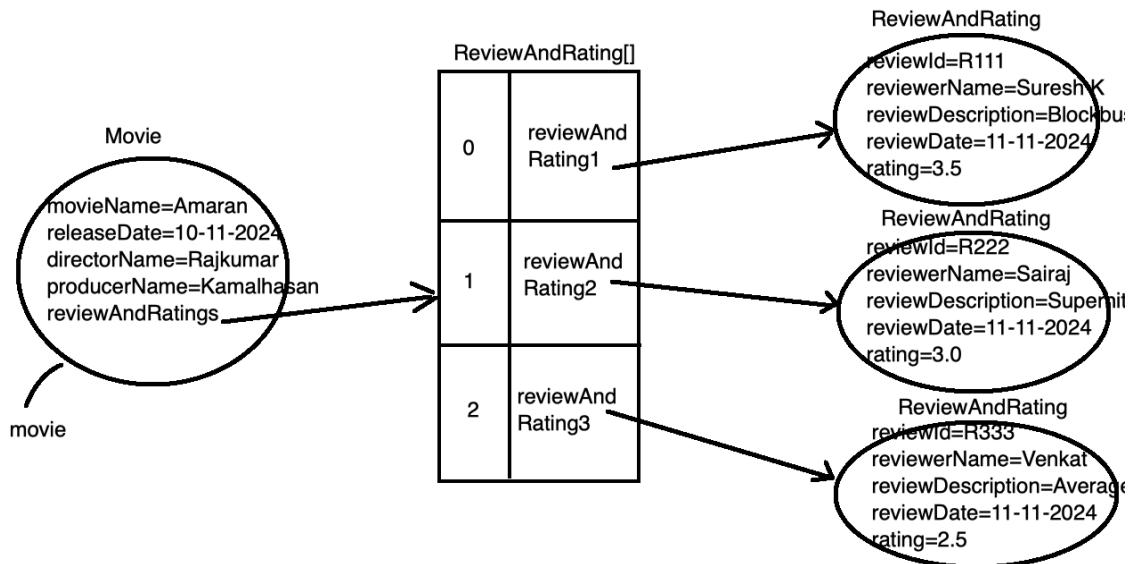
    Review ID       : R222

```

Reviewer Name : Sairaj
 Review Description : Superhit
 Review Date : 10-11-2024
 Rating : 3.0

Review#3

Review ID : R333
 Reviewer Name : Venkat N
 Review Description : Average
 Review Date : 10-11-2024
 Rating : 2.5



Many-To-One Association:

It is a relation between entities, where multiple instances of an entity should be mapped with exactly one instance of another entity.

Example on Constructor Dependency Injection:
 Multiple Orders made a single Customer

EX:

Customer.java

```
public class Customer {  
    private String cid;  
    private String cname;  
    private String mobile;  
    private String email;  
    private String caddr;  
  
    public Customer(String cid, String cname, String mobile,  
String email, String caddr) {  
        this.cid = cid;  
        this.cname = cname;  
        this.mobile = mobile;  
        this.email = email;  
        this.caddr = caddr;  
    }  
  
    public String getCid() {  
        return cid;  
    }  
  
    public String getCname() {  
        return cname;  
    }  
  
    public String getMobile() {  
        return mobile;  
    }  
  
    public String getEmail() {  
        return email;  
    }  
}
```

```
public String getCaddr() {
    return caddr;
}
}

Order.java
public class Order {

    private String orderID;
    private String itemName;
    private String deliveryDate;
    private String deliveryAddress;
    private Customer customer;

    public Order(String orderID, String itemName, String
deliveryDate, String deliveryAddress, Customer customer) {
        this.orderID = orderID;
        this.itemName = itemName;
        this.deliveryDate = deliveryDate;
        this.deliveryAddress = deliveryAddress;
        this.customer = customer;
    }

    public void getOrderDetails(){
        System.out.println("Order Details");
        System.out.println("-----");
        System.out.println("Order ID : " + orderID);
        System.out.println("Item Name : " + itemName);
        System.out.println("Delivery Date : " +
deliveryDate);
        System.out.println("Delivery Address : " +
deliveryAddress);
        System.out.println("Customer Details");
        System.out.println("-----");
    }
}
```

```
        System.out.println("Customer Id      :  
"+customer.getCid());  
        System.out.println("Customer Name     :  
"+customer.getCname());  
        System.out.println("Customer Mobile   :  
"+customer.getMobile());  
        System.out.println("Customer Email    :  
"+customer.getEmail());  
        System.out.println("Customer Address  :  
"+customer.getAddr());  
        System.out.println();  
    }  
}
```

Main.java

```
public class Main {  
    public static void main(String[] args) {  
  
        Customer customer = new Customer("C111", "Durga",  
"9988776655", "durga@gmail.com", "Hyd");  
  
        Order order1 = new Order("O-111", "Samsung Galaxy-22",  
"20-11-2024", "360/3r, S R Nagar, Hyd", customer);  
        Order order2 = new Order("O-222", "Lenovo Laptop",  
"20-11-2024", "360/3r, S R Nagar, Hyd", customer);  
        Order order3 = new Order("O-333", "Gold Ring",  
"20-11-2024", "360/3r, S R Nagar, Hyd", customer);  
  
        order1.getOrderDetails();  
        order2.getOrderDetails();  
        order3.getOrderDetails();  
  
    }  
}
```

Order Details

Order ID : 0-111
Item Name : Samsung Galaxy-22
Delivery Date : 20-11-2024
Delivery Address : 360/3r, S R Nagar, Hyd

Customer Details

Customer Id : C111
Customer Name : Durga
Customer Mobile : 9988776655
Customer Email : durga@gmail.com
Customer Address : Hyd

Order Details

Order ID : 0-222
Item Name : Lenovo Laptop
Delivery Date : 20-11-2024
Delivery Address : 360/3r, S R Nagar, Hyd

Customer Details

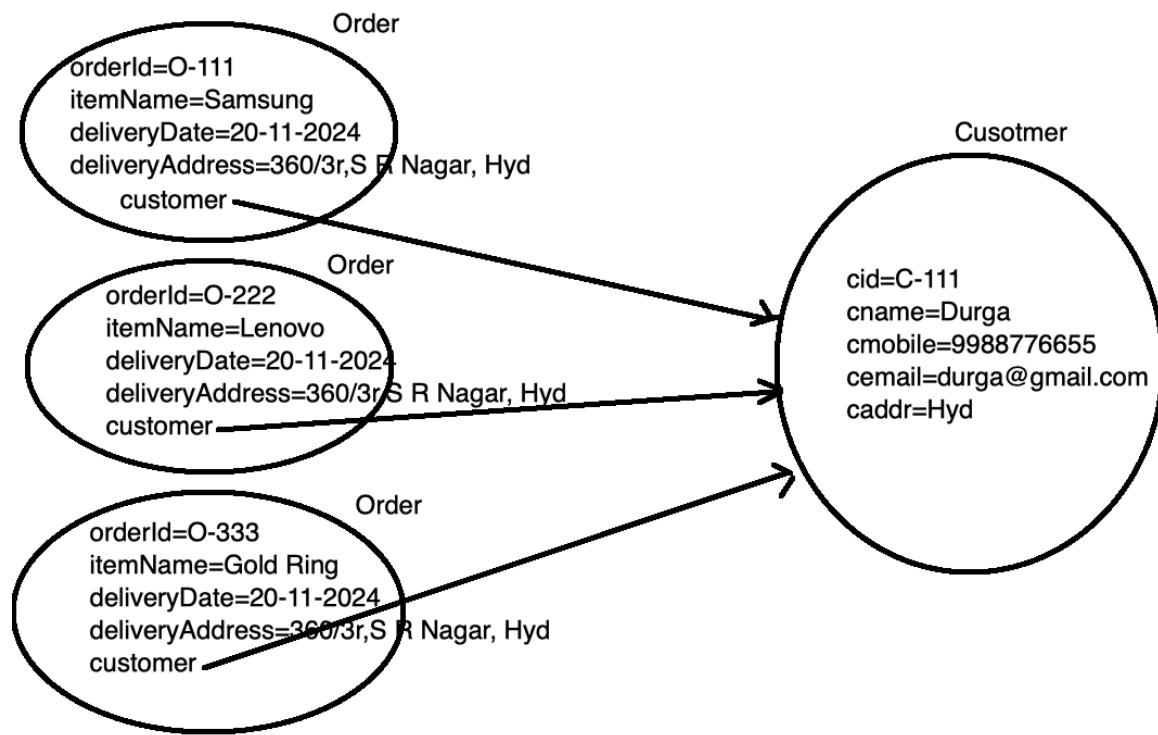
Customer Id : C111
Customer Name : Durga
Customer Mobile : 9988776655
Customer Email : durga@gmail.com
Customer Address : Hyd

Order Details

Order ID : 0-333
Item Name : Gold Ring
Delivery Date : 20-11-2024
Delivery Address : 360/3r, S R Nagar, Hyd

Customer Details

Customer Id : C111
Customer Name : Durga
Customer Mobile : 9988776655
Customer Email : durga@gmail.com
Customer Address : Hyd



Example on the Setter Method Dependency Injection:

Multiple Items may have the same Customer

EX:

Customer.java

```
public class Customer {  
  
    private String cid;  
  
    // Getters and setters...  
}
```

```
private String cname;
private String caddr;
private String cemail;
private String cmobile;

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 String getCaddr() {
    return caddr;
}

public void setCaddr(String caddr) {
    this.caddr = caddr;
}

public String getCemail() {
    return cemail;
}

public void setCemail(String cemail) {
```

```
        this.cemail = cemail;
    }

public String getCmobile() {
    return cmobile;
}

public void setCmobile(String cmobile) {
    this.cmobile = cmobile;
}
}
```

Item.java

```
public class Item {

    private String itemId;
    private String itemName;
    private int itemPrice;
    private String mfgDate;
    private String exprDate;
    private Customer customer;

    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 getItemPrice() {
    return itemPrice;
}

public void setItemPrice(int itemPrice) {
    this.itemPrice = itemPrice;
}

public String getMfgDate() {
    return mfgDate;
}

public void setMfgDate(String mfgDate) {
    this.mfgDate = mfgDate;
}

public String getExprDate() {
    return exprDate;
}

public void setExprDate(String exprDate) {
    this.exprDate = exprDate;
}

public Customer getCustomer() {
    return customer;
}
public void setCustomer(Customer customer) {
    this.customer = customer;
}
```

```
public void getItemDetails(){
    System.out.println("Item Details");
    System.out.println("-----");
    System.out.println("Item Id : " + itemId);
    System.out.println("Item Name : " + itemName);
    System.out.println("Item Price : " + itemPrice);
    System.out.println("Item MfgDate : " + mfgDate);
    System.out.println("Item ExprDate : " + exprDate);
    System.out.println("Customer Details");
    System.out.println("-----");
    System.out.println("Customer Id : " +
customer.getCid());
    System.out.println("Customer Name : " +
customer.getCname());
    System.out.println("Customer Address : " +
customer.getAddr());
    System.out.println("Customer Email Id : " +
customer.getEmail());
    System.out.println("Customer Mobile No : " +
customer.getMobile());
    System.out.println();
}
}
```

Main.java

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

        Customer customer = new Customer();
        customer.setCid("C-111");
        customer.setCname("Durga");
        customer.setAddr("Hyd");
        customer.setEmail("durga123@gmail.com");
```

```
customer.setCmobile("91-9988776655");

Item item1 = new Item();
item1.setItemId("I1");
item1.setItemName("Samsung Mobile");
item1.setItemPrice(50000);
item1.setMfgDate("10-12-2023");
item1.setExprDate("10-12-2028");
item1.setCustomer(customer);

Item item2 = new Item();
item2.setItemId("I2");
item2.setItemName("Laptop");
item2.setItemPrice(60000);
item2.setMfgDate("10-11-2024");
item2.setExprDate("10-11-2029");
item2.setCustomer(customer);

Item item3 = new Item();
item3.setItemId("I3");
item3.setItemName("Charger");
item3.setItemPrice(500);
item3.setMfgDate("10-12-2023");
item3.setExprDate("10-12-2026");
item3.setCustomer(customer);

item1.getItemDetails();
item2.getItemDetails();
item3.getItemDetails();

}
```

Item Details

Item Id : I1
Item Name : Samsung Mobile
Item Price : 50000
Item MfgDate : 10-12-2023
Item ExprDate : 10-12-2028

Customer Details

Customer Id : C-111
Customer Name : Durga
Customer Address : Hyd
Customer Email Id : durga123@gmail.com
Customer Mobile No : 91-9988776655

Item Details

Item Id : I2
Item Name : Laptop
Item Price : 60000
Item MfgDate : 10-11-2024
Item ExprDate : 10-11-2029

Customer Details

Customer Id : C-111
Customer Name : Durga
Customer Address : Hyd
Customer Email Id : durga123@gmail.com
Customer Mobile No : 91-9988776655

