

Class Notes @ Core Java 8:30AM : <https://tinyurl.com/java830amnotes>
Class Notes @ Core Java 10:00AM : <https://tinyurl.com/corejava10am>
Workshop Link: <https://attendee.gotowebinar.com/register/2341171852295977311>
Google Classroom Code: ug7ihlx
Mobile No: 7386095600

Object Orientation[OOPS]:

To prepare Applications we will use the following 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. Unstructured Programming languages are outdated programming languages, they are not suitable to the present application requirements.
EX: BASIC, FORTRAN

Structured Programming Languages are not outdated programming languages, they are suitable for the present application requirements.
EX: C, PASCAL,...

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

Structured Programming Languages have a structure to prepare applications.

3. -Unstructured programming Language uses Mnemonic codes, there are low level codes, they are available in very less number and they will provide less features to the application development, so application development is very much difficult.

Structured Programming Languages are using High level syntaxes, they are available in more numbers and they will provide more features to

the application development, so the application development is very simple.

4. Unstructured programming languages are using only goto statements to define flow of execution.

Structured Programming Languages are using more flow controllers to define a very good flow of execution, in general structured programming languages are using flow controllers like goto, if, switch, for, while, do-while, break, continue,....

5. Unstructured Programming Languages are not having functions, so code redundancy will be increased, it will increase the length of the program.

In Structured Programming languages, functions are supported, it will reduce code redundancy, it will increase code reusability and it will reduce length of the program.

EX

```
void task(){
    -----
    -----Task[100 loc]-----
    -----
}

void main(){

    task();

    task();

    task();

}
```

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 applications.

Object Oriented Programming Languages are providing a simple approach to prepare applications.

2. Modularization is not good in Structured programming languages.

Modularization is very good in Object Oriented Programming Languages.

3. Abstraction is not good in the Structured programming languages.

Abstraction is very good in the Object Oriented Programming Languages.

4. Security is not good in the Structured programming languages.

Security is very good in the Object Oriented Programming Languages.

5. Shareability is not good in the Structured Programming Languages.

Shareability is very good in the Object Oriented Programming Languages.

6. Reusability is not good in the Structured Programming Languages.

Reusability is very good in the Object Oriented Programming languages.

Aspect Oriented Programming Languages:

Aspect Oriented Programming Languages is a mis terminology, because no programming language exists on the basis of the Aspect Orientation.

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

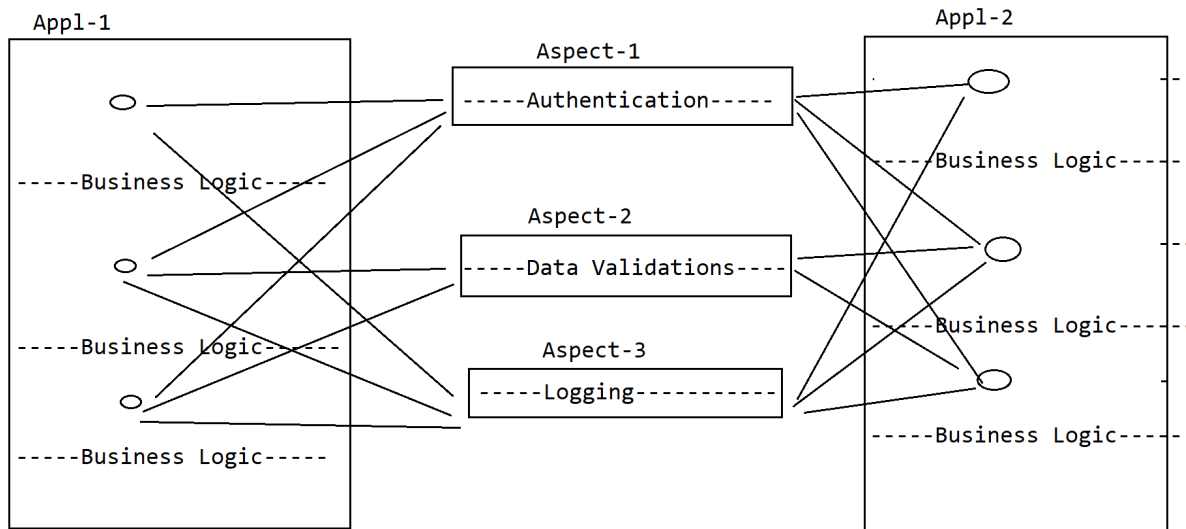
In Object Orientation, we will prepare the applications by combining both Services logic and business logic , this approach will not provide reusability and shareability.

In Object Orientation , to improve reusability and Shareability we have to use Aspect Orientation.

In Aspect Orientation,

1. Separate the Services logic from the Business logic.
2. Declare each and every service as an aspect.

3. Provide the aspects to the applications at runtime as per the requirement.



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 allow almost all the object oriented features including "Inheritance".

EX: Java

Object Based Programming Languages allow almost all the object oriented features excluding "Inheritance".

EX: Java script, SNOBOL.

Q) What are the differences between class and object?

Ans:

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

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

2. Class is Virtual.

Object is real or physical.

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

Object is the physical encapsulation of the properties and behaviors.

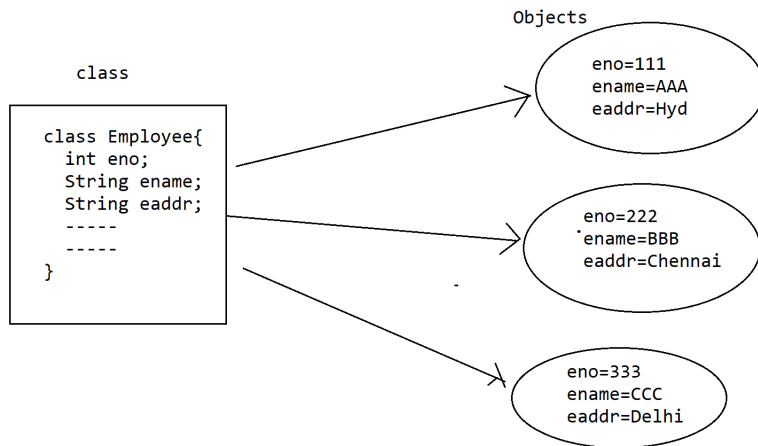
4. Class is a Generalization.

Object is the specialization.

5. Class is a model or blueprint for all the objects.

Object is an instance of the class.

EX:



Q)What is the difference between Encapsulation and Abstraction?

Ans:

Encapsulation is the process of binding data and coding parts.

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

The main Advantage of the ENcapsulation and Abstraction is "Security".

Security = Encapsulation + Abstraction

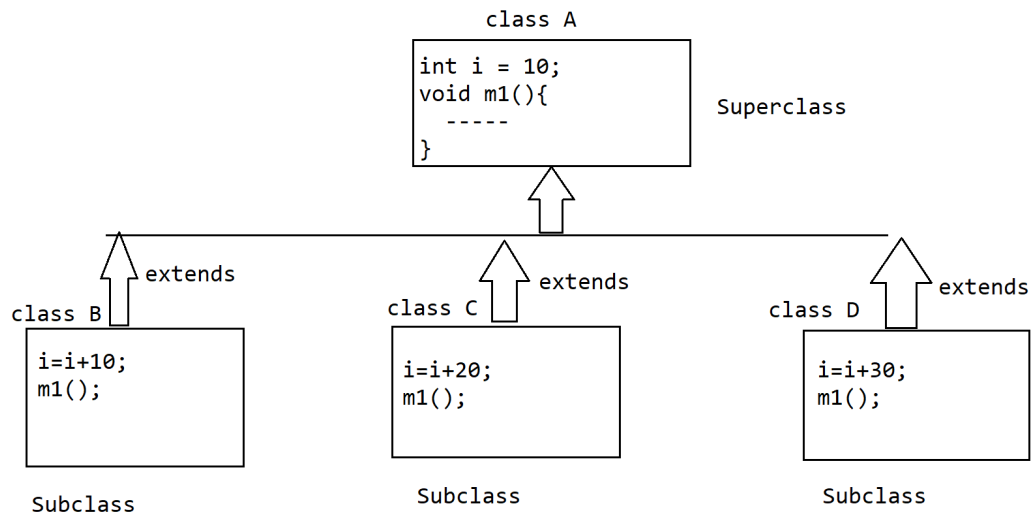
Inheritance:

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

In inheritance the class which is providing variables and methods to some other class is called a superclass or base class or Parent class.

In Inheritance the class which is getting variables and methods from superclass is called a subclass or a derived class or child class.

The main advantage of the inheritance is "Code Reusability".



Polymorphism:

Polymorphism is a GREEK word, where poly means many and morphism means structures or forms.

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

The main advantage of the Polymorphism is “Flexibility” in the application development.

Message Passing:

The process of passing data along with the flow of execution from one instruction to another instruction is called Message passing.

The main advantage of the Message Passing is

1. Improving communication between entity classes.
2. Improving Data Navigation between entity classes.

Containers:

Container is a topmost Java element, it contains some other Java 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 classes is to represent all real world entities in Java programming.

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

To represent entities data we have to use “variables” inside the classes.

To represent behaviors or actions or activities of an entity we will use “Methods” inside the classes.

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

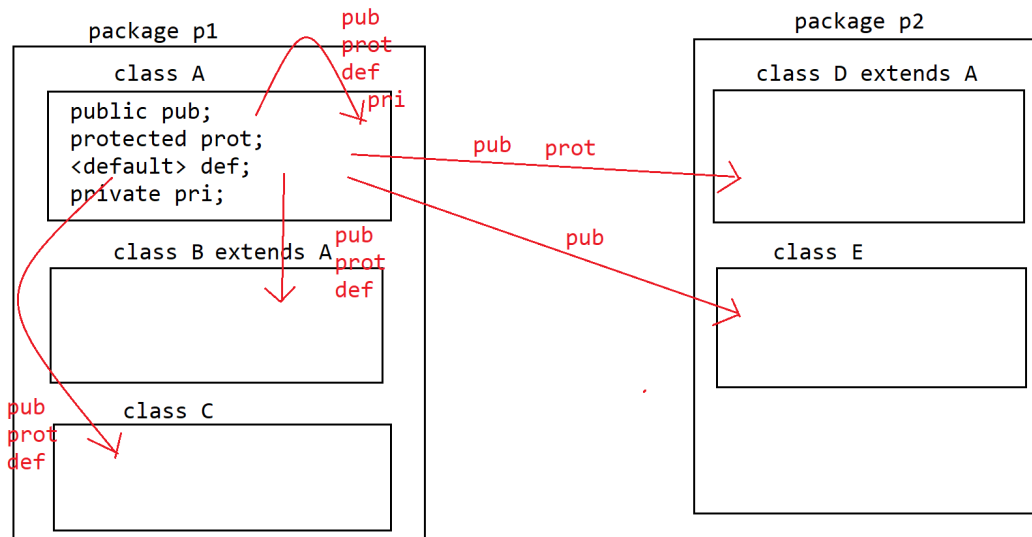
```
[Access Modifiers] class ClassName [extends SuperClass] [implements
intrerfaceList]{
    ---Variables-----
    ---- Methods-----
    ----- constructors-----
    ----- blocks -----
    ----- classes -----
    ----- abstract classes-----
    ----- interfaces-----
    ----- enums -----
}
```

Access Modifiers in Java:

To define scopes and some extra nature to the programming elements like variables, methods, classes,... we will use Access Modifiers.

There are two types of Access Modifiers.

1. To define Scopes to the programming elements like variables , methods, classes,... we will use the following access modifiers.
 - a. private
 - b. <default>
 - c. protected
 - d. public



Private members are having scope up to the respective class.

<default> members are having scope up to the current package.

Protected members are having scope throughout the present package and the subclasses existed in the other packages.

Public members are having scope throughout the application.

Note: To declare the elements with the scopes like public, protected and private Java has provided predefined access modifiers like public, protected and private, but to declare default scope Java has not provided any access modifier , but to provide default scope for any element just declare the elements without using public, protected and private.

In Java , classes allow the access modifiers public and <default>, classes do not allow the access modifiers private and protected.

EX:

```
public class Test{  
}
```

Status: Valid

EX:

```
protected class Test{  
}
```

Status: Invalid

EX:

```
class Test{  
}
```

Status: Valid

EX:

```
private class Test{  
}
```

Status: Invalid.

In Java applications, it is possible to provide public, protected, <default> and private for the inner classes.

EX:

```
class A{  
    public class B{  
    }  
    protected class C{  
    }  
    class D{  
    }  
    private class E{  
    }  
}
```

Status: Valid

Q)Why is private access modifier not allowed for the outer classes and why is it allowed for the inner classes?

Ans:

If we define any access modifier on the origin or boundaries of the classes then that access modifier is not applicable for the classes, but it is applicable for the members of the classes including inner classes.

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

```
class A{
    private class B{
    }
}
```

```
class A{
    protected class B{
    }
}
```

2. To define some extra nature to the programming elements like variables, methods, classes,... we will use the following access modifiers.

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

From the above list of access modifiers , classes are able to allow the access modifiers like abstract, final, strictfp.

EX-1:

```
Static class A{
}
```

Status: Invalid

EX-2:

```
abstract class A{
}
```

Status: Valid

EX-3:

```
final class A{
}
```

Status: Valid

EX-4:

```
native class A{
}
```

Status: Invalid

From the above list of access modifiers , inner classes are able to allow the access modifiers like static, abstract, final and strictfp.

EX-1:

```
class A{
    static class B{
    }
}
```

Status: Valid

EX-2:

```
class A{
    abstract class B{
    }
}
```

Status: Valid

```
class A{
    transient class B{
    }
}
```

Status: Invalid

Note: 'static' keyword is defined on the origin of the classes, so it is not applicable for the classes and it is applicable for the inner classes.

'class' is a Java keyword, it is able to represent 'Class' object oriented features.

'className' is an identity for the class in order to recognize the class individually.

'extends' is a java keyword, it can be used to provide a particular superclass name in the current class syntax.

Note: In Class syntax, 'extends' keyword is able to allow only one superclass name, it will not allow more than one superclass name.

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

Status: Valid

```
class A extends B,C{  
}
```

Status: Invalid

'implements' is a Java keyword, it is able to represent interfaces names in the current class syntax.

Note: extends keyword is able to allow only one superclass class , but implements keyword is able to allow one or more interfaces in the class syntax.

```
class A implements I{  
}
```

Status: Valid

```
class A implements I1, I2{  
}
```

Status: Valid

Note: In the class Syntax, both extends and implements keyword are optional,

1. we can Write a class without the extends keyword and with the implements keyword.
2. we can Write a class with the extends keyword and without the implements keyword.
3. we can Write a class without the extends and implements keywords.
4. We can Write a class with both extends and implements keywords, but in this context first we must write extends keyword then we must write implements keyword, we must not interchange the extends and implements keywords..

Q)Find the valid syntaxes from the following list?