Item Details

Item Id : I3
Item Name : Charger
Item Price : 500
Item MfgDate : 10-12-2023

Item ExprDate : 10-12-2026

Customer Details

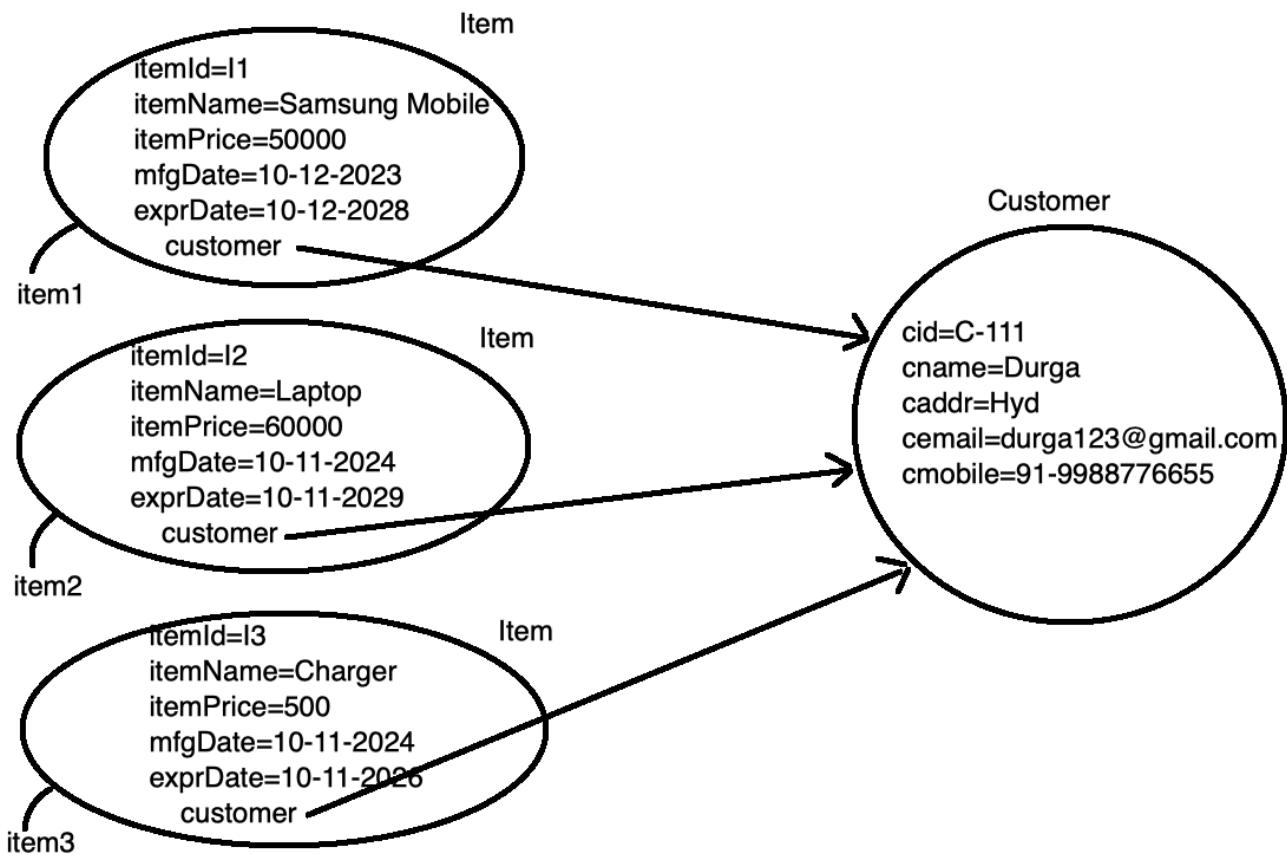
Customer Id : C-111

Customer Name : Durga

Customer Address : Hyd

Customer Email Id : durga123@gmail.com

Customer Mobile No : 91-9988776655



Many-To-Many Association:

It is a relation between entities, where multiple instances of an entity should be mapped with multiple instances of another entity.

Example on Constructor Dependency Injection:

Multiple Books prepared by Multiple Authors

Author.java

```
public class Author {  
  
    private String authorName;  
    private String authorQualification;  
    private String authorExperience;  
  
    public Author(String authorName, String authorQualification,  
String authorExperience) {  
        this.authorName = authorName;  
        this.authorQualification = authorQualification;  
        this.authorExperience = authorExperience;  
    }  
  
    public String getAuthorName() {  
        return authorName;  
    }  
  
    public String getAuthorQualification() {  
        return authorQualification;  
    }  
  
    public String getAuthorExperience() {  
        return authorExperience;  
    }  
}
```

Book.java

```
public class Book {  
  
    private String bookId;  
    private String bookName;  
    private String bookType;  
    private int bookPrice;  
    private Author[] authors;  
  
    public Book(String bookId, String bookName, String bookType,  
int bookPrice, Author[] authors) {  
        this.bookId = bookId;  
        this.bookName = bookName;  
        this.bookType = bookType;  
        this.bookPrice = bookPrice;  
        this.authors = authors;  
    }  
  
    public void getBookDetails(){  
        System.out.println("Book Details");  
        System.out.println("-----");  
        System.out.println("Book ID : " + bookId);  
        System.out.println("Book Name : " + bookName);  
        System.out.println("Book Type : " + bookType);  
        System.out.println("Book Price : " + bookPrice);  
        System.out.println("Authors Details:");  
        int count = 0;  
        for(Author author : authors){  
            count = count + 1;  
            System.out.println("Author#"+count+":");  
            System.out.println("\tAuthor Name : " +  
author.getAuthorName());  
            System.out.println("\tAuthor Qualification : " +  
author.getAuthorQualification());  
        }  
    }  
}
```

```
        System.out.println("\tAuthor Experience : " +
author.getAuthorExperience());
    }
    System.out.println();
}
}

Main.java
public class Main {
    public static void main(String[] args) {
        Author author1 = new Author("Anshul", "BTech", "5 Books");
        Author author2 = new Author("Sujeet", "BTech", "3 Books");
        Author author3 = new Author("Pranav", "BTech", "7 Books");

        Author[] authors1 = {author1, author2, author3};
        Author[] authors2 = {author1, author2};
        Author[] authors3 = {author2, author3};

        Book book1 = new Book("B-111", "JAVA", "Software", 700,
authors1);
        Book book2 = new Book("B-222", "PYTHON", "Software", 500,
authors2);
        Book book3 = new Book("B-333", "CCNA", "Hardware", 400,
authors3);

        book1.getBookDetails();
        book2.getBookDetails();
        book3.getBookDetails();
    }
}
```

Book Details

Book ID : B-111
Book Name : JAVA
Book Type : Software
Book Price : 700

Authors Details:

Author#1:

Author Name : Anshul
Author Qualification : BTech
Author Experience : 5 Books

Author#2:

Author Name : Sujeet
Author Qualification : BTech
Author Experience : 3 Books

Author#3:

Author Name : Pranav
Author Qualification : BTech
Author Experience : 7 Books

Book Details

Book ID : B-222
Book Name : PYTHON
Book Type : Software
Book Price : 500

Authors Details:

Author#1:

Author Name : Anshul
Author Qualification : BTech
Author Experience : 5 Books

Author#2:

Author Name : Sujeet
Author Qualification : BTech
Author Experience : 3 Books

Book Details

Book ID : B-333
Book Name : CCNA
Book Type : Hardware
Book Price : 400

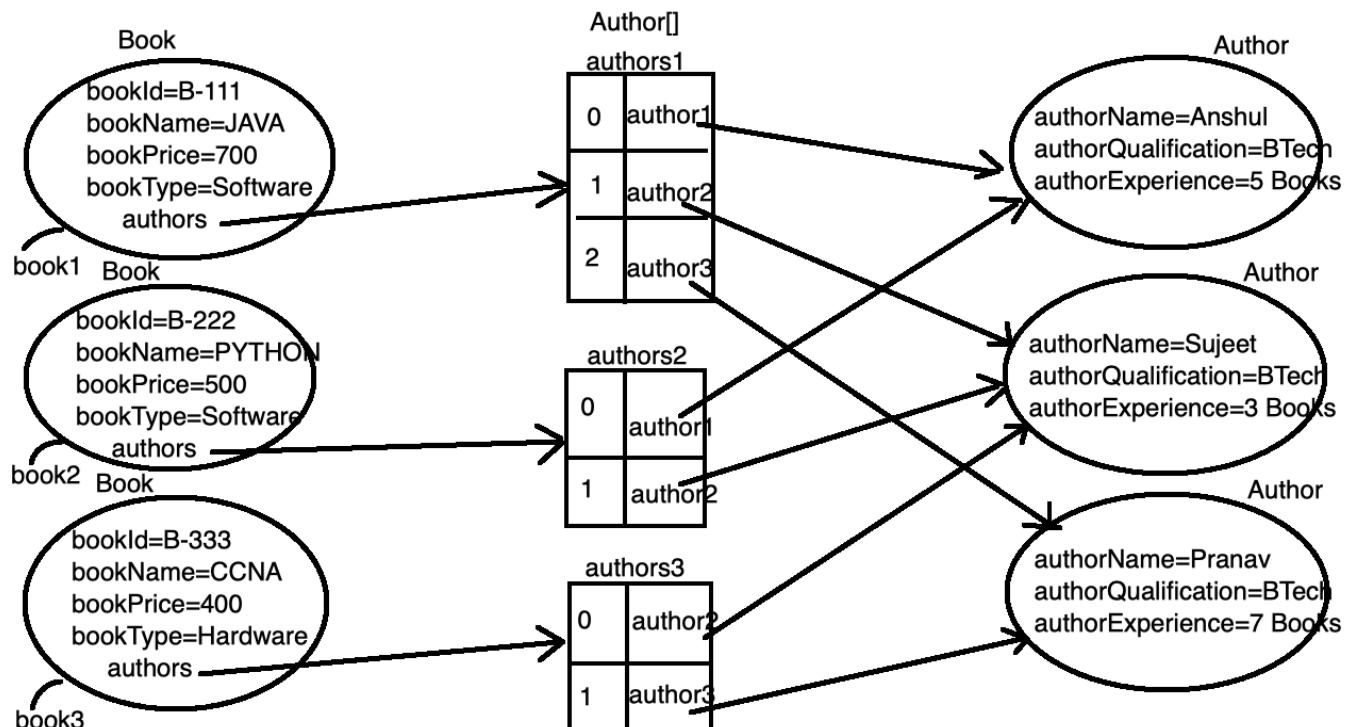
Authors Details:

Author#1:

Author Name : Sujeeet
Author Qualification : BTech
Author Experience : 3 Books

Author#2:

Author Name : Pranav
Author Qualification : BTech
Author Experience : 7 Books



Example on Setter Method Dependency Injection:

Multiple Students have joined with Multiple Courses.

Course.java

```
public class Course {  
  
    private String cid;  
    private String cname;  
    private int ccost;  
  
    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 getCCost() {  
        return ccost;  
    }  
  
    public void setCCost(int ccost) {  
        this.ccost = ccost;  
    }  
}
```

}

Student.java

```
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 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 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\tCCOST");
    System.out.println("-----");
    for (Course c : courses) {
        System.out.print(c.getCid()+"\t");
        System.out.print(c.getCname()+"\t");
        System.out.print(c.getCost()+"\n");
    }
    System.out.println();
}
}

```

Main.java

```

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

        Course course1 = new Course();
        course1.setCid("C111");
        course1.setCname("JAVA ");
        course1.setCost(50000);
    }
}

```

```
Course course2 = new Course();
course2.setCid("C222");
course2.setCname("PYTHON");
course2.setCcost(40000);

Course course3 = new Course();
course3.setCid("C333");
course3.setCname("AWS    ");
course3.setCcost(30000);

Course[] courses1 = {course1, course2, course3};
Course[] courses2 = {course1, course2};
Course[] courses3 = {course2, course3};

Student student1 = new Student();
student1.setSid("S-111");
student1.setSname("Swapnil");
student1.setSaddr("Pune");
student1.setCourses(courses1);

Student student2 = new Student();
student2.setSid("S-222");
student2.setSname("Priya");
student2.setSaddr("Guntur");
student2.setCourses(courses2);

Student student3 = new Student();
student3.setSid("S-333");
student3.setSname("Pranita");
student3.setSaddr("Odisha");
student3.setCourses(courses3);

student1.getStudentDetails();
```

```
    student2.getStudentDetails();
    student3.getStudentDetails();

}

}
```

Student Details

```
Student Id      : S-111
Student Name    : Swapnil
Student Address : Pune
CID            CNAME    CCOST
-----
C111 JAVA      50000
C222 PYTHON    40000
C333 AWS       30000
```