-
1. public class A{ } -----> Valid
 2. protected class A{ } -----> Invalid
 3. class A{ } -----> Valid
 4. private class A{ } -----> Invalid
 5. class A{ private class B{ } } -----> Valid
 6. class A{ protected class B{ } } -----> Valid
 7. class A{ public class B{ } } -----> Valid
 8. static class A{ } -----> Invalid
 9. abstract class A{ } -----> Valid
 10. native class A{ } -----> Invalid
 11. synchronized class A{ } -----> Invalid

```

12.  strictfp class A{    } -----> Valid
13.  class A{    native class B{    }    } -----> Invalid
14.  class A{    abstract class B{    }    } -----> Valid
15.  class A{    transient class B{    }    } -----> Invalid
16.  class A{    strictfp class B{    }    } -----> Valid
17.  class A{    static class B{    }    } -----> Valid
18.  class A extends B{    } -----> Valid
19.  class A extends B, C{    } -----> Invalid
20.  class A implements I{    } -----> Valid
21.  class A implements I1, I2{    } -----> Valid
22.  class A extends B implements I{    } -----> Valid
23.  class A extends B implements I1, I2{    } -----> Valid
24.  class A implements I extends B{    } -----> Invalid
25.  class A extends A{    } -----> Invalid
26.  class A extends B{    }
      class B extends A{    } -----> Invalid

```

Procedure To Use classes in Java applications:

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

EX:

File Name: D:\FullstackJava830\JAVA830\Test.java

```
class Employee{
```

```

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

    public void displayEmployeeDetails(){
        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);
    }

}

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

```

D:\FullstackJava830\JAVA830>javac Test.java

D:\FullstackJava830\JAVA830>java Test
Employee Details

```

-----
Employee Number   : 111
Employee Name     : Durga
Employee Salary   : 50000.0
Employee Address  : Hyd

```

EX:

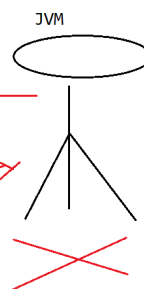
```

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

    public void displayEmployeeDetails(){
        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);
    }
}

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

```



CMD

```

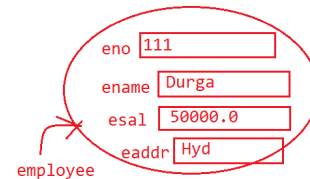
D:\java830>javac Test.java
D:\java830>java Test

```

```

Employee Details
-----
Employee Number   : 111
Employee Name     : Durga
Employee Salary   : 50000.0
Employee Address  : Hyd

```



EX:

File Name: D:\FullstackJava830\JAVA830\Test.java

```
class Student{

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

    public void displayStudentDetails(){
        System.out.println("Student Details");
        System.out.println("-----");
        System.out.println("Student Id      : "+sid);
        System.out.println("Student Name    : "+sname);
        System.out.println("Student Qual    : "+squal);
        System.out.println("Student Address : "+saddr);
    }

}

class Customer{

    String cid = "C-111";
    String cname = "Durga";
    String caddr = "Hyd";
    String cemail = "durga@dss.com";
    String cmobile = "91-9988776655";

    public void displayCustomerDetails(){
        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);
        System.out.println("Customer Email Id : "+ceemail);
        System.out.println("Customer Mobile No: "+cmobile);
    }

}

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



```

        System.out.println();

        Customer customer = new Customer();
        customer.displayCustomerDetails();

    }
}

```

D:\FullstackJava830\JAVA830>javac Test.java

D:\FullstackJava830\JAVA830>java Test

Student Details

```

-----
Student Id      : S-111
Student Name    : Durga
Student Qual    : BTech
Student Address : Hyd

```

Customer Details

```

-----
Customer Id      : C-111
Customer Name    : Durga
Customer Address : Hyd
Customer Email Id : durga@dss.com
Customer Mobile No: 91-9988776655

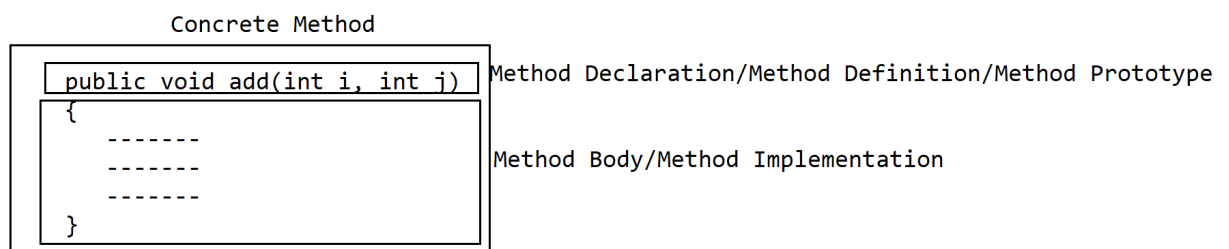
```

D:\FullstackJava830\JAVA830>

In Java there are two types of methods.

1. Concrete Methods:

Concrete method is a Java method, it will have both method declaration and method implementation.



2. Abstract Methods:

Abstract method is a Java method, it will have only method declaration without the method implementation.

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

Abstract Method

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

Method Declaration/ Method Prototype

Abstract Classes:

Abstract class is a Java class, it is able to allow both the concrete methods and abstract methods.

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

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

In Java applications, abstract classes are able to provide more shareability when compared with the classes.

Steps to use Abstract classes in Java applications:

1. Declare an abstract class by using the “abstract” keyword along with the “class” keyword.
2. Declare variables and methods[concrete Methods and abstract methods] inside the abstract class as per the application requirement.
3. Declare a subclass for the abstract class and provide implementation for all the abstract methods.
4. Declare the main class with the main() method.
5. Inside the main() method, create objects for the subclass and declare reference variables either for the abstract class or for the subclass.
6. Access the members of the abstract class by using the generated reference variable.

Note: If we declare reference variables for the abstract class then we are able to access only abstract class members, but if we declare reference variables for the subclass then we are able to access both abstract class 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");
    }
}
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
        System.out.println();

        B b = new B();
        b.m1();
        b.m2();
        b.m3();
        b.m4();
    }
}
```

D:\FullstackJava830\JAVA830>javac Test.java

D:\FullstackJava830\JAVA830>java Test

m1-A

m2-B

m3-B

m1-A
m2-B
m3-B
m4-B

Q)What are the differences between concrete classes and abstract classes?

Ans:

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

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

2. To declare a concrete class only “class” keyword is sufficient.

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

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

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

4. Concrete classes are able to provide less shareability.

Abstract classes are able to provide more shareability.

Interfaces:

Interface is a Java feature, it is able to allow only abstract methods.

To declare an interface we have to use a separate keyword in the form of “interface”.

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

Inside the interfaces, by default all variables are “public static final”, no need to declare explicitly.

Inside the interfaces, by default all the methods are “public and abstract”, no need to declare explicitly.

Steps to use interfaces in the Java applications:

1. Declare an interface by using the “interface” keyword.
2. Declare the variables and methods inside the interface.
3. Declare an implementation class[Same power of the subclass] for the interface.
4. Provide implementation for all the abstract methods of the interface.
5. Declare Main class with main() method.
6. Inside the main() method, create an object for the implementation class and declare a reference variable either for the interface or for the implementation class.
7. Access the interface members.

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

Note: If we declare a variable inside an interface then we are able to access that variable in the following four ways.

1. By Using Interface Name.
2. By Using the Interface reference variable.
3. By Using Implementation class Name.
4. By Using the 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");
        }
    }
    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);

        }
    }
}

```

D:\FullstackJava830\JAVA830>javac Test.java

D:\FullstackJava830\JAVA830>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 class, abstract class and interface?

Ans:

1. Classes are able to allow only concrete methods.

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

Interfaces are able to allow only abstract methods.

Note: From Java 8 version onwards interfaces are able to allow default methods and static methods with the implementations along with the abstract methods.

Note: From Java 9 version onwards interfaces are able to allow private methods with the implementations along with the abstract methods.

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

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

To declare interfaces we have to use the "interface" keyword.

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

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

4. Inside the interfaces, by default all variables are "public static final".

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

5. INside the interfaces, by default all methods are "public abstract".

Inside the classes and abstract classes, no default cases exist for the methods.

6. Inside the interfaces, by default all inner classes are “static inner classes”.

Inside the classes and abstract classes, no default cases exist for the inner classes.

7. Classes and abstract classes are able to allow constructors.

Interfaces are not allowing constructors.

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

Interfaces do not allow static blocks and instance blocks.

9. Classes are able to provide less shareability.

Abstract classes are able to provide middle level shareability.

Interfaces are able to provide more shareability.

10. In general, in Java applications, we will use interfaces to declare services only.

In general, in Java applications, we will use Classes to implement the services which are declared by the interfaces.

In general, in Java applications, we will use abstract classes to declare some services and to implement some other services.

Methods In Java:

Method is a set of instructions representing a particular behavior or an action of an entity.

EX-1: Transaction

```
====> deposit
====> withdraw
====> transferFunds
```


EX-2: Account:

```
==> createAccount  
==> updateAccount  
==> deleteAccount
```

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

```
[Access_Modifiers] ReturnType methodName([ParamList])[throws ExceptionList]{  
    -----instructions-----  
}
```

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

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

Where the main purpose of the “return type” is to specify the type of data which we are returning from the present method.

Java is able to allow all the primitive data types[byte, short, int, long, float, double, char, boolean] and all user defined data types[classes, abstract classes, interfaces, arrays, enums,...] and void as return type to the methods.

Where “void” return type is representing no data exists to return from the method, so no need to use return statement inside the method.

Note: In Java methods, if we provide any return type other than ‘void’ then it is mandatory to use “return” statement to return a value, if we use void as return type then we must not use return statement.

Where “methodName” is an identity for the method in order to access that method.

Where “paramList” is the list of parameters to provide input data to the method in order to perform the required action.

Where “throws” is a Java keyword, it is able to bypass an exception from the present method to the caller method in order to handle the exception.

In Java applications, if we want to access methods of any class then we have to create objects for the respective class and we have to use the generated reference variables.

EX:

```
class Account{

    String accountNumber = null;
    String accountHolderName = null;
    String accountType = null;
    int accountBalance = 0;

    public void createAccount(String accNo, String accHolderName, String
accType, int balance){
        if(accountNumber == accNo){
            System.out.println("Account "+accNo+" Existed Already");
        }else{
            accountNumber = accNo;
            accountHolderName = accHolderName;
            accountType = accType;
            accountBalance = balance;
            System.out.println("Account "+accNo+" Created
Successfully");
        }
    }

    public void searchAccount(String accNo){
        if(accountNumber != accNo){
            System.out.println("Account "+accNo+" Does Not Exist");
        }else{
            System.out.println("Account "+accNo+" Existed");
            System.out.println("Account Details");
            System.out.println("-----");
            System.out.println("Account Number      :
"+accountNumber);
            System.out.println("Account Holder Name    :
"+accountHolderName);
            System.out.println("Account Type          : "+accountType);
            System.out.println("Account Balance      :
"+accountBalance);
        }
    }
}
```

```

        public void updateAccount(String accNo, String accHolderName, String
accType, int balance){
            if(accountNumber == accNo){
                accountHolderName = accHolderName;
                accountType = accType;
                accountBalance = balance;
                System.out.println("Account "+accNo+" Updated
Successfully");
            }
            else{
                System.out.println("Account "+accNo+" Does Not Exist");
            }
        }

        public void deleteAccount(String accNo){
            if(accountNumber == accNo){
                accountNumber = null;
                accountHolderName = null;
                accountType = null;
                accountBalance = 0;
                System.out.println("Account "+accNo+" Deleted
Successfully");
            }
            else{
                System.out.println("Account "+accNo+" Does Not Exist");
            }
        }
    }

    class Test{
        public static void main(String[] args){
            Account account = new Account();

            account.createAccount("abc123", "Durga", "Savings", 25000);
            account.createAccount("abc123", "Durga", "Savings", 25000);
            System.out.println();

            account.searchAccount("abc123");
            account.searchAccount("xyz123");
            System.out.println();

            account.updateAccount("xyz123", "AAA", "Savings", 20000);
            account.updateAccount("abc123", "Nag", "Current", 50000);

```

```

        account.searchAccount("abc123");
        System.out.println();

        account.deleteAccount("xyz123");
        account.deleteAccount("abc123");
        account.searchAccount("abc123");

    }
}

```

D:\FullstackJava830\JAVA830>javac Test.java

D:\FullstackJava830\JAVA830>java Test

Account abc123 Created Successfully

Account abc123 Existed Already

Account abc123 Existed

Account Details

```

Account Number      : abc123
Account Holder Name : Durga
Account Type        : Savings
Account Balance     : 25000
Account xyz123 Does Not Exist

```

Account xyz123 Does Not Exist

Account abc123 Updated Successfully

Account abc123 Existed

Account Details

```

Account Number      : abc123
Account Holder Name : Nag
Account Type        : Current
Account Balance     : 50000

```

Account xyz123 Does Not Exist

Account abc123 Deleted Successfully

Account abc123 Does Not Exist

D:\FullstackJava830\JAVA830>

In Java, there are two ways to describe the method details.

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 the name of the method and the parameter list of the method.

EX:

`forName(String className)`

Method Prototype is the method description that includes the Access Modifiers , Return Type, Method Name, parameter List and Throws Exception List.

EX:

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

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

1. Mutator Methods
2. Accessor Methods

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

Ans:

Mutator Method is a Java Method, it is able to set / modify data in the object.

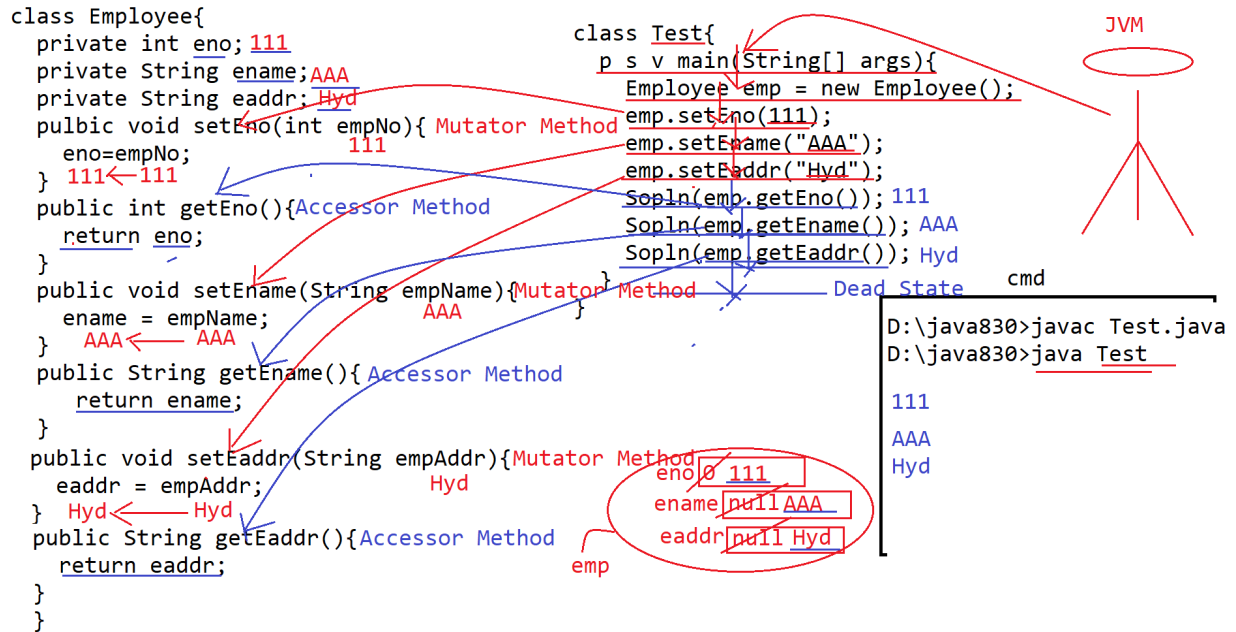
EX: All `setXXX()` methods in the Java Bean classes are Mutator Methods.

Accessor Method is a Java method, it is able to access / get data from the Object.

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

Note: Java Bean is a reusable software component, it is a class to represent the state of an entity, it is a normal class with the private properties and with the respective `setXXX()` methods and `getXXX()` methods.

The main advantage of declaring all properties as private and all methods as public is to achieve "Encapsulation" in the Java applications.



EX:

```

class Employee{

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

    public void setEno(int empNo){
        eno = empNo;
    }
    public int getEno(){
        return eno;
    }

    public void setName(String empName){
        ename = empName;
    }
    public String getName(){
        return ename;
    }

    public void setEsal(float empSal){
        esal = empSal;
    }
}
  
```

```

    }
    public float getEsal(){
        return esal;
    }

    public void setEaddr(String empAddr){
        eaddr = empAddr;
    }
    public String getEaddr(){
        return eaddr;
    }
}
class Test{
    public static void main(String[] args){

        Employee employee = new Employee();

        employee.setEno(111);
        employee.setName("Durga");
        employee.setEsal(50000.0f);
        employee.setEaddr("Hyd");

        System.out.println("Employee Details");
        System.out.println("-----");
        System.out.println("Employee Number      : "+employee.getEno());
        System.out.println("Employee Name        : "+employee.getName());
        System.out.println("Employee Salary      : "+employee.getEsal());
        System.out.println("Employee Address     : "+employee.getEaddr());

    }
}

```

D:\FullstackJava830\JAVA830>javac Test.java

D:\FullstackJava830\JAVA830>java Test

Employee Details

```

Employee Number      : 111
Employee Name        : Durga
Employee Salary      : 50000.0
Employee Address     : Hyd

```

Var-Arg Methods:

In Java applications, if we declare a method with 'n' number of parameters then we have 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 parameter values or 'n+1' number of parameter values.

In Java applications, if we want to access a method by passing a variable number of parameter values then we have to use "Variable-Argument Method", in short "Var-Arg Method".

Var-Arg method is a normal java method, it must have a Var-Arg parameter.
Var-Arg parameter : DataType ... varName

EX:

```
void m1(int ... a){  
}
```

In Java applications, when we access a variable argument method with the argument values then the variable argument parameter will be converted as an array of the same data type with the argument values as array elements.

```
m1();=====> int[] a = {};  
m1(10);=====> int[] a = {10};  
m1(10,20);=====> int[] a = {10,20};  
m1(10,20,30);====> int[] a = {10,20,30};
```

EX:

```
class A{  
    void m1(int ... a){// int[] a = {}  
        int result = 0;  
        System.out.println("No Of Arguments    : "+a.length);  
        System.out.print("Argument List      : ");  
        for(int index = 0; index < a.length; index++){  
            System.out.print(a[index]+" ");  
            result = result + a[index];  
        }  
        System.out.println();  
        System.out.println("Arguments SUM      : "+result);  
        System.out.println("-----");  
    }  
}  
  
class Test{
```



```

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

```

D:\FullstackJava830\JAVA830>javac Test.java

D:\FullstackJava830\JAVA830>java Test

No Of Arguments : 0

Argument List :

Arguments SUM : 0

No Of Arguments : 1

Argument List : 10

Arguments SUM : 10

No Of Arguments : 2

Argument List : 10 20

Arguments SUM : 30

No Of Arguments : 3

Argument List : 10 20 30

Arguments SUM : 60

D:\FullstackJava830\JAVA830>

Q) Is it possible to provide normal parameters along with the variable-Argument parameter in the Variable-Argument 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, not after the var-arg parameter, because in the Var-Arg method var-arg parameter must be the last parameter.

EX-1: void m1(int ... i, float f){ } ---> Invalid

EX-2: void m1(float f, int ... i){ } ---> Valid

EX:

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

D:\FullstackJava830\JAVA830>javac Test.java

D:\FullstackJava830\JAVA830>java Test

Var-Arg Method

Var-Arg Method

Var-Arg Method

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

Ans:

No, it is not possible to provide more than one Var-Arg parameter in a single Var-Arg method , because in the Var-Arg method Var-Arg parameter must be the last parameter, if we provide more than one Var-Arg parameter in a single Var-Arg method then the last var-Arg parameter is valid, but the last but one Var-Arg parameter is Invalid.

EX:

```
void m1(int ... a, float ... f){    } -----> Invalid
```

EX:

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

```

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

```

```

D:\FullstackJava830\JAVA830>javac Test.java
Test.java:2: error: varargs parameter must be the last parameter
    void m1(float ... f, int ... a){
                ^

```

Objects in Java:

Q)What is the requirement of the objects in java?

Ans:

The main purpose of the objects in Java applications is

1. Java is an object oriented programming language, in java applications majority of the operations are associated with the objects only, so in java applications it is a convention to create objects for the classes.
2. To store entities data temporarily we have to create objects.
3. To access the members of a particular class we have to create objects and we have to use the generated reference variables.

Syntax:

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

Where “ClassName([ParamValues])” is a constructor in the respective class.

EX:

```

class A{
    -----
}

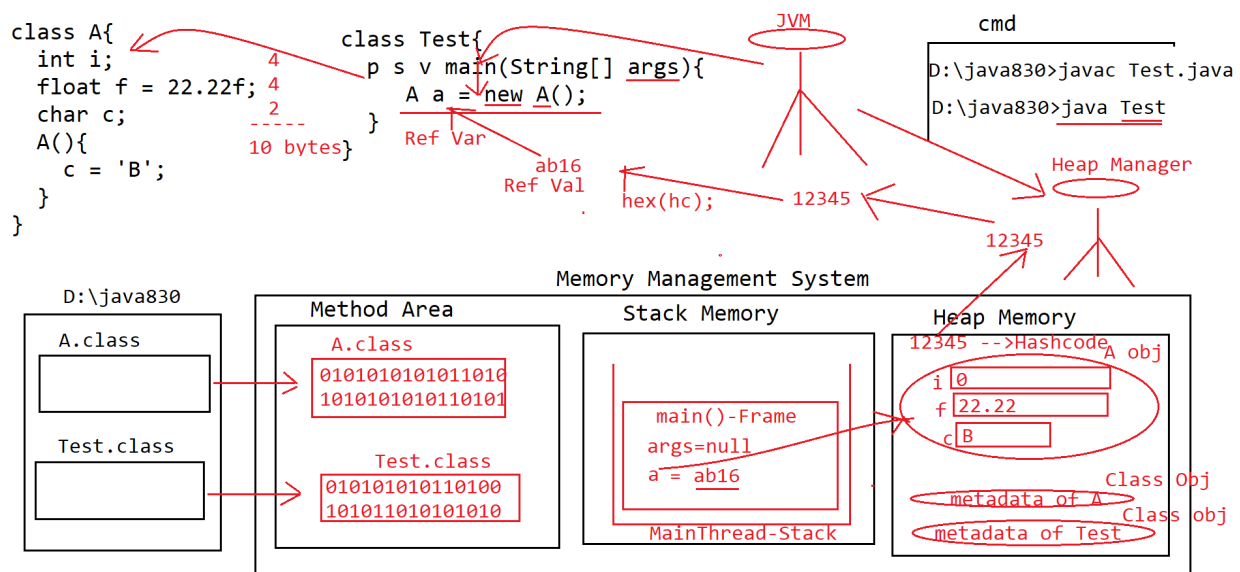
```

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

In Java applications, when we execute the above object creation statement , JVM will execute “new” keyword, where “new” will provide the total object creation process to the JVM like below.

1. JVM will identify the class name to which an object is created that is the “Constructor Name”.
2. JVM will search for the respective class’s .class file at the current location, at java predefined library and at the locations referred by the “classpath” environment variable.
3. If the respective class’s .class file is available at either of the above locations then the JVM will load the .class file provided bytecode to the memory , that is in Method Area.
4. When the class bytecode is loaded in the method area, JVM will create an object of the class java.lang.Class with the metadata of the loaded class, where the metadata of the loaded class including Class Name, access modifiers of the class, superclass details, implemented interfaces details, declared variables, declared methods, declared constructors,.....
5. After loading the class in Method Area, JVM will goto the loaded class , JVM will recognize all the instance variables and their data types of the class and JVM will find the minimal memory size for the object.
6. After finding the minimum object memory size , JVM will request the Heap manager about to create an object for the respective class.
7. As per the JVM request, Heap Manager will create an Object[Block of memory] for the respective class.
8. After creating the Memory for the object, Heap Manager will assign an unique identity for the object in the form of an integer value called “Hashcode” of the object.
9. After creating a Hashcode value to the Object, Heap Manager will send the generated Hashcode value to the JVM.
10. JVM will convert the provided Hashcode value into its hexadecimal form called “Reference Value” and JVM will assign the generated reference value to a variable called “Reference Variable”.
11. After creating the identities for the Object, JVM will provide memory allocation for all the instance variables on the basis of their data types.
12. After the memory allocation for the instance variables, JVM will provide the initialization for the instance variables inside the objects by searching the initializations at class level variables declaration and at the constructor.
13. If the instance variables are initialized at class level or at constructor then the JVM will provide that initial values inside the object, if no initialization is identified at both the class level

declaration and at the constructor then the JVM will provide default value as the initialization for that variable depending on the 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 java method declared in java and implemented in a non java programming language like C, Pascal, H/W languages,...

EX: `hashCode()` method is a native method declared in Java and implemented in a H/W language in order to find the hashcode value from the Heap Manager in order to send that hashcode value to the JVM and Java application.

In java applications, to get the reference value of an object we have to use the following method.

```
public String toString()
```

EX:

```
class A{

}

class Test{
    public static void main(String[] args) {
        A a = new A();
        int hc = a.hashCode();
        String refValue = a.toString();
        System.out.println("Hashcode      : "+hc);
        System.out.println("Ref Value      : "+refValue);

    }
}
```

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

```
D:\java830>java Test
```

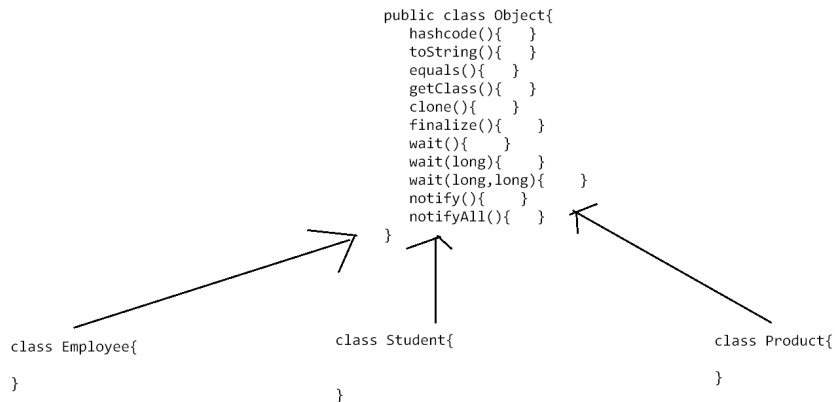
```
Hashcode      : 918221580
```

```
Ref Value      : A@36baf30c
```

In Java, there is a common and default superclass for each and every java class that is “java.lang.Object” class. java.lang.Object class is able to provide the following 11 methods in each and every java class by default.

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

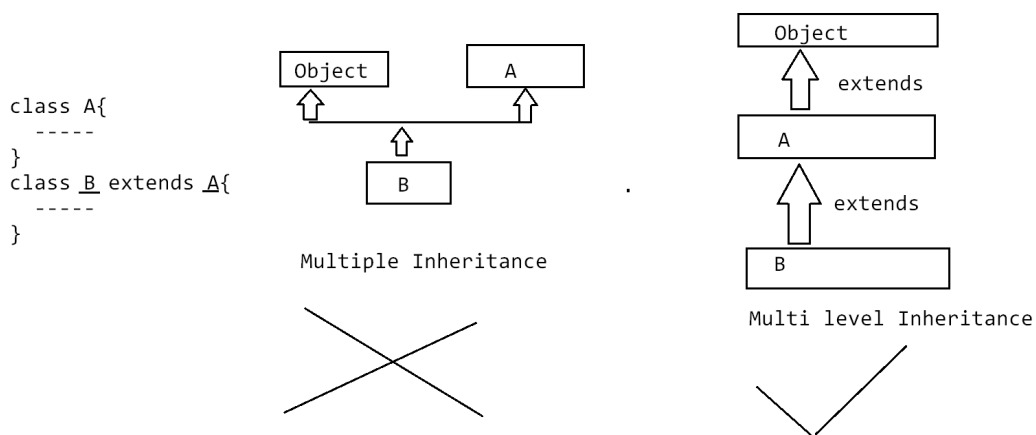
In Java applications, we can access all the above java.lang.Object class provided methods by using any class reference variable in any class.



Q) In Java applications, if we extend a class to a subclass explicitly then the respective subclass will have two superclasses like the default superclass `java.lang.Object` and the explicit superclass, it represents Multiple inheritance, so 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 each and every class when the class is not extending from any superclass explicitly, if a class is extending from a superclass explicitly then the `java.lang.Object` class is not directly a superclass to the subclass, it will be a superclass to the subclass indirectly through Multi level inheritance, but not through Multiple Inheritance.



Importance of toString() method in Java:

toString() method was defined in java.lang.Object class, it was implemented in such a way that to return a String containing "ClassName@RefVal".

In Java applications, if we pass a particular class object reference variable as parameter to System.out.println() method then JVM will access toString() method internally on the reference variable which we passed as parameter to System.out.println() method. In this case, JVM will search for the toString() method in the respective class whose reference variable we passed as parameter to System.out.println() method. If the toString() method does not exist in the respective class then JVM will search for the toString() method in its superclass, if no superclass is existed then the JVM will search for the toString() method in the common and default superclass java.lang.Object class.

EX:

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

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

```
D:\java830>java Test
```

```
A@36baf30c
```

```
A@36baf30c
```

```
A@36baf30c
```


In Java applications, we want to display our own messages when compared with the Object reference values when we pass reference variables as parameters to `System.out.println()` method , to achieve this requirement we have to provide our own `toString()` method in the respective class.

EX:

```
class Account{
    String accNo = "abc123";
    String accHolderName = "Durga";
    String accType = "Savings";
    int balance = 50000;
    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 Test{
    public static void main(String[] args) {
        Account acc = new Account();
        System.out.println(acc);// Sopln(acc.toString());
    }
}
```

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

```
D:\java830>java Test
```

```
Account Details
```

```
-----
```

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

Note: In Java, some predefined classes like `String`, `StringBuffer`, `Exception`, `Thread`, `Wrapper` classes, `Collection` classes,.... Are not dependent on the `Object` class provided `toString()` methods, they are having their own `toString()` methods to display their own data when we pass the respective class object reference variables as parameters to `Sopln()` method.

EX:

```
import java.util.*;
class Test{
    public static void main(String[] args){
        String str = "Durgasoft";
        System.out.println(str);

        Exception exception = new Exception("My Own Exception");
        System.out.println(exception);

        Thread thread = new Thread();
        System.out.println(thread);

        Integer in = new Integer(100);
        System.out.println(in);

        List list = Arrays.asList(10,20,30,40,50);
        System.out.println(list);

    }
}
```

```
D:\FullstackJava830\JAVA7>javac Test.java
D:\FullstackJava830\JAVA7>java Test
Durgasoft
java.lang.Exception: My Own Exception
Thread[Thread-0,5,main]
100
[10, 20, 30, 40, 50]
```

There are two types of Objects in Java.

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 will not allow the modifications on their content, here data is allowed for the modifications but the resultant modified data will not be stored back in the original object, here the resultant modified data will be stored by creating another new object.

EX: String class objects are immutable objects.

EX: All Wrapper classes objects are immutable objects.

Mutable Objects are the java objects, they will allow the modifications directly on their content.

EX: By default all Java objects are mutable objects.

EX: StringBuffer

EX:

```
class Test{
    public static void main(String[] args){
        String str1 = new String("Durga ");
        String str2 = str1.concat("Software ");
        String str3 = str2.concat("Solutions");

        System.out.println(str1);
        System.out.println(str2);
        System.out.println(str3);
        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);

    }
}
```

D:\FullstackJava830\JAVA7>javac Test.java

```
D:\FullstackJava830\JAVA7>java Test
```

Durga

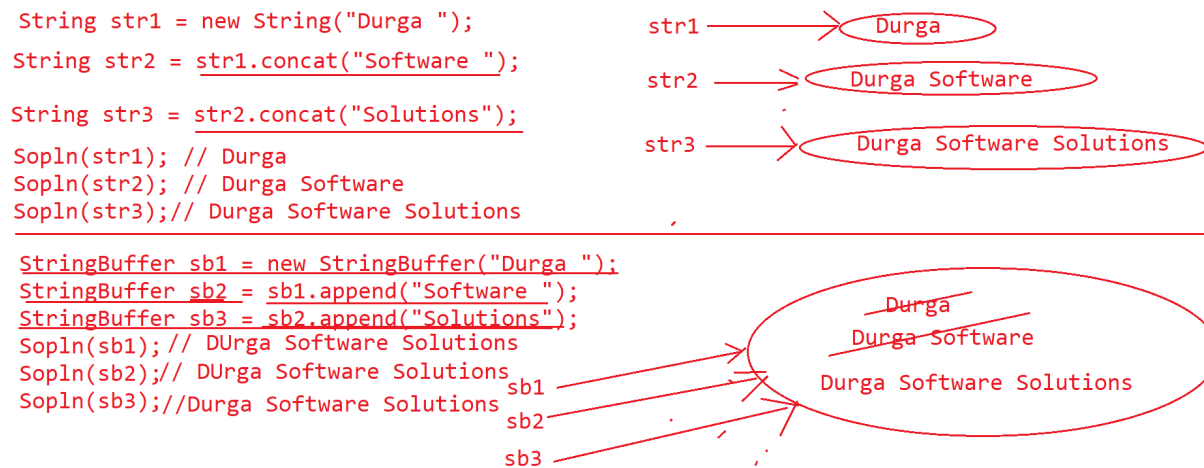
Durga Software

Durga Software Solutions

Durga Software Solutions

Durga Software Solutions

Durga Software Solutions



Q)What is the difference between Object and Instance?

Ans:

Object is a block of memory to store data.

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

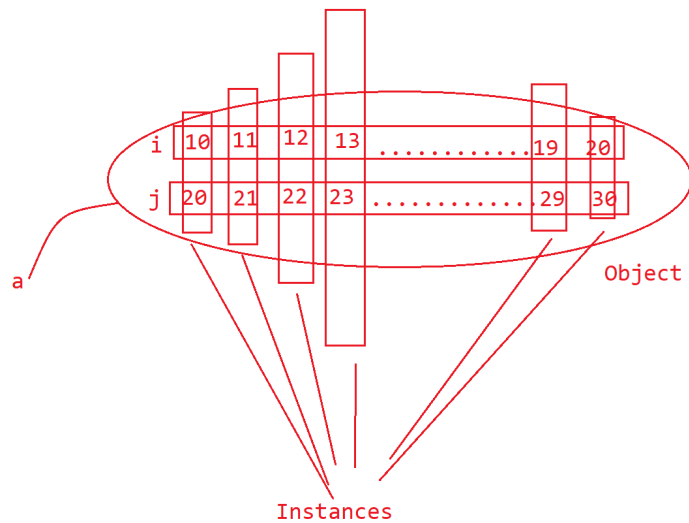
Single Object can have multiple instances, but we are always able to get the latest instance only, because the latest instances are overriding the previous instances inside the object.

EX:

```

class A{
    int i = 10;
    int j = 20;
}
class Test{
    p s v main(String[] args){
        A a = new A();
        for(int x = 1; 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 constructor in the object creation process is to provide initial values inside the objects.
3. In Java applications, Constructors are recognized and executed 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. Constructor name must be the same as the respective class name.
6. Constructors are not having return types.
7. Constructors are not allowing the access modifiers like static, final, abstract,....
8. Constructors are allowing access modifiers like public, protected, <default> and private.
9. Constructors are allowing throws keyword to bypass the generated exception to the caller of the constructor.