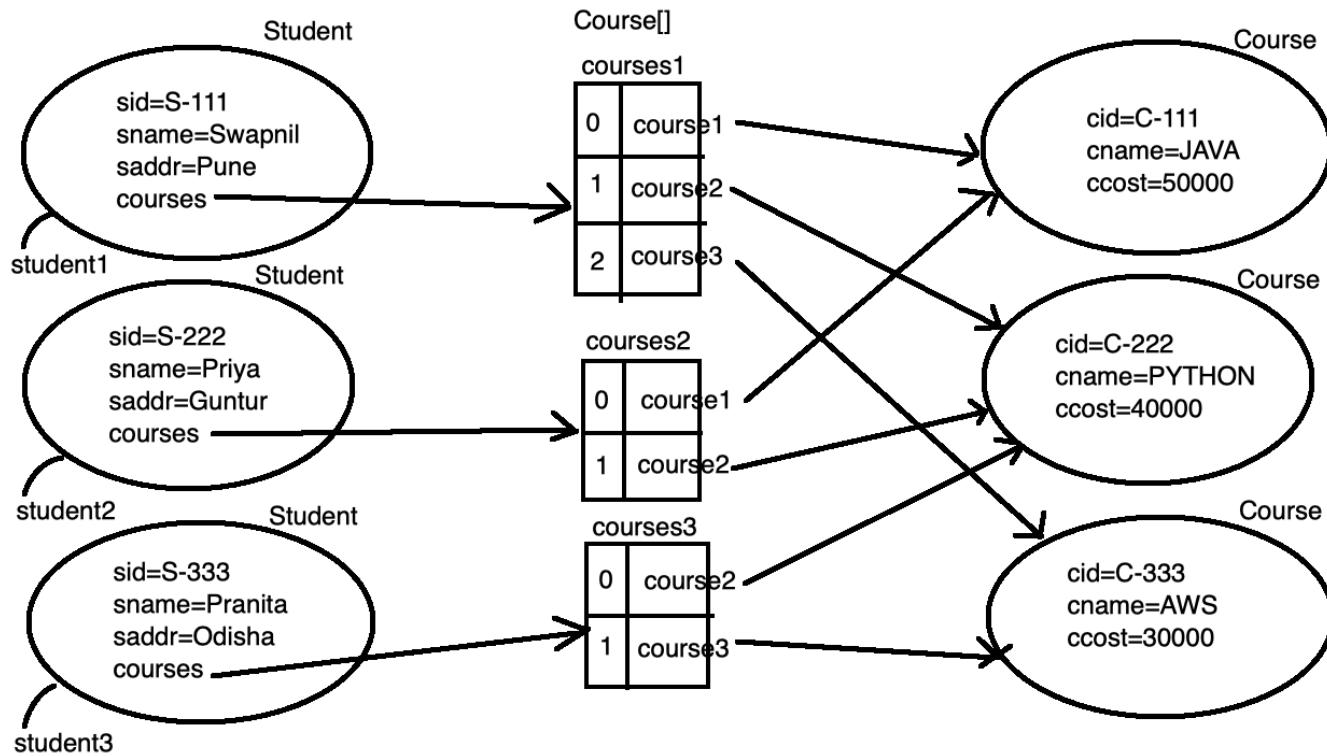
Student Details

```
Student Id      : S-222
Student Name    : Priya
Student Address : Guntur
CID            CNAME    CCOST
-----
C111 JAVA      50000
C222 PYTHON    40000
```

Student Details

```
Student Id      : S-333
Student Name    : Pranita
Student Address : Odisha
CID            CNAME    CCOST
-----
```

C222 PYTHON	40000
C333 AWS	30000



In Object Orientation, Associations are represented in the following two ways.

1. Composition
2. Aggregation

Q) What is the difference between Composition and Aggregation?

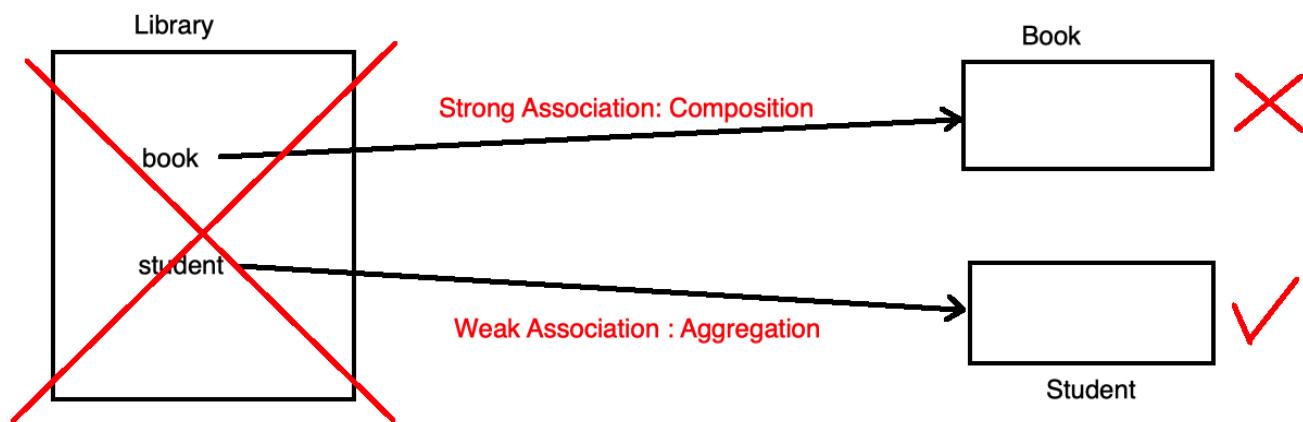
Ans:

Composition is a strong association between entities, if we close the container entity then there is no chance of existing Contained entity, so the lifetime of the Contained entity is dependent on the lifetime of the Container entity.

EX: In the association between Library and Books, if you close the Library then there is no chance of getting Books.

Aggregation is a weak association between entities, even if we close the Container entity then there is a chance of existing Contained entity, so the lifetime of the Contained entity is not dependent on the container entity.

EX: In the association between Library and Students, even if we close the Library then there is a chance for student to exist in the College.



Inheritance:

Inheritance 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 is called a Superclass or a Base class or a Parent class.

Where the class which gets variables and methods is called a Subclass or a Derived class or a child class.

The main advantage of the Inheritance is “Code Reusability”.

If we declare variables and methods in a superclass then we are able to reuse those variables and methods in all the subclasses.

In Object orientation, initially there are two types of inheritances.

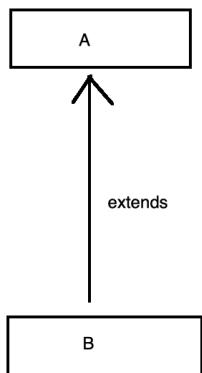
1. Single Inheritance
2. Multiple Inheritance

On the basis of the above two inheritances there are three more inheritances.

1. Multilevel Inheritance
2. Hierarchical Inheritance
3. Hybrid Inheritance

Single Inheritance:

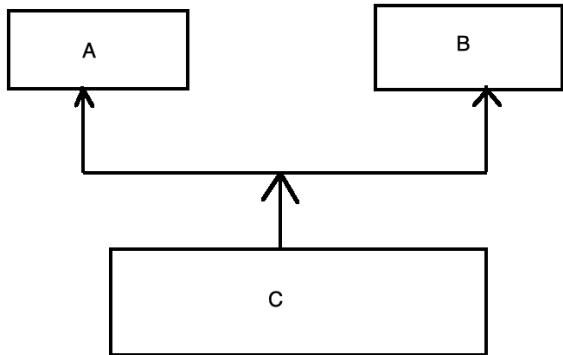
It is a relation between classes, it will provide variables and methods from only one superclass to one or more subclasses.



Java does support Single inheritance.

Multiple Inheritance:

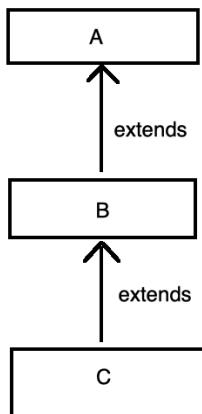
It is a relation between classes, it will provide variables and methods from more than one superclass to one or more subclasses.



Java does not support Multiple Inheritance.

Multilevel Inheritance:

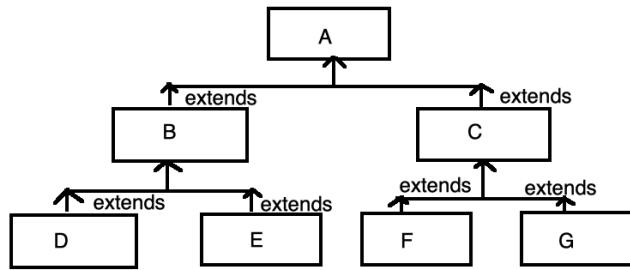
It is the combination of single inheritances in more than one level.



Java does support Multilevel inheritance.

Hierarchical Inheritance:

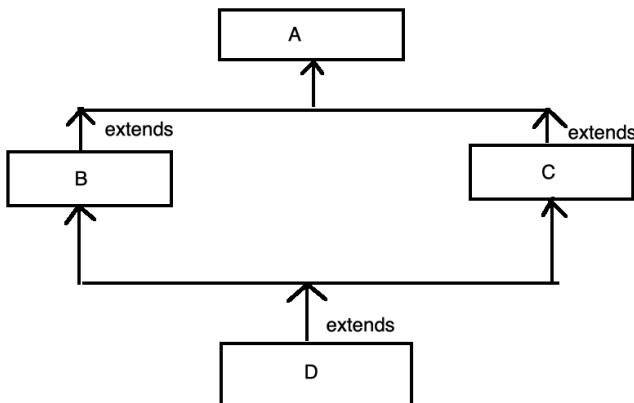
It is the combination of single inheritances in a particular hierarchy.



Java does support Hierarchical Inheritance.

Hybrid Inheritance:

It is the combination of Single and Multiple Inheritances.



Java does not support hybrid inheritance.

In Java applications, by using superclass reference variables we are able to access only superclass members, but by using subclass reference variables we are able to access both superclass members and subclass members.

EX on Single INheritance:

```
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();
        a.m1();
        B b = new B();
        b.m1();
        b.m2();
    }
}
m1-A
m1-A
m2-B
```

Example on Multiple INheritance:

```
class A{
    void m1(){
        System.out.println("m1-A");
    }
}
class B{
    void m2(){
        System.out.println("m2-B");
    }
}
```

```

        }
    }
class C extends A, B{
    void m3(){
        System.out.println("m3-C");
    }
}

```

Status: Compilation Error

Example on Multilevel Inheritance:

```

class A{
    void m1(){
        System.out.println("m1-A");
    }
}
class B extends A{
    void m2(){
        System.out.println("m2-B");
    }
}
class C extends B{
    void m3(){
        System.out.println("m3-C");
    }
}
public class Main {
    public static void main(String[] args){

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

        B b = new B();
        b.m1();
    }
}

```

```
b.m2();  
  
C c = new C();  
c.m1();  
c.m2();  
c.m3();  
  
}  
}
```

```
m1-A  
m1-A  
m2-B  
m1-A  
m2-B  
m3-C
```

Example Hierarchical Inheritance:

```
class A{  
    void m1(){  
        System.out.println("m1-A");  
    }  
}  
class B extends A{  
    void m2(){  
        System.out.println("m2-B");  
    }  
}  
class C extends A{  
    void m3(){  
        System.out.println("m3-C");  
    }  
}  
class D extends B{
```

```
void m4(){
    System.out.println("m4-D");
}
}

class E extends B{
    void m5(){
        System.out.println("m5-E");
    }
}

class F extends C{
    void m6(){
        System.out.println("m6-F");
    }
}

class G extends C{
    void m7(){
        System.out.println("m7-G");
    }
}

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

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

        B b = new B();
        b.m1();
        b.m2();

        C c = new C();
        c.m1();
        c.m3();

        D d = new D();
    }
}
```

```
d.m1();
d.m2();
d.m4();

E e = new E();
e.m1();
e.m2();
e.m5();

F f = new F();
f.m1();
f.m3();
f.m6();

G g = new G();
g.m1();
g.m3();
g.m7();

}

}
```

m1-A
m1-A
m2-B
m1-A
m3-C
m1-A
m2-B
m4-D
m1-A
m2-B
m5-E

```
m1-A  
m3-C  
m6-F  
m1-A  
m3-C  
m7-G
```

Example on Hybrid Inheritance:

```
class A{  
    void m1(){  
        System.out.println("m1-A");  
    }  
}  
class B extends A{  
    void m2(){  
        System.out.println("m2-B");  
    }  
}  
class C extends A{  
    void m3(){  
        System.out.println("m3-C");  
    }  
}  
class D extends B,C{  
    void m4(){  
        System.out.println("m4-D");  
    }  
}
```

Status: Compilation Error

If we declare variables and methods without private in a superclass then we are able to access them directly in the subclasses, but if we declare variables and methods in a subclass then it is not possible to access them in the superclass.

EX

```
class A{
    int i = 10;
    void m1(){
        System.out.println("m1-A");
        //System.out.println(j); ----> Error
        //m2(); ----> Error
    }
}
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();
        a.m1();
        B b = new B();
        b.m2();
    }
}
```

m1-A

m2-B

10

m1-A

Static Context in Inheritance:

In Java applications, static context is represented by the following three elements.

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

Where the static variables and static blocks are recognized and executed the moment when we load the respective class bytecode to the memory.

Where the static method will be executed the moment when we access it.

In Inheritance, when we create an object for a subclass , first JVM will has to load the subclass bytecode to the memory, but in Java, JVM will load subclass bytecode to the memory after loading superclass bytecode only.

In Inheritance, JVM will load all the classes bytecode from superclass to subclass ordering. In this context, if we provide static members in both superclass and subclass then JVM will execute superclass static members at the time of loading superclass bytecode and the subclass static members at the time of loading subclass bytecode, so here all the static members are executed as per the classes loading order that is from superclass to subclass.

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();
    }
}
```

SB-A

SB-B

SB-C

EX:

```
class A{
    static {
        System.out.println("SB-A");
    }
    static int i = m1();
    static int m1(){
        System.out.println("m1-A");
    }
}
```

```
        return 10;
    }
}

class B extends A{
    static int m2(){
        System.out.println("m2-B");
        return 20;
    }
    static{
        System.out.println("SB-B");
    }
    static int j = m2();
}

class C extends B{
    static int k = m3();
    static {
        System.out.println("SB-C");
    }
    static int m3(){
        System.out.println("m3-C");
        return 30;
    }
}

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

SB-A
m1-A
SB-B
m2-B
m3-C
SB-C

EX:

```
class A{
    static int i = m1();
    static{
        System.out.println("SB-A");
    }
    static int m1(){
        System.out.println("m1-A");
        return 10;
    }
}
class B extends A{
    static{
        System.out.println("SB-B");
    }
    static int m2(){
        System.out.println("m2-B");
        return 20;
    }
    static int j = m2();
}
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();
C c3 = new C();
}
}
```

m1-A
SB-A
SB-B
m2-B
m3-C
SB-C

Instance Context In Inheritance

In Java applications, instance context is represented by the following three elements.