Syntax:

```

[Access Modifier] ClassName([ParamList])[throws ExceptionList]{
    -----
}

```

EX:

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

D:\FullstackJava830\JAVA830>javac Test.java

D:\FullstackJava830\JAVA830>java Test

A-Con

In Java applications, if we provide a different name to the constructor not the same as the class name then the compiler will raise an error like “Invalid Method declaration; return type required”, because if we provide different name to the constructor not same as the class name then the compiler will treat the provided constructor as a normal java method and the compiler will check its syntax against to the method syntax, in Java methods return type is mandatory.

EX:

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

D:\FullstackJava830\JAVA830>javac Test.java

Test.java:2: error: invalid method declaration; return type required

In Java applications, constructors are not having return type, in the case if we provide a return type to the constructor then the provided constructor

will be converted as a java method with the same class name. In this context, we have to access the constructor like the java method.

EX:

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

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

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

```
A-Con
```

In general, constructors are not allowing Access modifiers like static, final, abstract,.... , but if we provide these access modifiers to the constructor then the compiler will raise an error like “modifier XXX not allowed here”.

EX:

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

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

```
Test.java:2: error: modifier static not allowed here
```

In Java applications, Constructors are able to allow the access modifiers like public, protected, <default> and private, if we declare any constructor as private then we are able to access that constructor in the same class, not possible to access that constructor outside of the respective class.

EX:

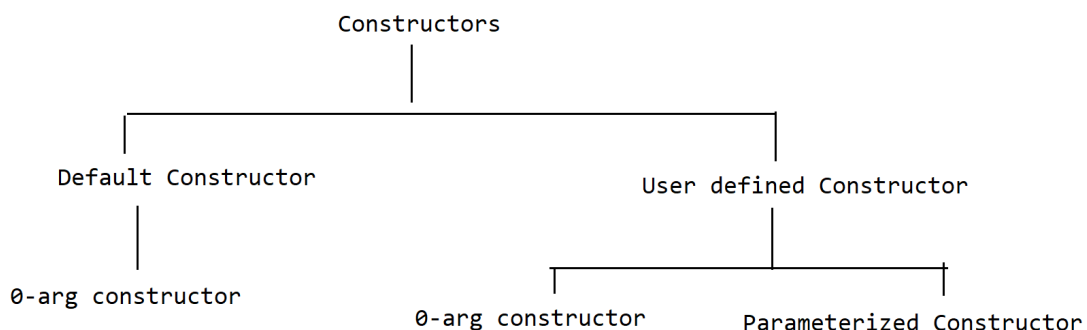
```
class A{
    private A(){
        System.out.println("A-Con");
    }
}
class Test{
    public static void main(String[] args){
        A a = new A();
    }
}
```

```
D:\FullstackJava830\JAVA830>javac Test.java
Test.java:8: error: A() has private access in A
    A a = new A();
               ^
```

Types of Constructors:

There are two types of constructors in Java.

1. Default Constructor.
2. User defined constructor.



Default Constructor:

In a class, if we have not provided any constructor explicitly then the compiler will add a 0-arg constructor to the respective class, here the compiler provided 0-arg constructor is called a Default Constructor, if we

provide at least one constructor in the class then the compiler will not provide default constructor.

EX:

```
class Test{  
}
```

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

```
D:\FullstackJava830\JAVA830>javap Test  
Compiled from "Test.java"  
class Test extends java.lang.Object{  
    Test(); -----> Default Constructor.  
}
```

In Java, by default the compiler will provide a 0-arg constructor as default constructor, all default constructors are 0- arg constructors but all 0-arg constructors are not default constructors, some 0-arg constructors which are provided by the compiler are default constructors, some other 0-arg constructors which are not provided by the compiler and which are defined by the developers are called user defined constructors.

User Defined Constructor:

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

There are two types of user defined constructor.

1. 0-arg constructor
2. Parameterized Constructor

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

If we declare any constructor with the parameters then that constructor is called Parameterized constructor.

EX:

```
Account.java  
package com.durgasoft.entities;  
public class Account {
```

```

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

public void setAccountDetails() {
    accNo = "abc123";
    accHolderName = "Durga";
    accType = "Savings";
    balance = 50000;
}

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      : "+balance);
}
}

Test.java
package com.durgasoft.test;
import com.durgasoft.entities.Account;
public class Test {
    public static void main(String[] args) {
        Account account = new Account();
        account.setAccountDetails();
        account.getAccountDetails();
    }
}

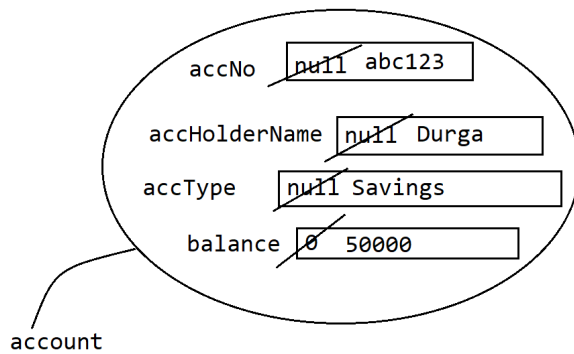
```

Account Details

```

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

```



In the above program, when the Account class object is created , the default constructor has been executed, it does not provide initializations for the instance variables , so JVM provides default values for the Account details inside the Account object.

In the above context, after creating the Account class object , when we access setAccountDetails() method then only the real account details are created inside the object, here the provided account details through the setAccountDetails() method is not the first data inside the Account, it is the second data, but as per the applications requirement we must provide all account details as the first data at the time of creating Account only, that is at the time of creating Account class object only.

In Java applications, if we want to provide data in an object at the time of creating the object then we have to use a constructor explicitly[User defined Constructor].

EX:

Account.java

```
package com.durgasoft.entities;
public class Account {

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

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

```

        balance = 50000;
    }

    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     : "+balance);
    }
}

```

Test.java

```

package com.durgasoft.test;
import com.durgasoft.entities.Account;
public class Test {
    public static void main(String[] args) {
        Account account = new Account();
        account.getAccountDetails();
    }
}

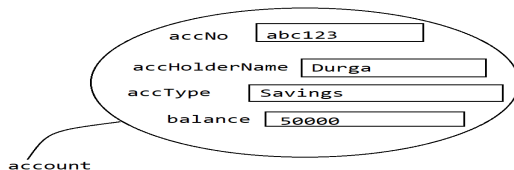
```

Account Details

```

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

```



Consider the below program

ACcount.java

```

package com.durgasoft.entities;
public class Account {

    String accNo;
    String accHolderName;

```

```

String accType;
int balance;

public Account() {
    accNo = "abc123";
    accHolderName = "Durga";
    accType = "Savings";
    balance = 50000;
}

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    : "+balance);
}
}

```

Test.java

```

package com.durgasoft.test;
import com.durgasoft.entities.Account;
public class Test {
    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();
    }
}

```

Account Details

```

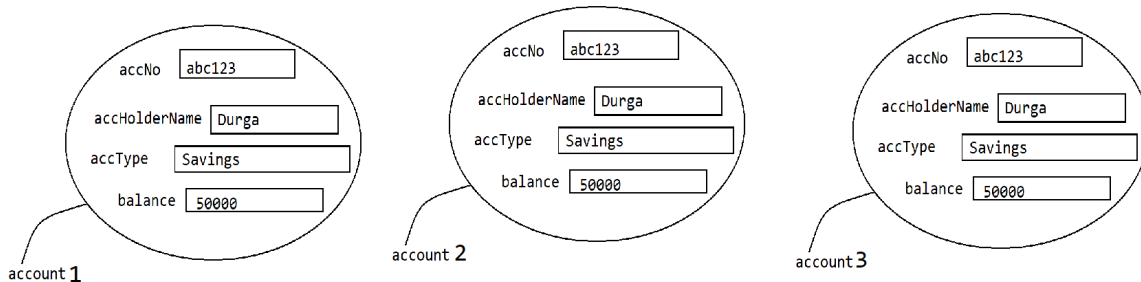
-----
Account Number      : abc123
Account Holder Name : Durga

```

Account Type : Savings
Account Balance : 50000
Account Details

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

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



In the above program, if we create multiple objects for the Account class then in all the objects the account data must be the same but as per the applications requirement, all the Account class objects must have the different account details . If we want to provide different data inside the objects we have to use the parameterized constructor in place of the 0-arg constructor.

EX:

Account.java

```
package com.durgasoft.entities;  
public class Account {
```

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

```
    public Account(String accountNumber, String accountHolderName, String  
accountType, int accountBalance) {  
        accNo = accountNumber;
```

```

        accHolderName = accountHolderName;
        accType = accountType;
        balance = accountBalance;
    }

    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     : "+balance);
    }
}

Test.java
package com.durgasoft.test;
import com.durgasoft.entities.Account;
public class Test {
    public static void main(String[] args) {
        Account account1 = new Account("a111", "Durga", "Savings",
25000);
        account1.getAccountDetails();
        System.out.println();

        Account account2 = new Account("a222", "Anil", "Savings", 50000);

        account2.getAccountDetails();
        System.out.println();

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

```

Account Details

```

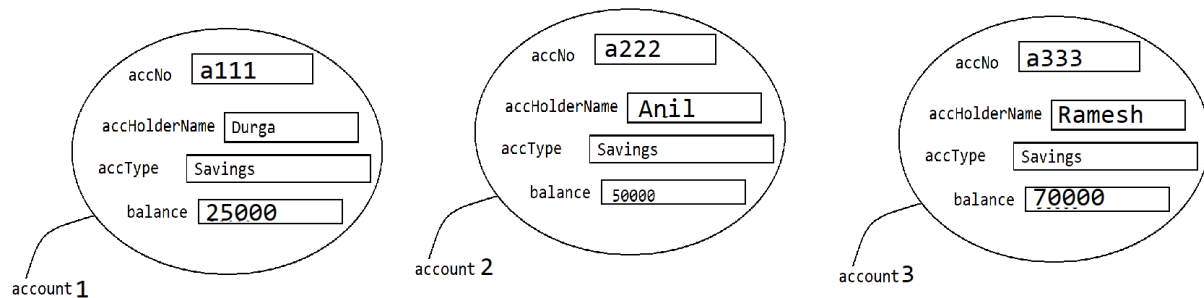
Account Number      : a111
Account Holder Name : Durga
Account Type        : Savings
Account Balance     : 25000

```

Account Details

Account Number : a222
Account Holder Name : Anil
Account Type : Savings
Account Balance : 50000
Account Details

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



Q)What is Constructor Overloading?

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

EX:

```
package com.durgasoft.test;
class A{
    int i, j, k;
    public A() {

    }
    public A(int x) {
        i = x;
    }
    public A(int x, int y) {
        i = x;
        j = y;
    }
    public 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 Test {
    public static void main(String[] args) {
        A a1 = new A();
        a1.add();// 0

        A a2 = new A(10);
        a2.add();// 10

        A a3 = new A(10,20);
        a3.add();// 30

        A a4 = new A(10,20,30);
        a4.add();

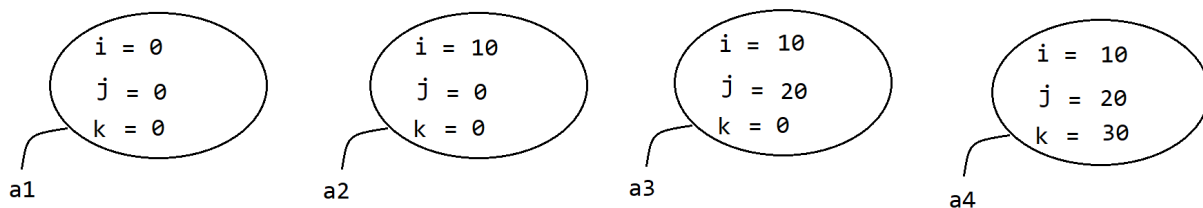
    }
}

```

```

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

```



Instance Context / Instance Flow of Execution

In Java applications, when we load a class bytecode to the memory a separate context or environment will be created called Static Context.

In Java applications, when we create an object a separate context or environment will be created called Instance Context.

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

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

Instance variable:

If any variable changes its value from one instance to the another instance of an object then that variable is called an Instance Variable.

Instance variable is a Java variable, it will be recognized and initialized just before executing the respective class constructor in the Object creation process.

In Java applications, instance variables are declared at class level only without using the 'static' keyword, that is a class level and non-static variable is called an Instance variable.

In Java applications, to access an instance variable of a particular class then we have to create an object for the respective class and we have to use the generated reference variable.

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

EX:

```
package com.durgasoft.test;
class A{
    int i = 10;
}
public class Test {
    public static void main(String[] args) {
        A a = new A();
        System.out.println(a.i);
        A a1 = null;
        //System.out.println(a1.i); java.lang.NullPointerException
    }
}
```

```
    }  
}
```

10

```
Exception in thread "main" java.lang.NullPointerException: Cannot read field  
"i" because "a1" is null  
    at com.durgasoft.test.Test.main(Test.java:11)
```

Instance Method:

Instance method is a Java method, it will be recognized and accessed the moment when we access that method.

In Java applications, if we declare a method without using the 'static' keyword then that method is called an instance method.

In Java applications, to access instance methods of a particular class we have to create an object for the respective class and we have to use the generated reference variable.

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

EX:

```
package com.durgasoft.test;  
class A {  
    void m1() {  
        System.out.println("m1-A");  
    }  
}  
public class Test {  
    public static void main(String[] args) {  
        A a = new A();  
        a.m1();  
        A a1 = null;  
        a1.m1();  
    }  
}
```

m1-A

Exception in thread "main" [java.lang.NullPointerException](#): Cannot invoke "com.durgasoft.test.A.m1()" because "a1" is null
at com.durgasoft.test.Test.main([Test.java:15](#))

EX:

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

m1-A
A-Con

Q) What are the differences between Constructor and Method in Java?

Ans:

1. Constructor is a Java feature, it can be used to create the objects.

Method is a Java feature, it can be used to represent a particular action of an entity.

2. The main purpose of the constructor in the object creation process is to provide initial values inside the objects.

The main purpose of the methods in Java applications is to represent a particular task of an entity.

3. In Java applications, constructors are recognized and executed automatically when we are creating objects by using new keyword.

In Java applications, methods are recognized and executed the moment when we access that method, in Java applications methods won't be executed without calling the methods.

4. In Java applications, we can access the constructors along with the new keyword in the object creation statement and by using 'this' and 'super' keywords.

In Java applications, we can access the methods in the following ways

- a. Without using any reference variable or any keyword if the method is the current class method.
- b. By using 'this' keyword if the method is the current class method.
- c. By Using the respective class reference variable if the method is an instance method from a particular class.
- d. By Using the respective class reference variable and class name if the method is a static method from a particular class.

5. Constructors name and the respective class name must be the same.

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

6. Constructors are not allowing return types.

In Java applications, Methods must have the return types.

7. Constructors must not allow the access modifiers like static, final , abstract ,....

Methods are able to allow the access modifiers like static, final, abstract,....

8. In Java applications, default constructors are possible, when we have not provided any constructor in a class then the compiler will provide a 0-arg constructor inside the class internally , here the compiler provided 0-arg constructor is called a Default Constructor.

No default methods exist in the Java applications.

Instance Blocks:

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

In Java applications, instance blocks are having the same power of the constructors , but we are unable to use instance blocks as the replacement to the constructors.

Syntax:

```
{  
    -----INSTRUCTIONS-----  
}
```

EX:

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

IB-A

A-Con

EX:

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

```

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

```

m1-A

IB-A

A-Con

EX:

```

package com.durgasoft.test;
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 Test {
    public static void main(String[] args) {
        A a1 = new A();
        A a2 = new A();
    }
}

```

m1-A

IB-A

A-Con

m1-A

IB-A

A-Con

‘this’ keyword:

The main purpose of the ‘this’ keyword is to represent the current class object.

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

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.

1. Referring the current class variables by using this keyword:

If we want to refer to the current class variables by using 'this' keyword then we have to use the following syntax.