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

Where the instance Variables and instance blocks are recognized and executed automatically just before executing the respective class constructor in Object creation.

Where the Instance Method will be recognized and executed the moment when we access it.

In inheritance, when we create an object for Subclass then JVM has to execute the subclass constructor, but in java, before executing the subclass constructor JVM must recognize and execute a 0-arg constructor in the superclass, after executing the superclass's 0-arg constructor only JVM will execute the subclass

constructor. in the case if 0-arg constructor is not available in the superclass then the compiler will raise an error.

In inheritance, if we provide instance context in all classes then the instance context of the classes are executed just before executing the respective class constructor, so the instance context execution order is dependent on the constructor execution order that is from superclass to subclass order.

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();
    }
}
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{
}
class C extends B{
```

```

C(){
    System.out.println("C-Con");
}
}

public class Main {
    public static void main(String[] args) {
        C c = new C();
    }
}
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();
B(){
    System.out.println("B-Con");
}

```

```
int m2(){
    System.out.println("m2-B");
    return 20;
}
class C extends B{
    int m3(){
        System.out.println("m3-C");
        return 30;
    }
    int k = m3();
    {
        System.out.println("IB-C");
    }
    C(){
        System.out.println("C-Con");
    }
}
public class Main {
    public static void main(String[] args) {
        C c = new C();
    }
}
```

IB-A
m1-A
A-Con
IB-B
m2-B
B-Con
m3-C
IB-C
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();
B(){
    System.out.println("B-Con");
}
int m2(){
    System.out.println("m2-B");
    return 20;
}
}
class C extends B{
int m3(){
    System.out.println("m3-C");
    return 30;
}
int k = m3();
{
```

```
        System.out.println("IB-C");
    }
    C(){
        System.out.println("C-Con");
    }
}
public class Main {
    public static void main(String[] args) {
        C c1 = new C();
        System.out.println();
        C c2 = new C();
    }
}
```

IB-A
m1-A
A-Con
IB-B
m2-B
B-Con
m3-C
IB-C
C-Con

IB-A
m1-A
A-Con
IB-B
m2-B
B-Con
m3-C
IB-C
C-Con

EX:

```
class A{
    A(){
        System.out.println("A-Con");
    }
    static{
        System.out.println("SB-A");
    }
    static int l = m4();
    {
        System.out.println("IB-A");
    }
    int i = m1();
    int m1(){
        System.out.println("m1-A");
        return 10;
    }
    static int m4(){
        System.out.println("m4-A");
        return 40;
    }
}
class B extends A{
    static{
        System.out.println("SB-B");
    }
    static int m = m5();
    static int m5(){
        System.out.println("m5-B");
        return 50;
    }
    B(){
        System.out.println("B-Con");
    }
}
```

```
{  
    System.out.println("IB-B");  
}  
int j = m2();  
int m2(){  
    System.out.println("m2-B");  
    return 20;  
}  
}  
}  
class C extends B{  
    int m3(){  
        System.out.println("m3-C");  
        return 30;  
    }  
    static int m6(){  
        System.out.println("m6-C");  
        return 60;  
    }  
    int k = m3();  
    static int n = m6();  
{  
    System.out.println("IB-C");  
}  
static{  
    System.out.println("SB-C");  
}  
C(){  
    System.out.println("C-Con");  
}  
}  
}  
public class Main {  
    public static void main(String[] args) {  
        C c1 = new C();  
        System.out.println();
```

```
C c2 = new C();  
}  
}
```

SB-A
m4-A
SB-B
m5-B
m6-C
SB-C
IB-A
m1-A
A-Con
IB-B
m2-B
B-Con
m3-C
IB-C
C-Con

IB-A
m1-A
A-Con
IB-B
m2-B
B-Con
m3-C
IB-C
C-Con

‘super’ Keyword:

‘super’ is a Java keyword, it is able to represent the superclass object from the subclasses.

In java applications, there are three ways to utilize the super keyword.

1. To refer to the superclass variables.
2. To refer to the superclass methods.
3. To refer to the superclass constructors.

referring superclass variables by using the ‘super’ keyword:

If we want to refer to the superclass variables by using the super keyword then we have to use the following syntax.

`super.varName;`

Note: In inheritance, when we have the same set of variables at superclass and at subclass, if we access that variables in the subclass then the subclass variables data will be accessed, in this context to refer to the superclass variables over the subclass variables then we have to use the ‘super’ keyword.

EX:

```
class A{  
    int i = 10;  
    int j = 20;  
}  
class B extends A{  
    int i = 100;  
    int j = 200;
```

```

B(int i, int j){
    System.out.println("Local Variables : "+i+
"+j);
    System.out.println("Class Level variables : "+
+this.i+" "+this.j);
    System.out.println("Super Class Level variables : "+
+super.i+" "+super.j);
}
}

public class Main {
    public static void main(String[] args) {
        B b = new B(1000,2000);
    }
}

```

Local Variables : 1000 2000
 Class Level variables : 100 200
 Super Class Level variables : 10 20

Referring superclass methods by using the ‘super’ keyword:

If we want to access a particular superclass method by using the ‘super’ keyword then we have to use the following syntax.

`super.methodName([ParamValues]);`

Note: In Java applications, when we have the same method at both superclass and subclass, if we access that method in the subclass then the subclass method implementation will be executed, in this context, if we want to execute the superclass method over the subclass method then 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();
    }
}

```

m2-B
m1-B
m1-B
m1-A

Referring Superclass constructors by using the “super” keyword:

If we want to refer to[Accessing] a particular superclass constructor by using the “super” keyword then we have to use the following syntax.

```
super([ParamValues]);  
  
super(): Accessing Superclass's 0-arg constructor.  
super(10): Accessing Superclass provided int parameterized  
constructor.  
super(22.22f): Accessing Superclass provided float parameterized  
constructor.
```

Note: In inheritance, when we access subclass constructor as part of subclass object creation, JVM has to execute subclass constructor, but in JAVA before executing the subclass constructor JVM must execute the 0-arg constructor in the superclass, in this context, in superclass if we want to execute a particular constructor like parameterized constructor in place of the 0-arg constructor,... then we have to use the "super" keyword in the subclasses.

EX:

```
class A{  
    A(){  
        System.out.println("A-Con");  
    }  
    A(int i){  
        System.out.println("A-Int-Param-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();  
}  
}
```

A-Int-Param-Con
B-Con

In Inheritance, to access a particular superclass constructor from subclasses if use the super keyword then the respective super statement must be provided as the first statement.

In Inheritance, to access a particular superclass constructor from subclasses if use the super keyword then the respective super statement must be provided in a subclass constructor only, not possible to provide the respective super() statement in the normal java methods.

EX:

```
class A{  
    A(){  
        System.out.println("A-Con");  
    }  
    A(int i){  
        System.out.println("A-Int-Param-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

EX:

```
class A{
    A(){
        System.out.println("A-Con");
    }
    A(int i){
        System.out.println("A-Int-Param-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();
        b.m1();
    }
}
Status: Compilation Error
```

Q) Is it possible to refer to the more than one superclass constructor from a single subclass constructor by using super keywords?

Ans:

No, it is not possible to access more than one superclass constructor from a single subclass constructor by using super keywords, because in the subclass constructor, the super statement must be the first statement. If we provide more than one super() statement in the subclass constructor then the first super() statement is valid and all the remaining super() statements are invalid.

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); -----> Invalid
        System.out.println("B-Con");
    }
}
```

```
public class Main {  
    public static void main(String[] args) {  
        B b = new B();  
  
    }  
}
```

Status: Compilation Error.

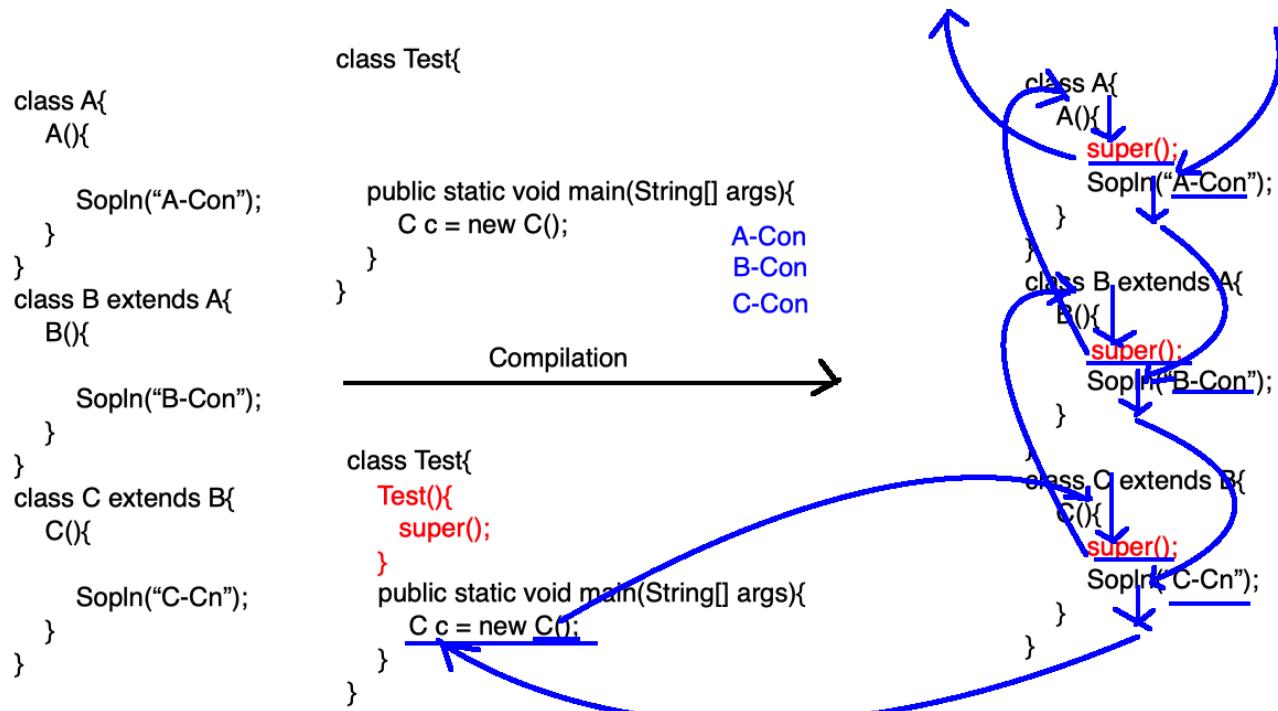
In java applications, in inheritance, when we compile a java file that has the number of classes with inheritance relation then the compiler will perform the following actions.

1. Compiler will go to each and every class and check whether any requirement to provide default constructor or not, if any user defined constructor is provided explicitly in any class then the compiler will not provide default constructor in that class, if no user defined constructor is provided in any class then the compiler will provide a default constructor in the respective class.
2. Compiler will go to each and every constructor in each and every class and the compiler will check whether any super() statement is available or not to access superclass constructor, if no super() is provided explicitly in any constructor then the compiler will provide a super() statement in order to access a 0-arg constructor in the superclass, if a super() statement is provided explicitly in a constructor then the compiler will not provide super() statement in the respective constructor.
3. Compiler will check whether all the superclasses are having right constructors or not as per the subclasses constructors provided super() statements, if any superclass does not have the right constructor as per the super() statement provided

in the subclass constructor then the compiler will raise an error, if all the superclasses are having right constructors as per the super() statements provided in the subclasses constructors then the compiler will not raise any error.

After the successful compilation, at runtime if we create an object for the subclass , as part of it if we access the subclass constructor then JVM will perform the following actions.

1. JVM will go inside the subclass constructor, when it encounters a super() statement in the subclass constructor, JVM will go to the respective superclass and execute the respective constructor.
2. After executing the superclass constructor JVM will come back to the respective subclass constructor and continue its remaining execution part.



```

class Test{
class A{
    A(){
        SopIn("A-Con");
    }
}
class B extends A{
    B(int l){
        SopIn("B-Con");
    }
}
class C extends B{
    C(){
        SopIn("C-Cn");
    }
}

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

```

Compilation Error:

```

class A{
    A(){
        super();
        SopIn("A-Con");
    }
}
class B extends A{
    B(int l){
        super();
        SopIn("B-Con");
    }
}
class C extends B{
    C(){
        super();
        SopIn("C-Cn");
    }
}

```

Compilation

A-Con
B-Con
C-Con

```

class Test{
class A{
    A(){
        super();
        SopIn("A-Con");
    }
}
class B extends A{
    B(int l){
        super();
        SopIn("B-Con");
    }
}
class C extends B{
    C(){
        super(10);
        SopIn("C-Cn");
    }
}

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

```

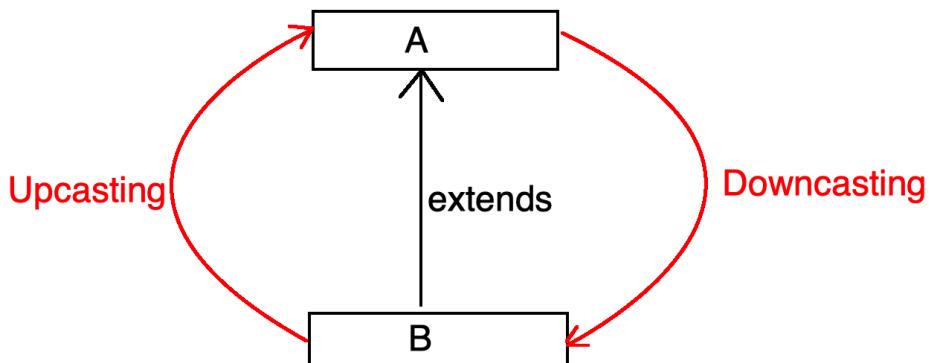
Class Level Type Casting:

The process of converting the data from one user defined data type to another user defined data type is called User defined Data Types type casting or Class level type casting.

To perform the User defined data types type casting we need either extends or implements relation between the two user defined data types.

There are two types of User defined data types type castings.

1. Upcasting
2. Downcasting



Upcasting:

The process of converting the data from subclass types to the respective superclass types is called Upcasting.

To perform Upcasting we have to assign Subclass reference variables to the respective superclass reference variables directly.

EX:

```
class A{  
}  
class B extends A{  
}
```

Case-1:

```
B b = new B();  
A a = b; -----> Assignment Statement
```

Case-2:

```
A a = new B(); ----->Assignment Statement
```

When we compile the above assignment statements, the compiler will perform the following actions.

1. Compiler will recognize the right side variable's data type and the left side variable's data type.
2. Compiler will check whether the right side variable's data type is compatible with the left side variable's data type.
3. If the right side variable's data type is not compatible with the left side variable's data type then the compiler will raise an error like "Incompatible Types Error".
4. If the right side variable's data type is compatible with the left side variable's data type then the compiler will not raise an error like "Incompatible Types Error".

Note: In Java applications, always subclass types are compatible with the superclass types, so we can assign subclass reference variables to the superclass reference variables directly without using any cast operator, but superclass types are not compatible with the subclass types, so it is not possible to assign superclass reference variables to the subclass reference variables directly, here to assign superclass reference variables

to the subclass reference variables we need cast operator explicitly.

Note: In Java applications, the compiler is responsible for type checking only, not typecasting, but JVM is responsible for Type casting at runtime of the application.

When JVM executes the above assignment statements, JVM will perform the following actions.

1. JVM will Convert the right side variables type to the left side variables type internally.
2. JVM will copy the value from the right side variable to the left side variable.

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;  
        a.m1();
```

```
    }  
}  
  
m1-A  
m2-B  
m1-A
```

In Java applications, when we have a situation like creating an object for a subclass and accessing only superclass members without the subclass members then we have to use Upcasting.

In Upcasting, we will create object for subclass so both superclass members and the subclass members are available , here if we want to access only superclass members , not subclass members then we have to use superclass reference variable, with the superclass reference variable we are able to access only superclass members.

Note: In Method Overriding, we have to access only superclass method but JVM must execute the respective subclass method, to achieve this requirement we have to use Upcasting, were in Upcasting we will create object for subclass and we will declare the reference variable for superclass, so here we are able to access only superclass method , but as per method overriding JVM will execute the respective subclass method

EX:

```
class A{  
    void m1(){  
        System.out.println("Old Implementation");  
    }  
}  
class B extends A{  
    void m1(){
```

```

        System.out.println("New Implementation");
    }
}

public class Main {
    public static void main(String[] args) {
        A a = new B();
        a.m1();
    }
}

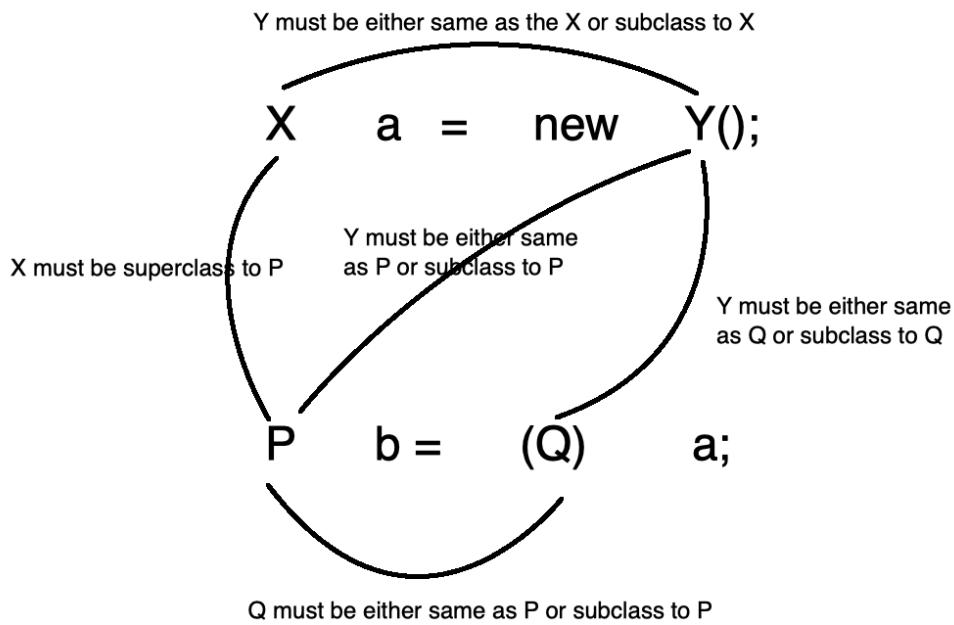
```

New Implementation

Downcasting:

The process of converting the data from superclass type to the subclass type is called Downcasting.

To perform Downcasting we have to use the following Pattern.



In Java Applications, 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, because Superclass types are not compatible with the subclass types, so we are unable to assign superclass reference variables to the subclass reference variable directly.

Case#2:

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

Status: No Compilation Error, but we will get java.lang.ClassCastException.

Reason: In Java applications, it is possible to keep a subclass object reference value in the superclass reference variable, but it is not possible to access superclass reference value in the subclass reference variable, if we keep superclass object reference value in the 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 and No ClassCastException.

In Case#3, if we compile the assignment statement then the compiler will perform the following actions.

1. Compiler will recognize the cast operator provided data type and the left side variable's data type.
2. Compiler will check whether the Cast Operator provided Data type is compatible with the left side variable's data type or not.
3. If the Cast operator provided data type is not compatible with the left side variable's data type then the compiler will raise an error like Incompatible Types.
4. If the Cast operator provided data type is compatible with the left side variable's data type then the compiler will not raise an error like Incompatible Types.

Note: In Java applications, Compiler is responsible for only Type Checking, it is not responsible for Type Casting.

When we execute the above Assignment statement in Case#3, JVM will perform the following actions.

1. JVM will convert the data from Right Side variable's data type to the Cast operator provided data type at right only, it is called Downcasting.
2. JVM will copy the value from the right side variable to the left side variable.

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();
    }
}
```

m1-A

m2-B

Consider the following classes.

```
class A{  
}  
class B extends A{  
}  
class C extends B{  
}  
class D extends C{  
}
```

EX-1:

```
A a = new A();  
D d = a;  
Status: Compilation Error
```

EX-2:

```
A a = new A();  
D d = (D)a;  
Status: No Compilation Error, ClassCastException
```

EX-3:

```
A a = new A();  
B b = (D)a;  
Status: No Compilation Error, ClassCastException
```

EX-4:

```
A a = new D();  
D d = (C)a;  
Status: Compilation Error
```

EX-5:

```
A a = new D();
C c = (D)a;
Status: No Compilation Error, No Exception
```

EX-6:

```
A a = new C();
C c = (D)a;
Status: No Compilation Error, ClassCastException
```

EX-7:

```
A a = new B();
B b = (D)a;
Status: No Compilation Error, ClassCastException
```

EX-8:

```
A a = new D();
D d = (D)(C)((AB))a;
Status: No Compilation Error, No Exception
```

EX-9:

```
A a = new C();
D d = (D)(C)(B)a;
Status: No Compilation Error, ClasCastException
```

EX:

```
abstract class Account{
    public abstract void getAccountDetails();
}

class SavingsAccount extends Account{
    public void getAccountDetails(){
        System.out.println("Savings Account Details.....");
    }
}

class CurrentAccount extends Account{
```

```
public void getAccountDetails(){
    System.out.println("Current Account Details.....");
}
}

class Bank{
    public Account getAccount(int value){
        if(value < 5){
            return new SavingsAccount();
        }else{
            return new CurrentAccount();
        }
    }
}

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

        Bank bank = new Bank();
        /*Account account1 = bank.getAccount(3);
        SavingsAccount savingsAccount = (SavingsAccount)
account1;*/
        SavingsAccount savingsAccount = (SavingsAccount)
bank.getAccount(3);
        savingsAccount.getAccountDetails();

        /* Account account2 = bank.getAccount(10);
        CurrentAccount currentAccount = (CurrentAccount)
account2;*/
        CurrentAccount currentAccount = (CurrentAccount)
bank.getAccount(10);
        currentAccount.getAccountDetails();
    }
}

Savings Account Details.....
```

Current Account Details.....

USES-A Relationship:

It is a relation between entities, where one entity must use another entity up to a particular behavior or method.

To provide USES-A relation between entities in Java applications we have to declare one Entity reference variable as parameter to the respective method.

EX:

Account.java

```
public class Account {  
  
    private String accountNumber;  
    private String accountHolderName;  
    private String accountType;  
    private long accountBalance;  
  
    public Account(String accountNumber, String accountHolderName,  
String accountType, long accountBalance) {  
        this.accountNumber = accountNumber;  
        this.accountHolderName = accountHolderName;  
        this.accountType = accountType;  
        this.accountBalance = accountBalance;  
    }  
  
    public String getAccountNumber() {  
        return accountNumber;  
    }  
  
    public void setAccountNumber(String accountNumber) {  
        this.accountNumber = accountNumber;  
    }  
}
```

```
}

public String getAccountHolderName() {
    return accountHolderName;
}

public void setAccountHolderName(String accountHolderName) {
    this.accountHolderName = accountHolderName;
}

public String getAccountType() {
    return accountType;
}

public void setAccountType(String accountType) {
    this.accountType = accountType;
}

public long getAccountBalance() {
    return accountBalance;
}

public void setAccountBalance(long accountBalance) {
    this.accountBalance = accountBalance;
}

}
```

Transaction.java

```
public class Transaction {
    private String transactionID;
    private String transactionType;
    private String transactionDateAndTime;
```

```
public Transaction(String transactionID, String
transactionType, String transactionDateAndTime) {
    this.transactionID = transactionID;
    this.transactionType = transactionType;
    this.transactionDateAndTime = transactionDateAndTime;
}

public void deposit(Account account, int depositAmount) {
    long accountBalance = account.getAccountBalance();
    accountBalance = accountBalance + depositAmount;
    account.setAccountBalance(accountBalance);

    System.out.println("Transaction Details");
    System.out.println("-----");
    System.out.println("Transaction ID : " +
transactionID);
    System.out.println("Account Number : " +
account.getAccountNumber());
    System.out.println("Account Holder Name : " +
account.getAccountHolderName());
    System.out.println("Account Type : " +
account.getAccountType());
    System.out.println("Transaction Type : " +
transactionType);
    System.out.println("Transaction Date : " +
transactionDateAndTime);
    System.out.println("Deposit Amount : " +
depositAmount);
    System.out.println("Account Balance : " +
account.getAccountBalance());
    System.out.println("Transaction Status : SUCCESS");
    System.out.println("*****Thank You, Visit
Again*****");
}
```

```
}
```

Main.java

```
public class Main {  
    public static void main(String[] args) {  
        Account account = new Account(  
            "abc123",  
            "Durga",  
            "Savings",  
            25000  
        );  
        Transaction transaction = new Transaction(  
            "1234bavd3456",  
            "DEPOSIT",  
            "10:25:13 29-11-2024 "  
        );  
  
        transaction.deposit(account, 10000);  
  
    }  
}
```

Transaction Details

Transaction ID	:	1234bavd3456
Account Number	:	abc123
Account Holder Name	:	Durga
Account Type	:	Savings
Transaction Type	:	DEPOSIT
Transaction Date	:	10:25:13 29-11-2024
Deposit Amount	:	10000
Account Balance	:	35000
Transaction Status	:	SUCCESS

*****Thank You, Visit Again*****

Polymorphism:

Polymorphism is a GREEK word, where poly means many and morphism means forms or Structures,...

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

The main advantage of Polymorphism is “Flexibility” to develop the applications.

There are two types of Polymorphisms.

1. Static Polymorphism
2. Dynamic Polymorphism

Static Polymorphism:

If the Polymorphism exhibits at compilation time then that Polymorphism is called Static Polymorphism.

EX: Method Overloading.

Dynamic Polymorphism:

If the Polymorphism exhibits at runtime then that Polymorphism is called Dynamic Polymorphism.

EX: Method Overriding.

Method Overloading:

The process of extending the existing method functionality to a 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 the same method name and with different method signatures.

In method overloading, differences in the method parameters may be in either of the following forms.

1. Difference in the number of parameters.