```
this.varName;
```

In Java applications, when we have the same variables at class level and at local , where to refer the class level variables over the local variables there we will use 'this' keyword, because the compiler and JVM will provide highest priority for the local variables only.

EX:

```
package com.durgasoft.test;
class A{
    int i = 10;
    int j = 20;
    A(int i, int j){
        System.out.println(i+" "+j);
        System.out.println(this.i+" "+this.j);
    }
}
public class Test {
    public static void main(String[] args) {
        A a = new A(30,40);
    }
}

30    40
10    20
```

In Java applications, in Java Bean classes, we will provide a separate setXXX() method for each and every property, where the setXXX() method will have a parameter variable with the same name of the respective class level variable, in this case, inside the setXXX() method we must assign the

parameter variable value to the class level variable , in this case we have to represent the class level variable over the local variable , to represent class level variable over the local variable we have to use 'this' keyword.

EX:

Employee.java

```
package com.durgasoft.beans;
public class Employee {

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

    public void setEno(int eno) {
        this.eno = eno;
    }
    public int getEno() {
        return eno;
    }

    public void setName(String ename) {
        this.ename = ename;
    }
    public String getName() {
        return ename;
    }

    public void setEsal(float esal) {
        this.esal = esal;
    }
    public float getEsal() {
        return esal;
    }

    public void setEaddr(String eaddr) {
        this.eaddr = eaddr;
    }
    public String getEaddr() {
        return eaddr;
    }
}
```

Test.java

```
package com.durgasoft.test;
import com.durgasoft.beans.Employee;
public class Test {
    public static void main(String[] args) {
        Employee employee = new Employee();
        employee.setEno(111);
        employee.setName("Durga");
        employee.setEsal(50000);
        employee.setEaddr("Hyd");

        System.out.println("Employee Details");
        System.out.println("-----");
        System.out.println("Employee Number      : "+employee.getEno());
        System.out.println("Employee Name      : 
"+employee.getName());
        System.out.println("Employee Salary      : "+employee.getEsal());
        System.out.println("Employee Address      : 
"+employee.getEaddr());
    }
}
```

Employee Details

```
-----
Employee Number      : 111
Employee Name      : Durga
Employee Salary      : 50000.0
Employee Address      : Hyd
```

Referring current class methods by using 'this' keyword:

In general, in Java applications, to access the current class methods no need to use any reference variable and any special keywords, directly we can access the current class methods.

In Java applications, it is possible to access the current class methods by using 'this' keyword also.

Syntax:

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

EX:

```

package com.durgasoft.test;
class A{
    void m1() {
        System.out.println("m1-A");
        m2();
        this.m2();
    }
    void m2() {
        System.out.println("m2-A");
    }
}
public class Test {
    public static void main(String[] args) {
        A a = new A();
        a.m1();
    }
}

```

Referring Current class constructors by using 'this' keyword:

If we want to refer to a particular current class constructor by using 'this' keyword then we have to use the following syntax.

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

```
this(); ----> Accessing 0-arg constructor
```

```
this(10); ----> Accessing int-parameterized constructor
```

```
this(22.22f);-----> Accessing float parameterized constructor
```

```
this(10,20); -----> Accessing 2-parameterized constructor.
```

EX:

```

package com.durgasoft.test;
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");
    }
}
public class Test {
    public static void main(String[] args) {
        A a = new A();
    }
}

```

```

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

```

In the above program, class A contains more than one constructor with the same name and with the different parameter list , it is called Constructor Overloading.

In the above applications, in class A, all the constructors are executed in a chain fashion by using this keyword, it is called “Constructor Chaining”.

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

1. If we want to refer to the current class constructor by using ‘this’ keyword then the respective ‘this’ statement must be the first statement in the constructor. If we violate this rule then the compiler will raise an error.

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();
    }
}

```

D:\FullstackJava830\JAVA7>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(33.333);

2. If we want to refer to the current class constructor by using this keyword then we have to access that current class constructor from the other current class constructor only, not from the normal java method. If we violate this rule then the compiler will raise an error.

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();
    }
}

```

D:\FullstackJava830\JAVA7>javac Test.java
Test.java:9: error: call to this must be first statement in constructor
 this(10);
 ^

4. To return Current class object:

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

return this;

EX:

```

package com.durgasoft.test;
class A{
    public A getRef1(){
        A a = new A();//a222, a333, a444
        return a;
    }
    public A getRef2() {
        return this;// returns the same reference value on which we have
accessed this method.
    }
}
public class Test {
    public static void main(String[] args) {
        A a = new A();//a111
        System.out.println(a);//A@a111
        System.out.println();

        System.out.println(a.getRef1());//Sopln(a222);==> A@a222
        System.out.println(a.getRef1());//Sopln(a333); ==> A@a333
        System.out.println(a.getRef1());//Sopln(a444); ==> A@a444
        System.out.println();
    }
}

```

```
System.out.println(a.getRef2());//Sopln(a111.getRef2());==>Sopln(a111);==>A@a111
```

```
System.out.println(a.getRef2());//Sopln(a111.getRef2());==>Sopln(a111);==>A@a111
```

```
System.out.println(a.getRef2());//Sopln(a111.getRef2());==>Sopln(a111);==>A@a111
```

```
    }  
}
```

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

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

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

In the above program, when we access getRef1() method, every time JVM will execute the “new” keyword, every time JVM will create a new object and every time JVM will return the new object reference value , this approach will increase the number of duplicate objects.

In the above program, when we execute getRef2() method, every time, JVM will return this keyword, every time it will return the same object reference value on which we have accessed getRef2() method, this approach will reduce the number of duplicate objects.

‘Static’ keyword:

‘Static’ is a Java keyword, it will improve Shareability in Java applications.

In Java applications, the static keyword will be utilized in the following four ways.

1. Static Variables
2. Static Methods

3. Static Blocks
4. Static import

Static Variables:

-
1. Static Variable is a Java variable, it will be recognized and initialized at the time of loading the respective class bytecode to the memory.
 2. In Java applications, the last modified value of the static variable will be shared to all the previous objects which we have created already and to the future objects which we are going to create.
 3. In Java applications, static variable values will be stored in the method area.
 4. In Java applications, static variables must be declared as class level variables, not as local variables. If we declare any local variable as a static variable then the compiler will raise an error.
 5. In Java applications, we are able to access the static variables in the following approaches.
 - a. If the static variable is the current class variable then it is possible to access it directly or by using this keyword.
 - b. If the static variable is declared in a particular class then it is possible to access that static variable either by using class name directly or by using the respective class reference variable.

Note: In Java applications, it is always suggested to use class names to access static variables.

6. 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`, 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`.

EX:

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

    void m1(){
        //static int k = 30; ---> Error
        System.out.println("m1-A");
    }
}
```



```

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

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

```

D:\FullstackJava830\JAVA830>javac Test.java

D:\FullstackJava830\JAVA830>java Test

```

10
10
m1-A
10
10
10

```

EX:

```

package com.durgasoft.test;
class A{
    int i = 10;
    static int j = 10;
}
public class Test {
    public static void main(String[] args) {
        A a1 = new A();
        System.out.println(a1.i+" "+a1.j);
        a1.i = a1.i + 1;
        a1.j = a1.j + 1;

        A a2 = new A();
    }
}

```

```

        System.out.println(a2.i+"    "+a2.j);
        a2.i = a2.i + 1;
        a2.j = a2.j + 1;
        System.out.println(a1.i+"    "+a1.j);
        System.out.println(a2.i+"    "+a2.j);

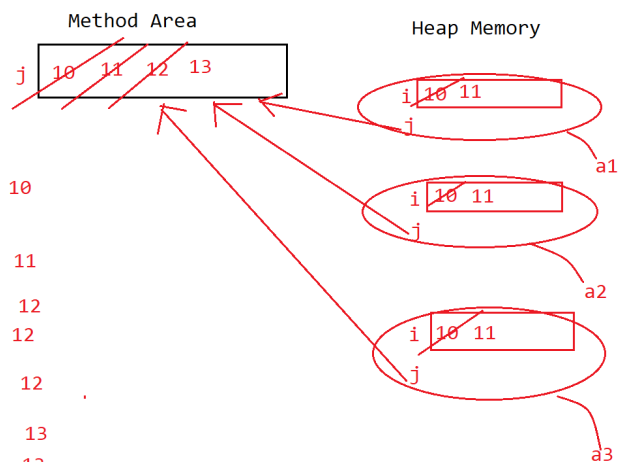
        A a3 = new A();
        System.out.println(a3.i+"    "+a3.j);
        a3.i = a3.i + 1;
        a3.j = a3.j + 1;
        System.out.println(a1.i+"    "+a1.j);
        System.out.println(a2.i+"    "+a2.j);
        System.out.println(a3.i+"    "+a3.j);
    }
}

```

```

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

```



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

Ans:

1. Instance variable is a java variable, its value will be varied from one instance to another instance of an object.

Static Variable is a Java variable, its values are not related to the instances of an object.

2. In Java applications, instance variables are recognized and initialized just before executing the respective class constructor.

In Java applications, static variables are recognized and initialized the moment when we load the respective class bytecode to the memory.

3. To declare instance variables no need to use any special keyword.

To declare static variables we must use the “static” keyword.

4. Instance variables are allowed in only instance methods directly, instance variables are not allowed in the static methods directly.

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

5. In Java applications, to access an instance variable of a particular class we must create an object for the respective class and we must use the generated reference variable.

In Java applications, to access the static variables of a particular class we have to use either the reference variable or class name directly.

6. `java.lang.NullPointerException` is applicable for the instance variables, 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`.

`java.lang.NullPointerException` is not applicable for the static Variables, if we access any static variable by using a reference Variable that contains null value then JVM will not raise an exception like `java.lang.NullPointerException`.

7. In Java applications, if we create multiple objects for a particular class then a separate copy of the instance variable will be created at each and every object.

In Java applications, if we create multiple objects for a particular class then a single copy of the static variable will be created for all the objects.

8. In Java applications, if we perform modifications on an instance variable by using a reference variable then that modification is available to the respective object only, it will not be available to the other objects.

In Java applications, if we perform modifications on a static variable by using a reference variable then that modification is available to every object of the respective class.

9. The scope of the instance variable is up to the respective object.

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

10. Instance variables are able to provide less shareability.

Static Variables are able to provide more shareability.

11. In Java applications, instance variables data will be stored inside the objects that are in the Heap memory.

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

12. In Java applications, we will use instance variables to represent data which is individual to the objects.

EX: User Name, Password are individual to the users.

In Java applications, we will use static variables to represent data which is common to every object.

EX: MIN_AGE and MAX_AGE for all the users who are participating in the JOB drives.

Static Methods:

1. Static method is a java method, it will be recognized and executed the moment when we access that method.
2. In Java applications, static methods are able to allow only static members of the current class directly static methods are not allowing instance members of the current class directly.

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

3. Static methods are not allowing 'this' keyword in their body, but to access the current class static methods from some other methods we are able to use 'this' keyword.
4. In Java applications, we are able to access static methods in the following three ways.

If the static method is the current class method then it is possible to access that static method directly without using any reference variable and any keyword, but we can use 'this' keyword to access the current class static method.

If the static method is from some other class then it is possible to access that static method by using class name or by using the reference variable of the respective class.

Note: Always it is suggestible to access static methods by using class name when compared with the reference variable.

5. In Java applications, if we access any 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 any exception like java.lang.NullPointerException.
6. IN Java applications, static methods are able to provide more shareability than the instance methods.

EX:

```
package com.durgasoft.test;
class A{
    int i = 10;
    static int j = 20;
    static void m1() {
        System.out.println("m1-A");
        //System.out.println(i); ---> Error
        A a = new A();
        System.out.println(a.i);
        System.out.println(j);
        //System.out.println(this.j); --> Error
    }
```

```

        void m2() {
            System.out.println("m2-A");
            m1();
            this.m1();
        }
    }

    public class Test {
        public static void main(String[] args) {
            A a = new A();
            a.m1();
            A.m1();
            a.m2();

            A a1 = null;
            a1.m1();
            //a1.m2(); ---> Exception
        }
    }

```

```

m1-A
10
20
m1-A
10
20
m2-A
m1-A
10
20
m1-A
10
20
m1-A
10
20

```

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

 Ans:

1. To declare an instance method no need to use any special keywords.

To declare a static method we must use the static keyword.

2. Instance methods are getting ready after creating the objects only, we can access instance methods after the object creation only, not possible to access instance methods at the time of loading the respective class bytecode.

Static methods are getting ready at the time of loading the respective class bytecode to the memory, so we can access static methods at load time and after creating objects.

3. Instance members are able to allow both static members and instance members directly.

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

4. IN Java applications, to access instance methods we must create objects for the respective class and we must use the generated reference variable.

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

5. INstance methods allow 'this' keyword in their body.

Static methods do not allow 'this' keyword in their body.

6. java.lang.NullPointerException is applicable for the instance methods,

java.lang.NullPointerException is not applicable for the static methods.

7. INstance methods will provide less shareability.

Static methods will provide more shareability.

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 the static variable and a static method combination.

EX:

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

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

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

```
Welcome To Durga Software Solutions
```

Note: The above question and answer are valid up to the JDK1.6 version, they are invalid from JDK1.7 version, because in JAVA 1.6 version, JVM will load the main class bytecode to the memory without checking the main() method availability in the main class, after loading the main class bytecode to the memory only JVM will search for the main() method, but from JAVA 1.7 version onwards, first JVM will check the main() method availability before loading main class bytecode to the memory, if the main() is available then only JVM will load main class bytecode to the memory, if the main() method is not available in the main class then the JVM will not load main class bytecode to the memory and JVM will provide an error message like

Error: main method not found in the class Test, please define the main method as : public static void main(String[] args)

```
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 Durga Software Solutions
```

```
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)
```

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 able to allow static context directly, Static blocks are not allowing instance members directly, if we want to use instance members inside the static block then we have to create an object for the respective class and we have to use the generated reference variable.

Static blocks are not allowing 'this' keyword.