```
add(int a, int b){    }
add(int a, int b, int c){    }
```

2. Difference in the Parameter types:

```
add(int a, int b){    }
add(float f1, float f2){    }
```

3. Difference in the order of the parameters:

```
add(int a, float f){    }
add(float f, int a){    }
```

EX:

```
class Employee{
    public void generateSalary(int basic, float hike, int ta,
float pf){
        double salary =
basic+((basic*hike)/100)+ta-((basic*pf)/100);
        System.out.println("Salary : "+salary);
    }
    public void generateSalary(int basic, float hike, int ta,
float pf, int bonus){}
```

```
        double salary =
basic+((basic*hike)/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(25000, 25.0f, 3000, 12.5f);
        employee.generateSalary(25000, 25.0f, 3000, 12.5f, 5000);

    }
}

Salary      : 31125.0
Salary      : 36125.0
```

Method Overriding:

The process of providing replacement for an existing method functionality with a new functionality is called Method Overriding.

To perform Method Overriding in Java applications, we have to provide inheritance, where we have to declare the same method in both superclass and subclass with the same prototype.

In the above context, the superclass method has the old functionality and the subclass method must have the new functionality, here subclass method functionality overrides the superclass method functionality.

To prove method overriding we have to access the superclass method but JVM must execute the respective subclass method in place of the superclass method.

To prove method overriding in Java applications 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 happened here  
Reason: Method Overriding needs Subclass objects.
```

Case#2:

```
B b = new B();  
b.m1();  
Status: Method Overriding has been performed here when we created  
a Subclass object, but to prove method overriding we must access  
superclass method but not subclass method.
```

Case#3:

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

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

OP: New Functionality.

Status: Method Overriding has been performed and proved successfully.

```
class A{
    void m1(){
        System.out.println("Old Functionality");
    }
}

class B extends A{
    void m1(){
        System.out.println("New Functionality");
    }
}

public class Main {
    public static void main(String[] args) {
        /*A a = new A();
        a.m1();*/
        
        /*B b = new B();
        b.m1();*/
        
        A a = new B();
        a.m1();
    }
}
```

New Functionality

EX:

```
class DBDriver {
    public void getDriver(){
```

```

        System.out.println("Type-1 Driver");
    }
}

class NewDBDriver extends DBDriver {
    public void getDriver(){
        System.out.println("Type-4 Driver");
    }
}

public class Main {
    public static void main(String[] args) {
        /*DBDriver driver = new DBDriver();
        driver.getDriver();*/

        /* NewDBDriver driver = new NewDBDriver();
        driver.getDriver();*/

        DBDriver driver = new NewDBDriver();
        driver.getDriver();
    }
}

```

Type-4 Driver

Note: In Java applications, if we want to change the functionality of a particular method without changing the code in the respective method then we have to use Method Overriding.

Rules and Regulations to perform Method Overriding:

1. In Method Overriding, the superclass method must not be declared as private.

EX:

```

class A{
    private void m1(){

```

```

        System.out.println("Old Functionality");
    }
}
class B extends A{
    void m1(){
        System.out.println("New Functionality");
    }
}
public class Main {
    public static void main(String[] args) {
        A a = new B();
        a.m1();
    }
}

```

Status: Compilation Error, m1() has private access in A

2. In Method Overriding, the subclass method must have the same return type of the superclass otherwise the compiler will raise an error.

EX:

```

class A{
    void m1(){
        System.out.println("Old Functionality");
    }
}
class B extends A{
    void m1(){
        System.out.println("New Functionality");
    }
}
public class Main {
    public static void main(String[] args) {

```

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

Status: Valid
```

EX:

```
class A{
    void m1(){
        System.out.println("Old Functionality");
    }
}

class B extends A{
    int m1(){
        System.out.println("New Functionality");
        return 10;
    }
}

public class Main {
    public static void main(String[] args) {
        A a = new B();
        a.m1();
    }
}
```

```
Status: Compilation Error, m1() in B cannot override m1() in A
return type int is not compatible with void
```

EX:

```
class A{
    int m1(){
        System.out.println("Old Functionality");
        return 20;
    }
}
```

```

}

class B extends A{
    int m1(){
        System.out.println("New Functionality");
        return 10;
    }
}

public class Main {
    public static void main(String[] args) {
        A a = new B();
        a.m1();
    }
}

```

Status: No Compilation Error

OP: New Functionality

3. In Method Overriding, the superclass method must not be declared as final irrespective of the subclass method final.

EX:

```

class A{
    final void m1(){
        System.out.println("Old Functionality");
    }
}

class B extends A{
    void m1(){
        System.out.println("New Functionality");
    }
}

public class Main {

```

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

Status: Compilation Error

EX:

```
class A{
    final void m1(){
        System.out.println("Old Functionality");
    }
}
class B extends A{
    final void m1(){
        System.out.println("New Functionality");
    }
}
public class Main {
    public static void main(String[] args) {
        A a = new B();
        a.m1();
    }
}
```

Status: Compilation Error

EX:

```
class A{
    void m1(){
        System.out.println("Old Functionality");
    }
}
```

```

    }
}

class B extends A{
    final void m1(){
        System.out.println("New Functionality");
    }
}

public class Main {
    public static void main(String[] args) {
        A a = new B();
        a.m1();
    }
}

```

Status: No Compilation Error

OP: New Functionality

4. In Method overriding, either superclass method or subclass method or both superclass and subclass methods must not be declared as static, if either superclass method or subclass method is declared as static then the compiler will raise an error, if we declare both superclass method and subclass method as static then the compiler will not raise any error, but JVM will not perform Method overriding, where JVM will perform Method Over hiding, here if we access the superclass method then JVM will execute only superclass method, JVM will not execute the subclass method.

EX:

```

class A{
    static void m1(){
        System.out.println("Old Functionality");
    }
}

```

```

    }
}

class B extends A{
    void m1(){
        System.out.println("New Functionality");
    }
}

public class Main {
    public static void main(String[] args) {
        A a = new B();
        a.m1();
    }
}

```

Status: Compilation Error, m1() in B cannot override m1() in A
 overridden method is static

EX:

```

class A{
    void m1(){
        System.out.println("Old Functionality");
    }
}

class B extends A{
    static void m1(){
        System.out.println("New Functionality");
    }
}

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

```

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

Status: Compilation Error, m1() in B cannot override m1() in A
overriding method is static

EX:

```
class A{
    static void m1(){
        System.out.println("Old Functionality");
    }
}
class B extends A{
    static void m1(){
        System.out.println("New Functionality");
    }
}
public class Main {
    public static void main(String[] args) {
        A a = new B();
        a.m1();
    }
}
```

Status: No Compilation Error

OP: Old Functionality

5. In Method Overriding, the subclass method must have either the same scope of the superclass method scope or wider than the superclass method scope.

EX:

```
class A{
    public void m1(){
        System.out.println("Old Functionality");
    }
}
class B extends A{
    public void m1(){
        System.out.println("New Functionality");
    }
}
public class Main {
    public static void main(String[] args) {
        A a = new B();
        a.m1();
    }
}
```

Status: No Compilation Error

OP: New Functionality.

EX:

```
class A{
    public void m1(){
        System.out.println("Old Functionality");
    }
}
class B extends A{
    protected void m1(){
        System.out.println("New Functionality");
    }
}
```

```
    }
}

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("Old Functionality");
    }
}

class B extends A{
    public void m1(){
        System.out.println("New Functionality");
    }
}

public class Main {
    public static void main(String[] args) {
        A a = new B();
        a.m1();
    }
}
```

Status: No Compilation Error

OP: New Functionality

6. In Method Overriding, the subclass method must have either the same access privileges when compared with the superclass method access privileges or weaker access privileges than the superclass method access privileges.

7. In Method Overriding, the subclass method throws exception must be either the same as the superclass method throws exception or subclass to the superclass method throws Exception.

EX:

```
class A{
    public void m1()throws Exception{
        System.out.println("Old Functionality");
    }
}
class B extends A{
    public void m1() throws Exception{
        System.out.println("New Functionality");
    }
}
public class Main {
    public static void main(String[] args)throws Exception {
        A a = new B();
        a.m1();
    }
}
```

Status: No Compilation Error

OP: New functionality

EX:

```
import java.io.IOException;
```

```
class A{
    public void m1()throws Exception{
        System.out.println("Old Functionality");
    }
}
class B extends A{
    public void m1() throws IOException {
        System.out.println("New Functionality");
    }
}
public class Main {
    public static void main(String[] args) throws Exception {
        A a = new B();
        a.m1();
    }
}
```

Status: No Compilation Error

OP: New functionality

EX:

```
import java.io.IOException;
```

```
class A{
    public void m1()throws IOException{
        System.out.println("Old Functionality");
    }
}
class B extends A{
    public void m1() throws Exception {
```

```
        System.out.println("New Functionality");

    }

}

public class Main {
    public static void main(String[] args) throws Exception {
        A a = new B();
        a.m1();
    }
}
```

Status: Compilation Error, m1() in B cannot override m1() in A
overridden method does not throw java.lang.Exception

Q) What are the differences between Method Overloading and Method Overriding?

Ans:

1. The process of extending the existing method functionality to a new functionality is called Method Overloading.

The process of providing replacement for the existing method functionality with a new functionality is called Method Overriding.

2. In Method overloading, we have to declare more than one method with the same name and with the different parameter list that is different method signatures.

In Method Overriding, we have to declare more than one method with the same method prototype.

3. Method Overloading does not require Inheritance , it is possible to perform method overloading with or without the inheritance.

Method Overriding requires Inheritance, without the inheritance method overriding is not possible.

Consider the following program

```
class A{
    void m1(){
        System.out.println("Old Implementation Contains 1000
LOC");
    }
}
class B extends A{
    void m1(){
        System.out.println("New Implementation Contains 100 LOC");
    }
}
public class Main {
    public static void main(String[] args) throws Exception {
        B b = new B();
        b.m1();
    }
}
```

New Implementation Contains 100 LOC

In the above Method Overriding program, when we access the superclass method JVM executes the corresponding subclass method and provides output from the subclass method.

In the above Method Overriding program, when we access the superclass method, JVM executes the corresponding subclass method only, here JVM does not execute the respective superclass method implementation, so providing implementation for the superclass method is unnecessary, it is suggestible to remove the superclass method implementation.

In the above situation, in Method Overriding , in superclass we need a method with the declaration and without the implementation, in JAVA programming if we want to declare a method without the implementation then we have to declare that method as an abstract method.

In Java applications, if we want to declare a method as an abstract method in a class then the respective class must be an abstract class.

EX:

```
abstract class A{
    abstract void m1();
}

class B extends A{
    void m1(){
        System.out.println("New Implementation Contains 100 LOC");
    }
}

public class Main {
    public static void main(String[] args) throws Exception {
        A a = new B();
        a.m1();
    }
}
```

New Implementation Contains 100 LOC

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

Ans:

1. Concrete Method is a normal Java method, it must have both Method Declaration and Method Implementation.

Abstract Method is a Java method, it must have only Method Declaration without the Method implementation.

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

Abstract Methods are possible in the abstract classes and in the interfaces.

3. To declare concrete methods no special keyword is required.

To declare abstract methods we must use the “abstract” keyword.

4. Concrete methods will provide less shareability.

Abstract Methods will provide more shareability.

Concrete Method:

```
public void add(int l, int j) {  
    int result = l + j;  
    System.out.println(result);  
}
```

Method Declaration / Method Prototype / Method Header

Method Implementation / Method Body

Abstract Method:

```
public abstract void add(int l, int j);
```

Q) What are the differences between Concrete class and abstract class?

Ans:

1. Classes are able to allow only concrete methods.

Abstract classes are able to allow zero or more concrete methods and zero or more abstract methods.

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

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

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

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

4. Classes are able to provide less shareability.

Abstract classes are able to provide more shareability.

In Java applications, if we declare an abstract class with abstract methods then it is a convention to take a subclass for the abstract class and to provide implementation to all the abstract methods.

EX:

```
abstract class A{  
    abstract void m1();
```

```
abstract void m2();
abstract void m3();
}
class B extends A{
    void m1(){
        System.out.println("m1-B");
    }
    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 B();
        a.m1();
        a.m2();
        a.m3();
        //a.m4(); ---> Error

        B b = new B();
        b.m1();
        b.m2();
        b.m3();
        b.m4();
    }
}
```

m1-B

```
m2-B  
m3-B  
m1-B  
m2-B  
m3-B  
m4-B
```

In Java applications, if we declare an abstract class with abstract methods then it is a convention to provide implementation for all the abstract methods inside the subclass, in the case if we provide implementation for some of the abstract methods , not for all abstract methods then the compiler will raise errors, in this context to avoid the compilation errors we have to declare the respective subclass as an abstract class , take another class for the sub abstract class in multilevel inheritance and provide the remaining abstract methods implementation in the subclass.

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");  
    }  
}  
abstract class C extends B{  
    void m2(){  
        System.out.println("m2-C");  
    }  
}
```

```
class D extends C{
    void m3(){
        System.out.println("m3-D");
    }
}
public class Main {
    public static void main(String[] args){
        A a = new D();
        a.m1();
        a.m2();
        a.m3();
    }
}
```

m1-B
m2-C
m3-D

In Java applications, it is possible to declare an abstract class without the abstract methods and full of concrete methods.

In Java applications, to declare a class as an abstract class it is not mandatory to have at least one abstract method, but to declare a method as an abstract method then the respective class must be an abstract class.

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();
    }
}

```

m1-A
m2-A
m3-A

In Java applications, it is possible to extend an abstract class to the concrete classes and it is possible to extend 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();
    }
}

```

m1-A
 m1-A
 m2-B
 m3-C
 m4-C

In abstract classes we are able to provide constructors, but it is not possible to create objects for the abstract classes.

The main purpose of the constructors inside the abstract classes is to recognize the instance context of the abstract class and to store that instance context inside the subclass object

EX:

```
abstract class A{
    A(){
        System.out.println("A-Con");
    }
}
abstract 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();
    }
}
```

A-Con

B-Con

C-Con

EX:

```
abstract class A{
```

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

A-Con
C-Con

Interfaces:

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

Ans:

1. Classes are able to provide only concrete methods.

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

Interfaces are able to provide only abstract methods.

2. To declare classes only the “class” keyword is sufficient.

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

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

3. For classes we are able to create 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. Inside the interfaces, by default all variables are “public static final”.

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

5. Inside the interfaces, by default all methods are “public and abstract”.

No default cases exist for the methods inside 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.

7. In classes and abstract classes constructors are possible.

Interfaces do not allow constructors.

8. Classes and abstract classes are able to allow static blocks, instance blocks,..

INterfaces do not allow static blocks, instance blocks,....

9. Inside the interfaces, by default all inner classes are static inner classes.

No default cases exist from the inner classes inside the classes and abstract classes.

10. Interfaces are mainly for declaring the Services.

Classes are mainly for implementing Services which are defined by the interfaces.

Abstract classes are mediators between interface and classes , they are able to provide some services implementation and some other services declarations.

In java applications, if we declare an interface with abstract methods then it is a convention to provide an implementation class and to provide implementation for all the abstract methods.

If we declare a variable in an interface then we are able to access that variable by using

- a. Interface Name
- b. Interface reference variable
- c. Implementation class Name
- d. Implementation class reference variable

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");
    }
}
public class Main {
    public static void main(String[] args){
        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);
}

}
```

m1-A

m2-A

m3-A

m1-A

m2-A

m3-A

m4-A

10

10

10

10

In Java applications, if we declare an interface with abstract methods then we have to provide implementation for all the abstract methods inside the implementation class, in the case if we provide implementation for some of the abstract methods, not for all the abstract methods then the compiler will raise errors, in this context to avoid compilation errors we have to declare the implementation class as an abstract class and provide

implementation for the remaining abstract methods by taking a subclass to 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");
    }
}

abstract class B extends A{
    public void m2() {
        System.out.println("m2-B");
    }
}

class C extends B{
    public void m3() {
        System.out.println("m3-C");
    }
}

public class Main {
    public static void main(String[] args){
        I i = new C();
        i.m1();
        i.m2();
        i.m3();
    }
}
```

m1-A

m2-B

m3-C

In Java applications, it is possible to implement more than one interface in a single implementation class, here the single implementation class must provide implementation for all abstract methods of 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();

}

}
```

```
m1-A
m2-A
m3-A
m1-A
m2-A
m3-A
```

In Java applications, it is not possible to extend more than one class to a single subclass, but 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();
    }
}
```

m1-A

m2-A

m1-A

m2-A

m3-A

In general, in java applications, we will utilize the interfaces to declare services and we will utilize the implementation classes to provide implementations for the services.

EX:

```
interface FordCar{
    public void getCarDetails();
}

class FordFiesta implements FordCar{
    public void getCarDetails() {
        System.out.println("Ford Fiesta Details");
        System.out.println("-----");
        System.out.println("Name : Ford Fiesta");
        System.out.println("Seating : 3+1");
        System.out.println("Steering Type : Power Steering");
        System.out.println("Price : 7L");
    }
}

class EchoSport implements FordCar{
    public void getCarDetails() {
        System.out.println("Echo Sport Details");
        System.out.println("-----");
        System.out.println("Name : Echo Sport");
        System.out.println("Seating : 4+1");
        System.out.println("Steering Type : Effective Power
Steering");
        System.out.println("Price : 14L");
    }
}

class FordEndeavor implements FordCar{
    public void getCarDetails() {
        System.out.println("FordEndeavor Details");
        System.out.println("-----");
        System.out.println("Name : FordEndeavor");
        System.out.println("Seating : 7+1");
        System.out.println("Steering Type : Effective Power
Steering");
        System.out.println("Price : 45L");
    }
}
```

```
    }
}

public class Main {
    public static void main(String[] args){
        FordCar fordFiesta = new FordFiesta();
        fordFiesta.getCarDetails();
        System.out.println();

        FordCar echoSport = new EchoSport();
        echoSport.getCarDetails();
        System.out.println();

        FordCar fordEndeavor = new FordEndeavor();
        fordEndeavor.getCarDetails();
    }
}
```

Ford Fiesta Details

```
Name      : Ford Fiesta
Seating   : 3+1
Steering Type : Power Steering
Price     : 7L
```

Echo Sport Details

```
Name      : Echo Sport
Seating   : 4+1
Steering Type : Effective Power Steering
Price     : 14L
```

FordEndeavor Details

Name	:	FordEndeavor
Seating	:	7+1
Steering Type	:	Effective Power Steering
Price	:	45L

In general, in JDBC, Driver is an interface provided by SUN Microsystems with the services in the form of abstract methods and these services are implemented by the Database vendors in their own way by providing implementation classes for the Driver interface.

EX:

```
interface Driver{// SUN Microsystems
    public void registerDriver();
    public void connect();
}

class OracleDriver implements Driver{// Oracle Vendor
    public void registerDriver() {
        System.out.println("OracleDriver Registered with the JDBC
Application");
    }
    public void connect() {
        System.out.println("Connection Established between Java
application and the Oracle Database");
    }
}
class MySQLDriver implements Driver{ // MySQL Vendor
    public void registerDriver() {
        System.out.println("MySQLDriver Registered with the JDBC
Application");
    }
    public void connect() {
```

```
        System.out.println("Connection Established between Java
application and the MySQL Database");
    }
}
class PostgreeSQLDriver implements Driver{// PostgreSQL Vendor
    public void registerDriver() {
        System.out.println("PostgreeSQLDriver Registered with the
JDBC Application");
    }
    public void connect() {
        System.out.println("Connection Established between Java
application and the PostgreSQL Database");
    }
}
public class Main {// JDBC Application
    public static void main(String[] args){
        Driver oracleDriver = new OracleDriver();
        oracleDriver.registerDriver();
        oracleDriver.connect();
        System.out.println();

        Driver mysqlDriver = new MySQLDriver();
        mysqlDriver.registerDriver();
        mysqlDriver.connect();
        System.out.println();

        Driver postgresqlDriver = new PostgreeSQLDriver();
        postgresqlDriver.registerDriver();
        postgresqlDriver.connect();
    }
}
OracleDriver Registered with the JDBC Application
```

Connection Established between Java application and the Oracle Database

MySQLDriver Registered with the JDBC Application
Connection Established between Java application and the MySQL Database

PostgreeSQLDriver Registered with the JDBC Application
Connection Established between Java application and the PostgreeSQL Database

Marker Interfaces:

If we have an interface without the abstract methods then that interface is called a Marker Interface.

In Java applications, Marker interfaces are not having abstract methods but it will provide some abilities to the objects at runtime of the applications.

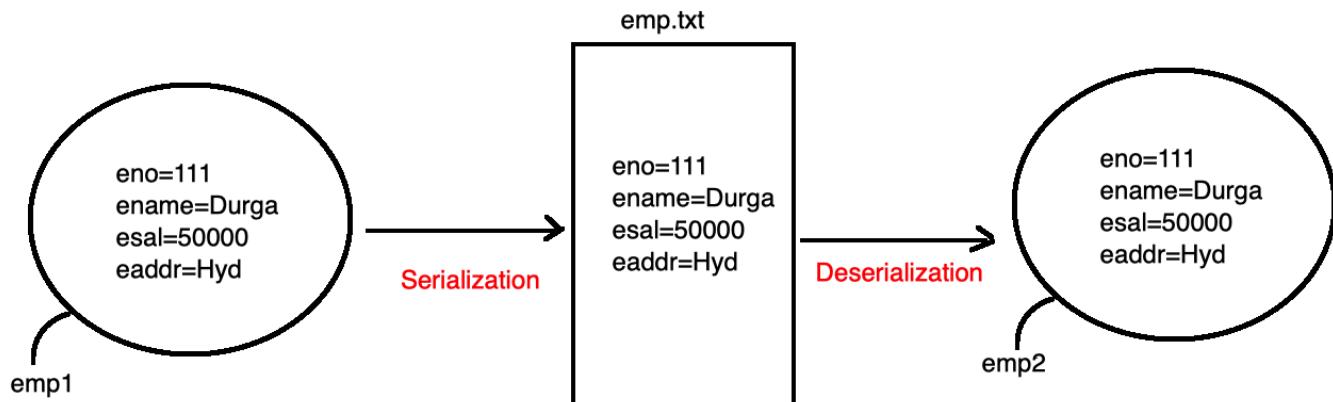
EX: `java.io.Serializable`, `java.lang.Cloneable`,....

`java.io.Serializable`:

`java.io.Serializable` marker interface can be used to perform Serialization and Deserialization over the objects in java applications.

The process of separating the data from an object is called Serialization.

The process of reconstructing an object on the basis of the data is called Deserialization.

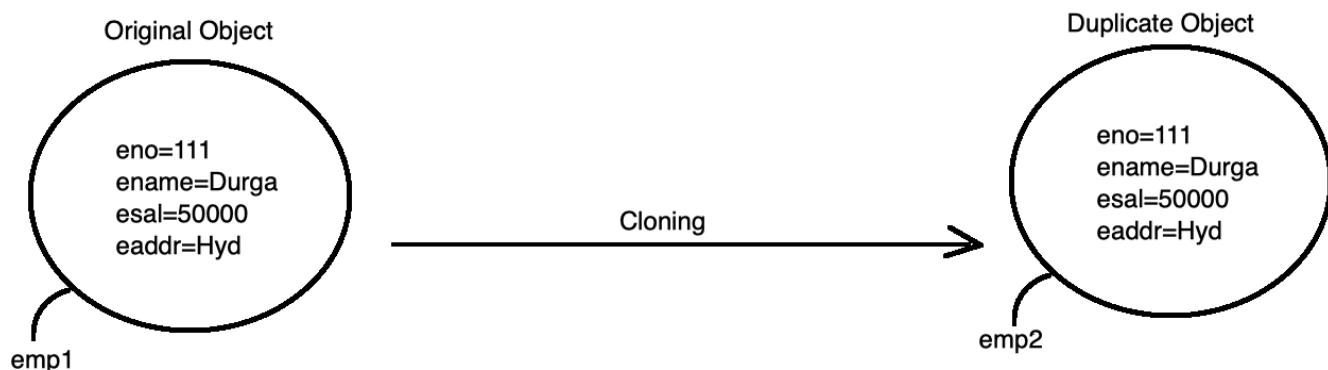


In Java, by default, all objects are not eligible for Serialization and Deserialization, only the objects whose classes are implementing `java.io.Serializable` marker interface are eligible for Serialization and Deserialization.

`java.lang.Cloneable`:

In Java applications, `java.lang.Cloneable` marker interface can be used to perform Object Cloning.

The process of generating a duplicate object from the original object is called Object Cloning.



IN Java, by default, all the 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 Java applications, we will declare interfaces with abstract methods , here we have to implement all the abstract methods in the implementation classes with or without the actual application requirement, this approach will increase unnecessary methods implementations in java applications.

In the above context, Adapter classes 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 each and every method.
3. At the implementation classes, extend the Mediator class instead of implementing the interface and override the required method.
4. Declare the mediator class as an abstract class in order to avoid object creation for the mediator class and giving an option to the developers to create objects for the implementation classes which are having business logic.
5. If we have any method common to all the implementation classes with variable implementation then it is suggestible to declare that method as an abstract method in the mediator class.

In the above solution, the mediator which we have provided between interface and implementation classes is called “Adapter Class”.

```
interface I{
    void m1();
    -----
    -----
    void m50();
}
```

```
abstract class M implements I{
    p v m1(){. }
    -----
    abstract p v m25();
    -----
    p v m5(){. }
}
```

Adapter class
Or
Generic Class

```
extends M
class C1 implements I{
    p v m1(){ ---}
    P v m25(){
        --X-impl--
    }
}
```

```
extends M
class C2 implements I{
    p v m2(){ ---}
    P v m25(){
        --Y-impl--
    }
}
```

```
extends M
class C10 implements I{
    p v m10(){ ---}
    P v m25(){
        --Z-impl--
    }
}
```

EX: WindowAdapter, MouseAdapter, KeyAdapter, ...

```
interface WindowListener{  
    p v windowOpened(WindowEvent we);  
    p v windowClosed(WindowEvent we);  
    p v windowClosing(WindowEvent we);  
    p v windowActivated(WindowEvent we);  
    p v windowDeactivated(WindowEvent we);  
    p v windowIconified(WindowEvent we);  
    p v windowDeiconified(WindowEvent we);  
}
```

implements

```
abstract class WindowAdapter implements WindowListener{  
    p v windowOpened(WindowEvent we){ }  
    p v windowClosed(WindowEvent we){ }  
    p v windowClosing(WindowEvent we){ }  
    p v windowActivated(WindowEvent we){ }  
    p v windowDeactivated(WindowEvent we){ }  
    p v windowIconified(WindowEvent we){ }  
    p v windowDeiconified(WindowEvent we){ }  
}
```

extends

```
class WindowListenerImpl extends WindowAdapter{  
    public void windowClosing(WindowEvent we){  
        System.exit(0);  
    }  
}
```

Object Cloning:

The process of generating a duplicate object from the original object is called Object Cloning.

To perform Object cloning we have to use the following steps.

1. Declare an user defined class and implement java.lang.Cloneable marker interface.

The main purpose of implementing java.lang.Cloneable marker interface is to make an object eligible for Object cloning, because in java applications by default all the objects are not eligible for object cloning, only the object whose

classes are implementing `java.lang.Cloneable` marker interface are eligible for Object cloning.

2. Override `java.lang.Object` class provided `clone()` method in the user defined class, where access to the superclass provided `clone()` method.