Syntax:

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

EX:

```
package com.durgasoft.test;  
class A{  
    int i = 10;  
    static int j = 20;  
    static {  
        System.out.println("SB-A");  
        System.out.println(j);  
        //System.out.println(i); ---> Error  
        A a = new A();  
        System.out.println(a.i);  
    }  
}
```

```

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

```

SB-A

20

10

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

Ans:

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

EX:

```

class Test{
    static{
        System.out.println("Welcome To Durga Software Solutions");
        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 Durga Software Solutions

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>
```

Note: When we execute the Java program, JVM will load main class bytecode to the memory, at the time of loading main class bytecode JVM will execute the provided static block, JVM will display the required message, when JVM executes `System.exit(0)` JVM will terminate the program execution immediately.

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 the main class bytecode to the memory without checking the `main()` method availability, but in the case of JAVA 1.7 version and above JVM will load main class bytecode to the memory after checking the `main()` method availability in the main class, if the `main()` method is not available in the main class then JVM will not load main class bytecode to the memory.

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

Ans:

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

EX:

```
class Test{
    static Object obj = new Object(){
        {
            System.out.println("Welcome To Durga Software Solutions");
            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 Durga Software Solutions
```

```
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>
```

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 the main class bytecode to the memory without checking the main() method availability, but in the case of JAVA 1.7 version and above JVM will load main class bytecode to the memory after checking the main() method availability in the main class, if the main() method is not available in the main class then JVM will not load main class bytecode to the memory.

Static import:

In java applications, to make available classes and interfaces of a particular package to the present java file we have to use “import” statements.

IN Java applications, if we want to import all static members of a particular class into the present java file in order to access them without using class name and reference variable we have to use “Static Import” .

Syntax:

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

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

EX: `import static java.lang.Thread.*;`

`import static packageName.ClassName.memberName;`

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

EX: `import static java.lang.Thread.MIN_PRIORITY;`

EX:

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

1
5
10

Static Context:

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

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

In the above three elements, only static variables and static blocks are recognized and executed automatically at the time of loading the class bytecode to the memory, where the static methods will be recognized and executed the moment when we access that method.

EX:

```

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

```

SB-A
m1-A

EX:

```

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

```

m1-A
SB-A

EX:

```

class A{

```

```

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

```

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

EX:

```

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

```

```

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

```

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();
    }
}

```



```
}  
}
```

In the above program, when we create an object for the class by using “new” keyword then JVM will perform class loading then Object creating, but as per the requirement we want to load the class bytecode to the memory only we dont want to create object.

In Java applications, if we want to load class bytecode to the memory without creating an object we have to use the following method from java.lang.Class.

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

EX:

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

When we execute the forName() method, JVM will perform the following actions.

1. Jvm will take the provided class name and search for its .class file at the current location, at the Java predefined library and at the locations referred by the classpath environment variable.
2. If the required .class file does not exist at either of the above locations then the JVM will raise an exception like java.lang.ClassNotFoundException.
3. If the required .class file exists at either of the above locations then JVM will not raise any exception and JVM will load its bytecode to the memory.
4. After loading the class bytecode to the memory, JVM will create java.lang.Class object with the metadata of the loaded class, where the metadata includes class name, access modifiers, superclass metadata,

implemented interfaces metadata, variables details, methods details, constructors details,.....

newInstance() method:

In Java applications, if we load a particular class bytecode to the memory and if we want to create an object for the loaded class then we have to use the following method from java.lang.Class.

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

EX:

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

When we execute the above cls.newInstance() method, JVM Will perform the following actions.

1. JVM will goto the Class object where the metadata of the loaded class is available.
2. JVM will search for the non private and 0-arg constructor in the loaded class.
3. If the loaded class contains the non private and 0-arg constructor then JVM will execute that constructor and JVM will create the loaded class object and return the generated object in the form of Object type.
4. If the constructor is not a 0-arg constructor then JVM will raise an exception like java.lang.InstantiationException.
5. If the constructor is a private constructor then JVM will raise an exception like java.lang.IllegalAccessException.
6. If the constructor is not a 0-arg constructor and a private constructor then the JVM will raise an exception like java.lang.InstantiationException.

EX:

```
class A{  
    static {  
        System.out.println("Class Loading.....");  
    }  
    A(){  
        System.out.println("Object Creating.....");  
    }  
}
```

```

    }
}
public class Main {
    public static void main(String[] args) throws Exception {
        Class cls = Class.forName("A");
        Object obj = cls.newInstance();

    }
}

```

Class Loading.....

Object Creating.....

EX:

```

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

    }
}

```

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

EX:

```

class A{
    static {
        System.out.println("Class Loading.....");
    }
    private A(){
        System.out.println("Object Creating.....");
    }
}
public class Main {
    public static void main(String[] args) throws Exception {

```

```

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

    }
}

```

Exception in thread "main" java.lang.IllegalAccessException: class Main cannot access a member of class A with modifiers "private"
 at
 java.base/jdk.internal.reflect.Reflection.newIllegalAccessException(Reflection.java:392)

EX:

```

class A{
    static {
        System.out.println("Class Loading.....");
    }
    private A(int i){
        System.out.println("Object Creating.....");
    }
}

public class Main {
    public static void main(String[] args) throws Exception {
        Class cls = Class.forName("A");
        Object obj = cls.newInstance();

    }
}

```

Exception in thread "main" java.lang.InstantiationException: A
 at java.base/java.lang.Class.newInstance(Class.java:639)
 at Main.main(Main.java:12)

In general, we will use the Class.forName() and newInstance() method in the following situations.

1. In JDBC, we need to load the driver class without creating an object for the Driver class, here we will use the Class.forName() method.
2. In general, all the server side components like Servlets, EJBs,... will be executed on the basis of their life cycle activities, for each and every server side component there are two cycle phases like Loading and

Instantiation. In this case, to perform loading we will use the `Class.forName()` method and to perform Instantiation we will use the `newInstance()` method.

Factory Methods:

Factory method is a Java method, it will return either the current class object or some other class object.

EX:

```
class A{
    private A(){
        System.out.println("A-Con");
    }
    public void m1(){
        System.out.println("m1-A");
    }
    static A getRef(){
        A a = new A();
        return a;
    }
}

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

A-Con

m1-A

There are two types of Factory Methods.

1. Static Factory Methods.
2. Instance Factory Method.

Static Factory Method:

It is a static method, it will return either the same class object or any other class object.

EX:

```
Class cls = Class.forName(--);
NumberFormat nf = NumberFormat.getInstance();
DateFormat df = DateFormat.getDateInstance();
```

```
Connection con = DriverManager.getConnection(--,--,--);
```

Instance Factory Method:

If any instance method returns an object then that instance method is called an instance factory method.

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

EX:

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

Singleton class:

If any Java class allows the creation of only one object for it then that java 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 a class.
2. Declare a private Constructor inside the class.
3. Prepare 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 that object reference .
 - c. If an object exists for the current class already then return the existing object reference.
4. In Main class, in main() method, Access Factory method multiple times and display the generated reference value and check whether the same reference value is displayed or not multiple times.

EX:

```
class A{
    static A a = null;
    private A(){
    }
    static A getInstance(){
        if(a == null){
```

```

        a = new A();
    }
    return a;
}
}
public class Main {
    public static void main(String[] args){
        A a1 = A.getInstance();
        A a2 = A.getInstance();
        A a3 = A.getInstance();
        System.out.println(a1);
        System.out.println(a2);
        System.out.println(a3);
    }
}

```

A@1b28cdfa

A@1b28cdfa

A@1b28cdfa

EX:

```

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

```

A@1b28cdfa
A@1b28cdfa
A@1b28cdfa

EX:

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

    private A(){
    }
    static A getInstance(){
        return a;
    }
}

public class Main {
    public static void main(String[] args){
        A a1 = A.getInstance();
        A a2 = A.getInstance();
        A a3 = A.getInstance();
        System.out.println(a1);
        System.out.println(a2);
        System.out.println(a3);
    }
}
```

A@1b28cdfa
A@1b28cdfa
A@1b28cdfa

In MVC[Model-View-Controller] based applications we have to use a class as a controller , as per the MVC rules and regulations only one controller must be provided per application, So the class which we used for the controller must allow to create only one object, So the controller class must be a Singleton class.

EX-1: Struts is a MVC based web framework, it uses ActionServer a servlet class as controller, it must be a Singleton class.

EX-2: JSF is a MVC based web framework, it uses FacesServlet a Servlet class as controller, it must be a singleton class.

EX-3: Spring WEB MVC modules is a MVC based Framework, it uses DispatcherServlet a servlet class as controller, so it must be a singleton class.

‘final’ keyword:

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

There are three ways to utilize the ‘final’ keyword.

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

‘final’ Variables:

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

final variables are not allowing re-assignments.

```
final int i = 0;
```

```
i = 10; -----> Error
```

EX: In a bank application, if we create an account then the account number must be generated , where the account number is not modifiable once it is created, the variable which we used for the account number must be a final variable.

In general, in loops, loop variables must not be declared as final variables , because loop variables must have the operations like increment / decrement,.....

EX:

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

Status: Compilation Error

final methods:

It is a Java method, it must not allow overriding, because method overriding will change the method functionality.

In Java applications, if we want to fix a particular functionality to the method then we must declare that method as the final method.

EX: In a Bank application, to return minBal value we will use getMinBal() method, it must return the fixed value every time depending on the bank, so getMinBal() method must be a final method.

EX:

```
class A{
    void m1(){
        -----X-impl-----
    }
}
class B extends A{
    void m1(){
        ---Y-impl-----
    }
}
```

Status: Valid

EX:

EX:

```
class A{
    final void m1(){
        -----X-impl-----
    }
}
class B extends A{
    void m1(){
        ---Y-impl-----
    }
}
```

Status: invalid

EX:

```
class A{
    void m1(){
        -----X-impl-----
    }
}
class B extends A{
    final void m1(){
        ---Y-impl-----
    }
}
```

Status: Valid

final class:

It is a normal java class, it must not have subclasses.

In general, inheritance superclasses are never declared as final.

IN general, every class will have a particular purpose, by taking subclasses we are able to change the purpose of the superclass, if we want to fix a particular purpose to a class then we must declare that class as final class, if we declare any class as final class then it is possible to change its purpose by taking subclasses.

EX: java.lang.System class is final class.

EX: java.lang.String class is final class.

EX:

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

Status: Invalid

EX:

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

Status: Valid

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

Where the main purpose of declaring constant variables as public is to make available constant variables throughout the application.

Where the main purpose of declaring constant variables as static is to access constant variables without creating objects, just by using class names.

Where the main purpose of declaring constant variables as a final is to fix a particular value throughout the application.

EX:

In Thread class

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

In System class

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

EX:

```
class UserStatus{
    public static final String AVAILABLE = "User is Available";
    public static final String BUSY = "User is Busy";
    public static final String IDLE = "User is 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);
    }
}
```

To declare constant variables in java applications by using the above approach we are able to get the following problems.

1. We must declare every constant with the public static final explicitly.

2. This approach allows different data types to declare the constants , it will reduce typedness in java applications, but Java is a strongly typed programming language.
3. In this approach, when we access constant variables , automatically the values of the constant variables are displayed, where the value of the constant variables may or may not reflect the actual intention of the constant variables.

To overcome all the above problems we have to use “enum” as an alternative to declare constant variables.

Inside the enum,

1. All constant variables are by default “public static final”, no need to declare explicitly.
2. All constant variables are by default the same enum type, no need to provide data types explicitly, it will increase typedness in java applications and it allows typesafe operations in the java applications.
3. All constant variables are by default Named constants, when we access a constant variable, automatically the constant variable name will be displayed instead of its value.

Syntax:

```
-----  
[accessModifier] enum enumName{  
    ----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);  
    }  
}
```

If we compile the above enum, the compiler will translate the enum into a final class like below.

```

final class UserStatus extends java.lang.Enum{
    public static final UserStatus AVAILABLE;
    public static final UserStatus BUSY;
    public static final UserStatus IDLE;
    -----
}

```

Note: Inheritance is not possible between enums, because enum is by default a final class.

Note: In Java every enum must be a child class to java.lang.Enum, in turn java.lang.Enum is a child class to java.lang.Object.

```

PS D:\FullstackJava830\JAVA830\INTELLIJ-APPS\app01\src> javac Main.java

```

```

PS D:\FullstackJava830\JAVA830\INTELLIJ-APPS\app01\src> java Main

```

AVAILABLE

BUSY

IDLE

```

PS D:\FullstackJava830\JAVA830\INTELLIJ-APPS\app01\src> javap UserStatus

```

Compiled from "Main.java"

```

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

```

```

PS D:\FullstackJava830\JAVA830\INTELLIJ-APPS\app01\src>

```

Q)Is it possible to provide normal variables, normal constructors and normal methods like a class inside the enum?

Ans:

Yes, it is possible to declare normal variables, normal methods and normal constructors inside the enum like a class along with constant variables.

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 Cost    : "+Apple.A.getPrice());
        System.out.println("B-Grade Apple Cost    : "+Apple.B.getPrice());
        System.out.println("C-Grade Apple Cost    : "+Apple.C.getPrice());
    }
}

```

```

A-Grade Apple Cost    : 500
B-Grade Apple Cost    : 300
C-Grade Apple Cost    : 100

```

If we compile the above enum Apple then the compiler will translate the Apple enum into the following final class.

```

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

```

EX:

```

enum Notebook{
    A(500, 200),B(300, 100),C(100, 60);

    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 Notebook : Pages : 500 Price : 200

B-Grade Notebook : Pages : 300 Price : 100

C-Grade Notebook : Pages : 100 Price : 60

EX:

```

enum Notebook{
    A(500, 200),B(300, 100),C(100, 60);

    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());*/
        NoteBook[] noteBooks = NoteBook.values();
        System.out.println("GRADE\tPAGES\tPRICE");
        System.out.println("-----");
        for (NoteBook noteBook: noteBooks){

System.out.println(noteBook+"\t\t"+noteBook.getPages()+"\t\t"+noteBook.getPri
ce());
        }
    }
}

```

GRADE	PAGES	PRICE
A	500	200
B	300	100
C	100	60

Importance of the main() method in Java:

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 to the JVM to enter into the application and to execute the application.
2. To manage the actual application logic that must be recognized and executed by the JVM automatically.

3. To define starting point and ending point to the application execution we have to use main() method, because main() method starting point is the starting point of the application execution and main() method ending point is the ending point of the application execution.

Syntax:

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

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

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

Ans:

In general, to execute Java applications, JVM must access main() method, to access main() method by JVM, first main() method scope must be available to JVM, to bring main() method scope to the JVM we have to use the following cases.

Case#1:

If the main() method is private then the main() method will have scope up to the respective main class, but it will not be available to the JVM existing inside the JAVA software in C drive.

Case#2:

If the main() method is <default> then the main() method will have scope up to the respective package, but it will not be available to the JVM existing inside the JAVA software in C drive.

Case#3:

If the main() method is Protected then the main() method will have scope up to the respective package and the child classes existed in the other packages, but it will not be available to the JVM existing inside the JAVA software in C drive.

Case#4:

If the main() method is public then the main() method will have scope throughout the system including JVM existing in the Java software in C drive.

The main purpose of the public in main() prototype is to bring main() method scope to the JVM in order to access the main() method by JVM.

Note: In Java applications, if we declare a main() method without public then the compiler will not raise any error, but JVM will provide the following exceptions.

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

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

Ans:

As per the internal implementations of the JVM, JVM must access main() method by using the class name which we provided along with the java command in command prompt, in JAVA and J2EE applications only static methods are allowed to access by using the class name, so main() method must be static in order to access it by using the main class name by the JVM.

Note: In Java applications, if we declare main() method without using static keyword then the compiler will not raise any error, because the compiler will treat main() method as a normal java method and normal java methods may be declared 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)

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

Ans:

In Java applications, to execute the application JVM must access the main() method, here JVM will start application execution at the starting point of the main() method and JVM will stop the application execution at the ending point of the main() method , because Main thread will start program execution at starting point of the main() method and Main Thread will stop the program execution at the ending point of the main() method.

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

Note: In Java applications, if we declare the main() method without using “void” return type then the compiler will not raise any error, because the Compiler will take main() method as a normal java method , but JVM will provide the following exception messages.

JAVA6: java.lang.NoSuchMethodError: main

JAVA7: Error: Main method must return a value of type void in class Test,
Please define the main method as:public static void main(String[] args)

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

Note: Java creators have given the name to this method as “main” to reflect its importance in the java applications, on the basis of this naming convention only JVM was prepared , where JVM is able to access this method with the name “main” only.

Parameter to the main() method:

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

Ans:

In Java applications, as per the requirement we will provide input data to perform operations.

There are three types of input data we are able to provide to the java applications.

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

Static Input:

If we provide input data to the java applications at the time of writing a java program then that input data is called Static input.

EX:

```
class Math{
    int i = 10;// Static Input
    int j = 20;// Static Input
    public int add(){
        int result = i + j;
        return result;
    }
}
```

Dynamic Input:

If we provide input data to the Java applications at runtime then that input data is called a Dynamic Input.