EX:

```
public class Student implements Cloneable{  
    -----  
    public Object clone()throws CloneNotSupportedException{  
        Object duplicate = super.clone();  
        return duplicate;  
    }  
    -----  
}
```

3. In the main class and in the `main()` method, create an object for the user defined class that is the original object and access the `clone()` method to get the duplicate object.

```
Student originalObject = new Student();  
Student duplicateObject = (Student)originalObject.clone();
```

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 duplicate = super.clone();
    return duplicate;
}

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 Exception {
        Student original = new Student("S123", "Durga", "Hyd");
        System.out.println("Original Student Details");
        original.getStudentDetails();
        System.out.println("Original Student Ref  : "+original);
        System.out.println();

        Student duplicate = (Student) original.clone();
        System.out.println("Duplicate Student Details");
        duplicate.getStudentDetails();
        System.out.println("Duplicate Student Ref  : "+duplicate);
    }
}

```

Original Student Details
 Student Details

Student Id : S123
Student Name : Durga
Student Address : Hyd
Original Student Ref : Student@30f39991

Duplicate Student Details

Student Details

Student Id : S123
Student Name : Durga
Student Address : Hyd
Duplicate Student Ref : Student@452b3a41

In Java applications, Object cloning is possible in the following two ways.

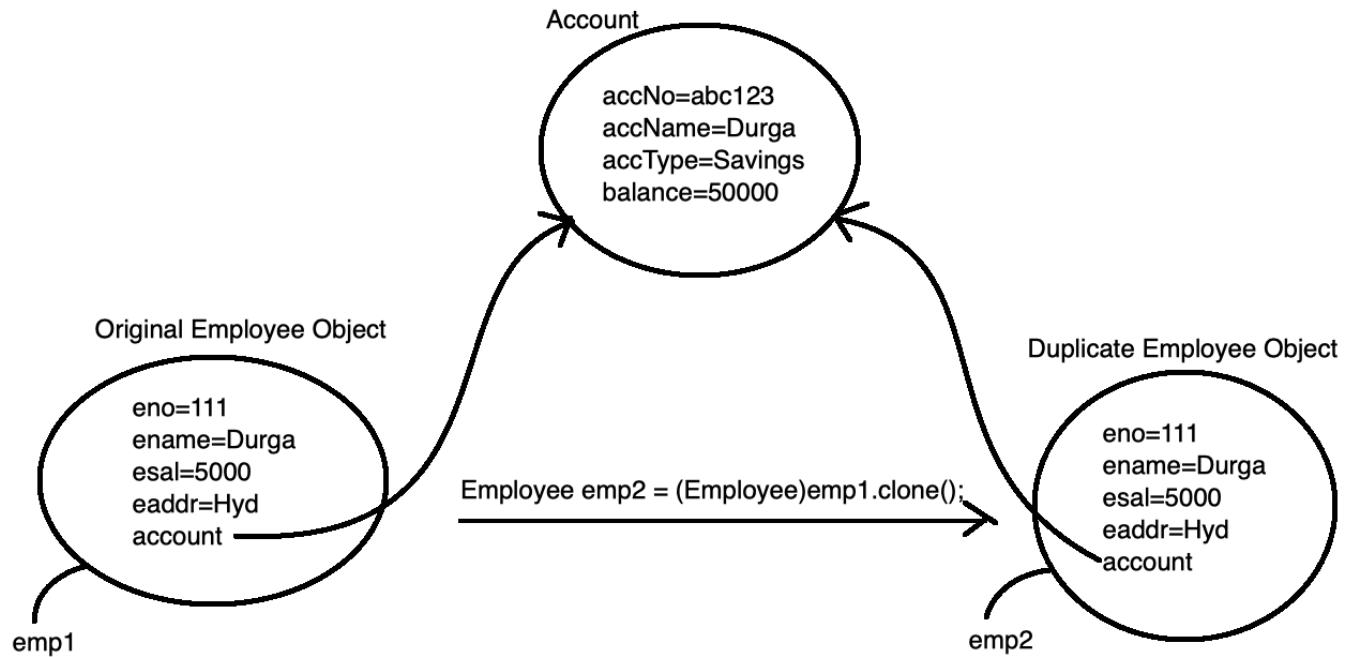
1. Shallow Cloning
2. Deep Cloning

Q)What is the difference between Shallow Cloning and Deep Cloning?

Ans:

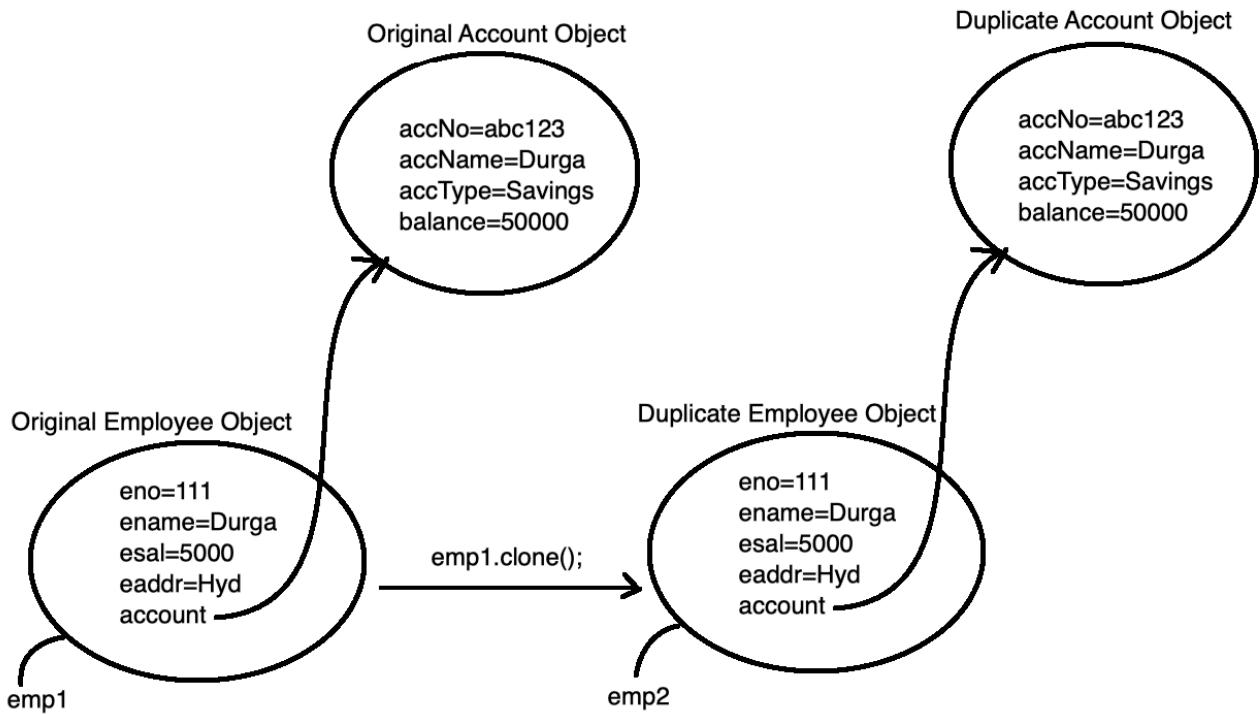
Shallow Cloning is the default cloning mechanism provided by the JAVA programming language , where the developers are not required to prepare cloning logic explicitly.

In java applications, if we have association between the objects and if we clone the container object JVM will perform cloning over the Container object only, JVM will not perform Cloning over the Contained object.



Deep Cloning is not the default cloning mechanism, it is an explicit cloning mechanism, where developers have to define the cloning logic explicitly.

In Java applications, if we have association between objects and if we clone the container object then JVM will perform cloning over the Contained object automatically along with the Container cloning.



EX On Shallow Cloning:

```

class Account{
    private String accNo;
    private String accName;
    private String accType;
    private long accBalance;

    public Account(String accNo, String accName, String accType,
long accBalance) {
        this.accNo = accNo;
        this.accName = accName;
        this.accType = accType;
        this.accBalance = accBalance;
    }

    public String getAccNo() {
        return accNo;
    }
}

```

```
public String getAccName() {
    return accName;
}

public String getAccType() {
    return accType;
}

public long getAccBalance() {
    return accBalance;
}
}

class Employee implements Cloneable{

private int eno;
private String ename;
private float esal;
private String eaddr;
private 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 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 Name       : "+account.getAccName());
    System.out.println("Account Type       : "+account.getAccType());
    System.out.println("Account Balance    : "+account.getAccBalance());
}

@Override
public Object clone() throws CloneNotSupportedException {
    return super.clone();
}
}

public class Main {
    public static void main(String[] args) throws Exception {
        Account account = new Account("abc123", "Durga",
"Savings", 50000);
        Employee emp1 = new Employee(111, "Durga", 5000, "Hyd",
account);
        System.out.println("Employee Details from Original
Object");
        emp1.getEmployeeDetails();
```

```

        System.out.println("Original Employee Object Ref : "+emp1);
        System.out.println("Original Account Object Ref : "+emp1.getAccount());

        Employee emp2 = (Employee) emp1.clone();

        System.out.println();
        System.out.println("Employee Details from Duplicate Object");
        emp2.getEmployeeDetails();
        System.out.println("Duplicate Employee Object Ref : "+emp2);
        System.out.println("Duplicate Account Object Ref : "+emp2.getAccount());

    }
}

```

Employee Details from Original Object
 Employee Details

Employee Number : 111
 Employee Name : Durga
 Employee Salary : 5000.0
 Employee Address : Hyd
 Account Details

Account Number : abc123
 Account Name : Durga
 Account Type : Savings
 Account Balance : 50000
 Original Employee Object Ref : Employee@681a9515
 Original Account Object Ref : Account@3af49f1c

Employee Details from Duplicate Object

Employee Details

Employee Number : 111
Employee Name : Durga
Employee Salary : 5000.0
Employee Address : Hyd

Account Details

Account Number : abc123
Account Name : Durga
Account Type : Savings
Account Balance : 50000
Duplicate Employee Object Ref : Employee@19469ea2
Duplicate Account Object Ref : Account@3af49f1c

EX on Deep Cloning:

```
class Account{
    private String accNo;
    private String accName;
    private String accType;
    private long accBalance;

    public Account(String accNo, String accName, String accType,
long accBalance) {
        this.accNo = accNo;
        this.accName = accName;
        this.accType = accType;
        this.accBalance = accBalance;
    }

    public String getAccNo() {
```

```
        return accNo;
    }

    public String getAccName() {
        return accName;
    }

    public String getAccType() {
        return accType;
    }

    public long getAccBalance() {
        return accBalance;
    }
}

class Employee implements Cloneable{

    private int eno;
    private String ename;
    private float esal;
    private String eaddr;
    private 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 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 Name         : "+account.getAccName());
    System.out.println("Account Type         : "+account.getAccType());
    System.out.println("Account Balance      : "+account.getAccBalance());
}
```

```
@Override
```

```
public Object clone() throws CloneNotSupportedException {
    Account duplicateAccount = new Account(
        account.getAccNo(),
        account.getAccName(),
        account.getAccType(),
        account.getAccBalance()
    );
```

```
    Employee duplicateEmployee = new Employee(
        this.eno,
        this.ename,
        this.esal,
```

```
        this.eaddr,
        duplicateAccount
    );
    return duplicateEmployee;
}
}

public class Main {
    public static void main(String[] args) throws Exception {
        Account account = new Account("abc123", "Durga",
"Savings", 50000);
        Employee emp1 = new Employee(111, "Durga", 5000, "Hyd",
account);
        System.out.println("Employee Details from Original
Object");
        emp1.getEmployeeDetails();
        System.out.println("Original Employee Object Ref : "
+emp1);
        System.out.println("Original Account Object Ref : "
+emp1.getAccount());

        Employee emp2 = (Employee) emp1.clone();

        System.out.println();
        System.out.println("Employee Details from Duplicate
Object");
        emp2.getEmployeeDetails();
        System.out.println("Duplicate Employee Object Ref : "
+emp2);
        System.out.println("Duplicate Account Object Ref : "
+emp2.getAccount());
    }
}
```

Employee Details from Original Object

Employee Details

```
-----  
Employee Number      : 111  
Employee Name        : Durga  
Employee Salary      : 5000.0  
Employee Address     : Hyd
```

Account Details

```
-----  
Account Number       : abc123  
Account Name         : Durga  
Account Type         : Savings  
Account Balance      : 50000  
Original Employee Object Ref : Employee@681a9515  
Original Account Object Ref : Account@3af49f1c
```

Employee Details from Duplicate Object

Employee Details

```
-----  
Employee Number      : 111  
Employee Name        : Durga  
Employee Salary      : 5000.0  
Employee Address     : Hyd
```

Account Details

```
-----  
Account Number       : abc123  
Account Name         : Durga  
Account Type         : Savings  
Account Balance      : 50000  
Duplicate Employee Object Ref : Employee@19469ea2  
Duplicate Account Object Ref : Account@13221655
```

instanceof operator:

```
-----
```

'instanceof' is a boolean operator, it will check whether a reference variable contains an instance of the specified class or not.

Syntax:

```
refVar instanceof ClassName
```

1. If the refVar class name is the same as the provided className then the instanceof operator will return true value.
2. If the refVar class name is subclass to the provided class name then the instanceof operator will return true value.
3. If the refVar class name is superclass to the provided className then the instanceof operator will return false value.
4. If the refVar class name is not related with the provided className then the compiler will raise an 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
}

true
true
true
false
true
```