```
D:\java830>javac Add.java
D:\java830>java Add
Enter First Value   : 10 ----> Dynamic Input
Enter Second Value  : 20 ----> Dynamic Input
ADD                : 30
```

Command Line Input:

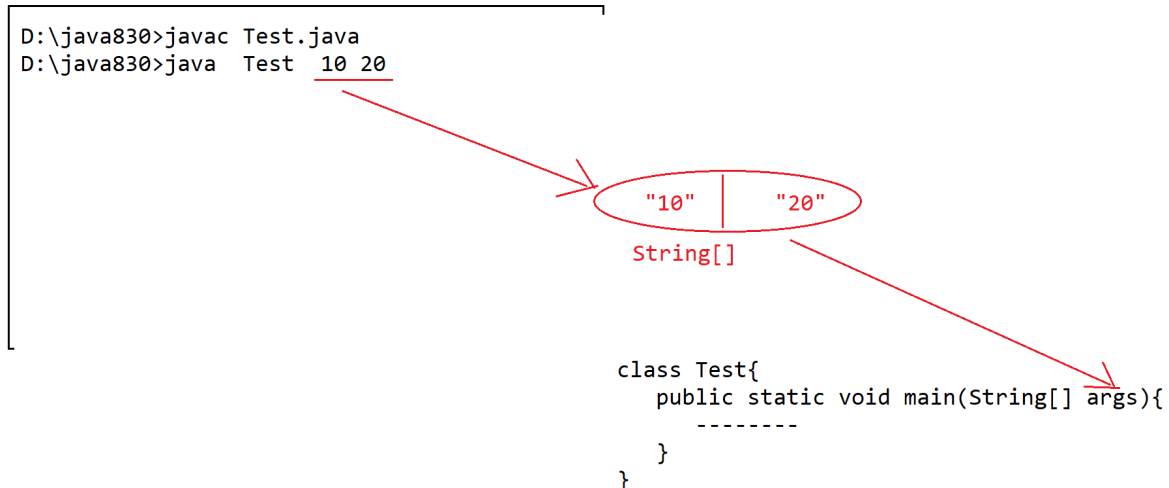
If we provide input data to the java program by providing input data along with the “java” command on command prompt then that input data is called the command line input.

EX:

```
D:\java830>javac Add.java
D:\java830>java Add 10 20 ----> Command Line Input.
ADD        : 30
```

If we provide command line input data to the java applications by providing input data along with java command then the JVM will read all the provided command line input from the java command , JVM will store all those command line input in the form of String[] and JVM will access the main() method by passing the generated String[] parameter.

The main purpose of the parameter to the main() method is to provide all the command line input to the java program in the form of String[].



EX:

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

```
D:\FullstackJava830\JAVA830>javac Test.java
D:\FullstackJava830\JAVA830>java Test 10 22.22f 33.333 'A' "abc" true
10
22.22f
33.333
'A'
abc
```

true

Note: To provide Command Line Arguments in the IntelliJ Idea IDE we have to edit the run configurations and we have to provide command line arguments in the "program Arguments" text field.

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

Ans:

From application to the application , from user to the user the types of the command line arguments are varied. Even though we provide the variable types of command line arguments our main() method parameter must hold all the types of command line arguments. In JAVA / J2EE applications only String data type is able to represent all the types of data, so to represent all the types of command line inputs we must provide String data type as parameter to the main() method.

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

Ans:

From application to the application , from user to the user the number of command line arguments are varied. Even though we provide the variable number of command line arguments our main() method parameter must hold all the number of command line arguments. In JAVA / J2EE applications Arrays are able to represent more than one element, so to represent the number of command line inputs we must provide array type as parameter to the main() method.

Note: In Java applications, if we declare main() method without the String[] parameter then the compiler will not raise any error, because the compiler will treat main() method as a normal java method, in java application we can declare methods without String[] parameter, but JVM will provide the following error messages.

JAVA 6: java.lang.NoSuchMethodError: main

JAVA 7: Error: Main method not found in class Test, please define the main method as: public static void main(String[] args)

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 java application , but not in a single java class, but they must be provided in the multiple java classes.

In the above context, if we provide more than one main() method in multiple classes then the java class which we provide along with the “java” command on the command prompt that class provided the main() method will be executed.

EX:

File Name : Test.java

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

D:\FullstackJava830\JAVA830>java A
main()-A


```
D:\FullstackJava830\JAVA830>java B
main()-B
```

```
D:\FullstackJava830\JAVA830>java C
main()-C
```

In the application, we can access one class main() method from the other class main() method just by using class name and by passing String[] parameter.

EX:

File Name: Test.java

```
class A{
    public static void main(String[] args){
        System.out.println("main()-A");
        String[] str = {"AAA","BBB"};
        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:\FullstackJava830\JAVA830>javac Test.java
```

```
D:\FullstackJava830\JAVA830>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, but it is not possible to override the main() method, because in java static method overloading is possible , static method overriding is not possible.

EX:

File Name: Test.java

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

D:\FullstackJava830\JAVA830>javac Test.java

D:\FullstackJava830\JAVA830>java Test
main()-String[]-Param

Q)Find the valid main() method syntaxes from the following list?

Ans:

1. public static void main(String[] args) -----> Valid
2. public static void main(String[] abc) -----> Valid
3. public static void main(String[][] args) ----> Invalid
4. public static void main(String args[]) -----> Valid
5. public static void main(String []args)-----> Valid
6. public static void main(String ... args)-----> Valid
7. public static void main(string[] args) -----> Invalid
8. public static void Main(String[] args)-----> Invalid
9. public static int main(String[] args) -----> Invalid
10. public final void main(String[] args) ----> Invalid
11. public void main(String[] args) -----> Invalid
12. static void main(String[] args) -----> Invalid
13. static public void main(String[] args)----> Valid

Relationships in Java:

The main purpose of the Relationships between classes is

1. To improve communication between entities.
2. To improve data navigation between classes.
3. To improve COde reusability.

There are three types of Relationships.

1. HAS-A Relationship
2. IS-A Relationship
3. USES-A Relationship

Q)What is the difference between HAS-A Relationship and IS-A Relationship?

Ans:

HAS-A Relationship is representing associations between the classes, these associations are able to improve Communication between classes and the data navigation between classes.

IS-A Relationship is representing inheritance between the classes, where inheritance relation is able to provide code Reusability.

Associations In Java:

There are four types of associations.

1. One-To-One Association
2. One-To-Many Association
3. Many-To-One Association
4. Many-To-Many Association

To provide associations in java applications we have to declare one class reference variable[s] in the other class.

EX:

```
class Account{
```

```

        String accNo;
        String accHolderName;
        String accType;
        long balance;
    }

class Employee{
    int eno;
    String ename;
    float esal;
    String eaddr;
    Account account;// One-To-One Association
}

EX:
class Student{// Container
    String sid;
    String snake;
    String saddr;
    Course[] courses;//One-To-Many
}
class Course{// Contained
    String cid;
    String cname;
    String cfee;
    ---
    ---
}

```

In the Associations, the class which has some other class reference variable is called a Container class.

In the associations, the class which is declared in a Container class is called a Contained class.

In the association “Employee has an Account”, Employee class is acting as a Container class and the Account class is acting as a Contained class.

To establish the Associations between entity classes, declaring Contained class reference variables in the Container class is not sufficient, really we must create the Contained class object and we must inject the generated

Contained class object in the Container class, to achieve this we have to use the “Dependency Injection” Design Principle.

The process of injecting dependent objects in the Container class object is called the Dependency Injection.

There are two ways to achieve Dependency Injection in the Java applications.

1. Constructor Dependency Injection.
2. Setter Method Dependency Injection.

Constructor Dependency Injection:

The process of injecting dependent objects through the Containers constructor is called “Constructor Dependency Injection”.

EX:

```
class Account{
    String accNo;
    String accHolderName;
    String accType;
    long balance;
    -----
}

class Employee{
    int eno;
    String ename;
    float esal;
    String eaddr;
    Account account;// One-To-One Association
    public Employee(int eno, String ename, float esal, String eaddr,
    Account account){
        -----
        this.account = account;
    }
}

public class Test{
    p s v main(String[] args){
        Account account = new Account(-----);
        ---
        Employee emp = new Employee(--,--,--,--,account);
    }
}
```

```
    }  
}
```

Setter Method Dependency Injection:

The process of injecting a Dependent object into the Container object through a setter method is called Setter Method Dependency Injection.

```
class Account{  
    String accNo;  
    String accHolderName;  
    String accType;  
    long balance;  
    -----  
}  
  
class Employee{  
    int eno;  
    String ename;  
    float esal;  
    String eaddr;  
    Account account;// One-To-One Association  
    ---  
    public void setAccount(Account account){  
        this.account = account;  
    }  
    public Account getAccount(){  
        return account;  
    }  
}  
  
public class Test{  
    p s v main(String[] args){  
        Account account = new Account();  
        Employee emp = new Employee();  
        ---  
        ---  
        emp.setAccount(account);  
    }  
}
```

One-To-One Association:

It is a relation between entity classes, where one instance of an entity should be mapped with exactly one instance of the another entity.

EX: Every Employee must have exactly one Account.

Account.java

```
package com.durgasoft.entities;
```

```
public class Account {  
    private String accNo;  
    private String accHolderName;  
    private String accType;  
    private long balance;  
  
    public Account(String accNo, String accHolderName, String accType, long  
balance) {  
        this.accNo = accNo;  
        this.accHolderName = accHolderName;  
        this.accType = accType;  
        this.balance = balance;  
    }  
  
    public String getAccNo() {  
        return accNo;  
    }  
  
    public String getAccHolderName() {  
        return accHolderName;  
    }  
  
    public String getAccType() {  
        return accType;  
    }  
}
```

```

        public long getBalance() {
            return balance;
        }
    }
}

```

Employee.java

```
package com.durgasoft.entities;
```

```

public class Employee {
    private int eno;
    private String ename;
    private float esal;
    private String eaddr;
    private Account account;

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

        System.out.println("Account Details");
        System.out.println("-----");
        System.out.println("Account Number      : "+account.getAccNo());
        System.out.println("Account Holder Name : 
"+account.getAccHolderName());
        System.out.println("Account Balance     : "+account.getBalance());
    }
}

```



```

}
Main.java
import com.durgasoft.entities.Account;
import com.durgasoft.entities.Employee;

public class Main {
    public static void main(String[] args) {
        Account account = new Account("abc123", "Durga", "Savings", 50000);
        Employee employee = new Employee(111, "Durga", 25000, "Hyd", account);
        employee.getEmployeeDetail();
    }
}

```

EX:

```

Account.java
import com.durgasoft.entities.Account;
import com.durgasoft.entities.Employee;

public class Main {
    public static void main(String[] args) {
        Account account = new Account();
        account.setAccNo("abc123");
        account.setAccHolderName("Durga");
        account.setAccType("Savings");
        account.setBalance(50000);

        Employee employee = new Employee();
        employee.setEno(111);
        employee.setEname("Durga");
        employee.setEsal(25000);
        employee.setEaddr("Hyd");
        employee.setAccount(account);

        employee.getEmployeeDetail();
    }
}

```

```

Employee.java
package com.durgasoft.entities;

public class Employee {
    private int eno;
    private String ename;
}

```

```
private float esal;
private String eaddr;
private Account account;

public int getEno() {
    return eno;
}

public void setEno(int eno) {
    this.eno = eno;
}

public String getName() {
    return ename;
}

public void setName(String ename) {
    this.ename = ename;
}

public float getEsal() {
    return esal;
}

public void setEsal(float esal) {
    this.esal = esal;
}

public String getEaddr() {
    return eaddr;
}

public void setEaddr(String eaddr) {
    this.eaddr = eaddr;
}

public Account getAccount() {
    return account;
}

public void setAccount(Account account) {
    this.account = account;
}
```

```

public void getEmployeeDetail(){
    System.out.println("Employee Details");
    System.out.println("-----");
    System.out.println("Employee NUmber      : "+eno);
    System.out.println("Employee Name       : "+ename);
    System.out.println("Employee Salary     : "+esal);
    System.out.println("Employee Address    : "+eaddr);
    System.out.println();

    System.out.println("Account Details");
    System.out.println("-----");
    System.out.println("Account Number      : "+account.getAccNo());
    System.out.println("Account Holder Name : 
"+account.getAccHolderName());
    System.out.println("Account Balance     : "+account.getBalance());
}
}

```

Main.java

```

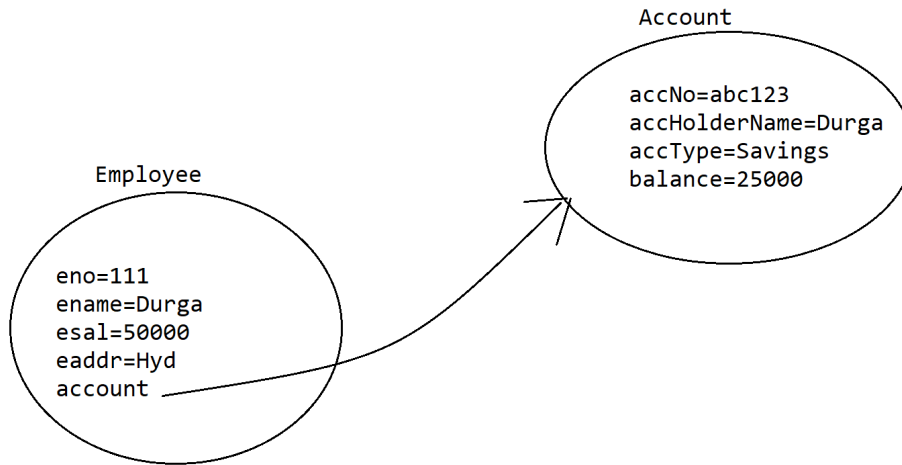
import com.durgasoft.entities.Account;
import com.durgasoft.entities.Employee;

public class Main {
    public static void main(String[] args) {
        Account account = new Account();
        account.setAccNo("abc123");
        account.setAccHolderName("Durga");
        account.setAccType("Savings");
        account.setBalance(50000);

        Employee employee = new Employee();
        employee.setEno(111);
        employee.setEname("Durga");
        employee.setEsal(25000);
        employee.setEaddr("Hyd");
        employee.setAccount(account);

        employee.getEmployeeDetail();
    }
}

```



One-To-Many Association:

It is a relation between entity classes, where one instance of an entity should be mapped with multiple instances of another entity.

EX: Single Order has multiple Items

EX:

Item.java

```
package com.durgasoft.entities;
```

```
public class Item {  
    private String itemId;  
    private String itemName;  
    private int itemCost;  
  
    public Item(String itemId, String itemName, int itemCost) {  
        this.itemId = itemId;  
        this.itemName = itemName;  
        this.itemCost = itemCost;  
    }  
  
    public String getItemId() {  
        return itemId;  
    }  
  
    public String getItemName() {
```

```

        return itemName;
    }

    public int getItemCost() {
        return itemCost;
    }
}

```

Order.java

```
package com.durgasoft.entities;
```

```

public class Order {
    private String orderId;
    private String orderName;
    private String deliveryLocation;
    private Item[] items;

    public Order(String orderId, String orderName, String deliveryLocation,
Item[] items) {
        this.orderId = orderId;
        this.orderName = orderName;
        this.deliveryLocation = deliveryLocation;
        this.items = items;
    }
    public void getOrderDetails(){
        System.out.println("Order Details");
        System.out.println("-----");
        System.out.println("Order Id          : "+orderId);
        System.out.println("Order Name       : "+orderName);
        System.out.println("Delivery Location : "+deliveryLocation);
        System.out.println();
        System.out.println("ItemId\tItemName\tItemCost");
        System.out.println("-----");
        for (Item item: items){
            System.out.print(item.getItemId()+"\t\t");
            System.out.print(item.getItemName()+"\t\t");
            System.out.print(item.getItemCost()+"\n");
        }
    }
}
}

```

Main.java

```

import com.durgasoft.entities.Item;
import com.durgasoft.entities.Order;

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

        Item item1 = new Item("i-1", "Mobile", 25000);
        Item item2 = new Item("i-2", "Computer", 32000);
        Item item3 = new Item("i-3", "Laptop", 50000);
        Item[] items = {item1, item2, item3};

        Order order = new Order("O-111", "Electronics", "23/3rt, S R Nagar,
Hyd-38", items);
        order.getOrderDetails();
    }
}

```

Order Details

```

-----
Order Id           : O-111
Order Name         : Electronics
Delivery Location  : 23/3rt, S R Nagar, Hyd-38

```

ItemId	ItemName	ItemCost
i-1	Mobile	25000
i-2	Computer	32000
i-3	Laptop	50000

EX:

Item.java

```

package com.durgasoft.entities;

public class Item {
    private String itemId;
    private String itemName;
    private int itemCost;

    public void setItemId(String itemId) {
        this.itemId = itemId;
    }
}

```

```

    public void setItemName(String itemName) {
        this.itemName = itemName;
    }

    public void setItemCost(int itemCost) {
        this.itemCost = itemCost;
    }

    public String getItemId() {
        return itemId;
    }

    public String getItemName() {
        return itemName;
    }

    public int getItemCost() {
        return itemCost;
    }
}

```

Order.java

```
package com.durgasoft.entities;
```

```

public class Order {
    private String orderId;
    private String orderName;
    private String deliveryLocation;
    private Item[] items;

    public String getOrderId(){
        return orderId;
    }

    public void setOrderId(String orderId) {
        this.orderId = orderId;
    }

    public String getOrderName() {
        return orderName;
    }
}

```

```

public void setOrderName(String orderName) {
    this.orderName = orderName;
}

public String getDeliveryLocation() {
    return deliveryLocation;
}

public void setDeliveryLocation(String deliveryLocation) {
    this.deliveryLocation = deliveryLocation;
}

public Item[] getItems() {
    return items;
}

public void setItems(Item[] items) {
    this.items = items;
}

public void getOrderDetails(){
    System.out.println("Order Details");
    System.out.println("-----");
    System.out.println("Order Id          : "+orderId);
    System.out.println("Order Name       : "+orderName);
    System.out.println("Delivery Location : "+deliveryLocation);
    System.out.println();
    System.out.println("ItemId\tItemName\tItemCost");
    System.out.println("-----");
    for (Item item: items){
        System.out.print(item.getItemId()+"\t\t");
        System.out.print(item.getItemName()+"\t\t");
        System.out.print(item.getItemCost()+"\n");
    }
}
}

```

Main.java

```

import com.durgasoft.entities.Item;
import com.durgasoft.entities.Order;

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

```



```

Item item1 = new Item();
item1.setItemId("I-1");
item1.setItemName("Mobile");
item1.setItemCost(25000);

Item item2 = new Item();
item2.setItemId("I-2");
item2.setItemName("Computer");
item2.setItemCost(35000);

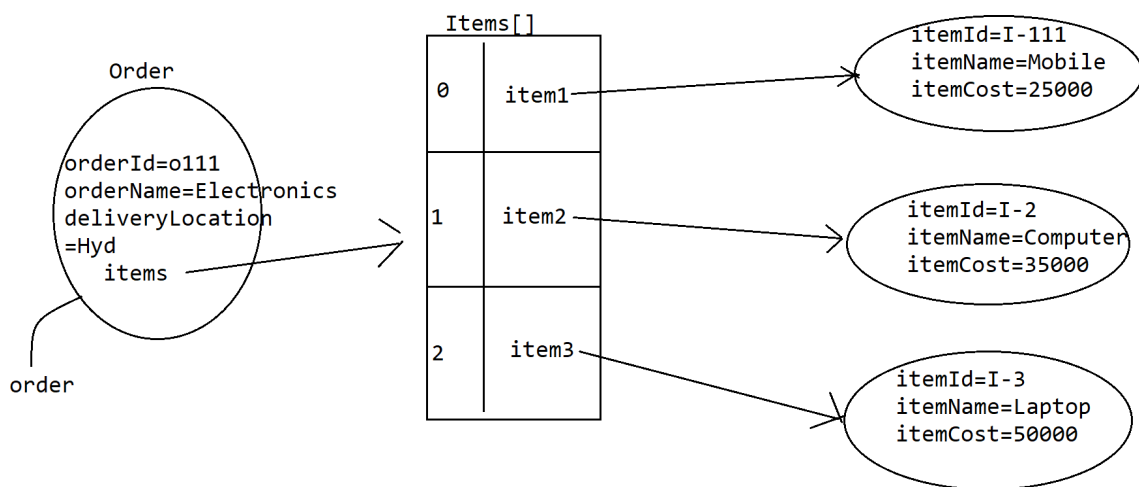
Item item3 = new Item();
item3.setItemId("I-3");
item3.setItemName("Laptop");
item3.setItemCost(50000);

Item[] items = {item1, item2, item3};

Order order = new Order();
order.setOrderId("O-111");
order.setOrderName("Electronics");
order.setDeliveryLocation("23/3rt, S R Nagar, Hyd-38");
order.setItems(items);

order.getOrderDetails();
}
}

```



Many-TO-One Association:

It is a relation between entity classes, where multiple instances of an entity should be mapped with a single instance of another entity.

EX:

Multiple Employees associated with a single Department.

Department.java

```
package com.durgasoft.entities;
```

```
public class Department {  
    private String departmentId;  
    private String departmentName;  
  
    public Department(String departmentId, String departmentName) {  
        this.departmentId = departmentId;  
        this.departmentName = departmentName;  
    }  
  
    public String getDepartmentId() {  
        return departmentId;  
    }  
  
    public String getDepartmentName() {  
        return departmentName;  
    }  
}
```

Employee.java

```
package com.durgasoft.entities;
```

```
public class Employee {  
    private int eno;  
    private String ename;  
    private float esal;  
    private String eaddr;  
  
    private Department department;  
  
    public Employee(int eno, String ename, float esal, String eaddr,  
        Department department) {
```

```

        this.eno = eno;
        this.ename = ename;
        this.esal = esal;
        this.eaddr = eaddr;
        this.department = department;
    }
    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("Department Id        : 
"+department.getDepartmentId());
        System.out.println("Department Name      : 
"+department.getDepartmentName());
    }
}

```

Main.java

```

import com.durgasoft.entities.Department;
import com.durgasoft.entities.Employee;

public class Main {
    public static void main(String[] args) {
        Department department = new Department("D111", "Admin");

        Employee employee1 = new Employee(111, "AAA", 5000, "Hyd",
department);
        Employee employee2 = new Employee(222, "BBB", 6000, "Hyd",
department);
        Employee employee3 = new Employee(333, "CCC", 7000, "Hyd",
department);

        employee1.getEmployeeDetails();
        System.out.println();

        employee2.getEmployeeDetails();
        System.out.println();

        employee3.getEmployeeDetails();
    }
}

```

```
}  
}
```

Employee Details

```
-----  
Employee Number    : 111  
Employee Name      : AAA  
Employee Salary    : 5000.0  
Employee Address   : Hyd  
Department Id      : D111  
Department Name    : Admin
```

Employee Details

```
-----  
Employee Number    : 222  
Employee Name      : BBB  
Employee Salary    : 6000.0  
Employee Address   : Hyd  
Department Id      : D111  
Department Name    : Admin
```

Employee Details

```
-----  
Employee Number    : 333  
Employee Name      : CCC  
Employee Salary    : 7000.0  
Employee Address   : Hyd  
Department Id      : D111  
Department Name    : Admin
```

EX:

Employee.java

```
package com.durgasoft.entities;
```

```
public class Employee {  
    private int eno;  
    private String ename;  
    private float esal;  
    private String eaddr;  
  
    private Department department;  
  
    public int getEno() {
```

```

        return eno;
    }

    public void setEno(int eno) {
        this.eno = eno;
    }

    public String getEname() {
        return ename;
    }

    public void setName(String ename) {
        this.ename = ename;
    }

    public float getEsal() {
        return esal;
    }

    public void setEsal(float esal) {
        this.esal = esal;
    }

    public String getEaddr() {
        return eaddr;
    }

    public void setEaddr(String eaddr) {
        this.eaddr = eaddr;
    }

    public Department getDepartment() {
        return department;
    }

    public void setDepartment(Department department) {
        this.department = department;
    }

    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("Department Id    : 
"+department.getDepartmentId());
        System.out.println("Department Name   : 
"+department.getDepartmentName());
    }
}

```

Department.java

```
package com.durgasoft.entities;
```

```

public class Department {
    private String departmentId;
    private String departmentName;

    public void setDepartmentId(String departmentId) {
        this.departmentId = departmentId;
    }

    public void setDepartmentName(String departmentName) {
        this.departmentName = departmentName;
    }

    public String getDepartmentId() {
        return departmentId;
    }

    public String getDepartmentName() {
        return departmentName;
    }
}

```

Main.java

```

import com.durgasoft.entities.Department;
import com.durgasoft.entities.Employee;

```

```

public class Main {
    public static void main(String[] args) {
        Department department = new Department();
        department.setDepartmentId("D-111");
        department.setDepartmentName("Admin");
    }
}

```

```
Employee employee1 = new Employee();
employee1.setEno(111);
employee1.setName("AAA");
employee1.setEsal(5000);
employee1.setEaddr("Hyd");
employee1.setDepartment(department);
```

```
Employee employee2 = new Employee();
employee2.setEno(222);
employee2.setName("BBB");
employee2.setEsal(6000);
employee2.setEaddr("Hyd");
employee2.setDepartment(department);
```

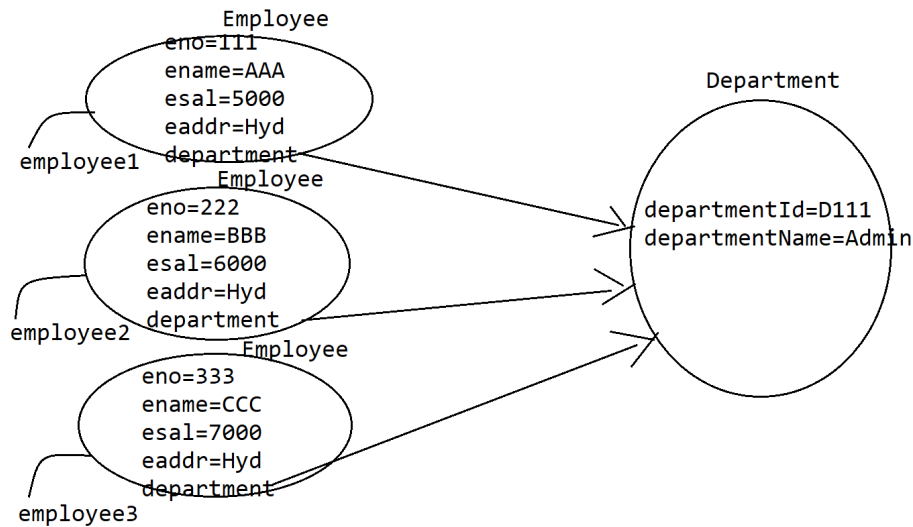
```
Employee employee3 = new Employee();
employee3.setEno(333);
employee3.setName("CCC");
employee3.setEsal(7000);
employee3.setEaddr("Hyd");
employee3.setDepartment(department);
```

```
employee1.getEmployeeDetails();
System.out.println();
```

```
employee2.getEmployeeDetails();
System.out.println();
```

```
employee3.getEmployeeDetails();
```

```
    }
}
```



Many-To-Many Association:

It is a relation between entity classes, where multiple instances of an entity should be mapped with multiple instances of another entity.

EX: Multiple Students have joined with multiple courses

EX:

Course.java

```
package com.durgasoft.entities;
```

```
public class Course {
    private String cid;
    private String cname;
    private int ccost;

    public Course(String cid, String cname, int ccost) {
        this.cid = cid;
        this.cname = cname;
        this.ccost = ccost;
    }

    public String getCid() {
        return cid;
    }
}
```



```

    public String getCname() {
        return cname;
    }

    public int getCcost() {
        return ccost;
    }
}

```

Student.java

```
package com.durgasoft.entities;
```

```

public class Student {
    private String sid;
    private String sname;
    private String saddr;
    private Course[] courses;

    public Student(String sid, String sname, String saddr, Course[] courses) {
        this.sid = sid;
        this.sname = sname;
        this.saddr = saddr;
        this.courses = courses;
    }

    public void getStudentDetails(){
        System.out.println("Student Details");
        System.out.println("-----");
        System.out.println("Student Id      : "+sid);
        System.out.println("Student Name    : "+sname);
        System.out.println("Student Address : "+saddr);
        System.out.println("CID\tCNAME\tCCOST");
        System.out.println("-----");
        for (Course course: courses){
            System.out.print(course.getCid()+"\t");
            System.out.print(course.getCname()+"\t");
            System.out.print(course.getCcost()+"\n");
        }
    }
}

```

Main.java

```
import com.durgasoft.entities.Course;
```

```

import com.durgasoft.entities.Student;

public class Main {
    public static void main(String[] args) {
        Course course1 = new Course("C-1", "JAVA ", 50000);
        Course course2 = new Course("C-2", "PYTHON", 30000);
        Course course3 = new Course("C-3", ".NET ", 20000);

        Course[] courses = {course1, course2, course3};

        Student student1 = new Student("S-111", "AAA", "Hyd", courses);
        Student student2 = new Student("S-222", "BBB", "Hyd", courses);
        Student student3 = new Student("S-333", "CCC", "Hyd", courses);

        student1.getStudentDetails();
        System.out.println();
        student2.getStudentDetails();
        System.out.println();
        student3.getStudentDetails();
    }
}

```

Student Details

```

-----
Student Id      : S-111
Student Name    : AAA
Student Address : Hyd
CID   CNAME CCOST
-----
C-1   JAVA      50000
C-2   PYTHON    30000
C-3   .NET      20000

```

Student Details

```

-----
Student Id      : S-222
Student Name    : BBB
Student Address : Hyd
CID   CNAME CCOST
-----
C-1   JAVA      50000
C-2   PYTHON    30000
C-3   .NET      20000

```

Student Details

Student Id : S-333
Student Name : CCC
Student Address : Hyd
CID CNAME CCOST

C-1	JAVA	50000
C-2	PYTHON	30000
C-3	.NET	20000

EX:

Course.java

```
package com.durgasoft.entities;
```

```
public class Course {  
    private String cid;  
    private String cname;  
    private int ccost;  
  
    public void setCid(String cid) {  
        this.cid = cid;  
    }  
  
    public void setCname(String cname) {  
        this.cname = cname;  
    }  
  
    public void setCcost(int ccost) {  
        this.ccost = ccost;  
    }  
  
    public String getCid() {  
        return cid;  
    }  
  
    public String getCname() {  
        return cname;  
    }  
  
    public int getCcost() {  
        return ccost;  
    }  
}
```

```
    }  
}
```

Student.java

```
package com.durgasoft.entities;
```

```
public class Student {  
    private String sid;  
    private String sname;  
    private String saddr;  
    private Course[] courses;  
  
    public String getSid() {  
        return sid;  
    }  
  
    public void setSid(String sid) {  
        this.sid = sid;  
    }  
  
    public String 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\tCNAME\tCCOST");
    System.out.println("-----");
    for (Course course: courses){
        System.out.print(course.getCid()+"\t");
        System.out.print(course.getCname()+"\t");
        System.out.print(course.getCcost()+"\n");
    }
}
}

```

Main.java

```

import com.durgasoft.entities.Course;
import com.durgasoft.entities.Student;

public class Main {
    public static void main(String[] args) {
        Course course1 = new Course();
        course1.setCid("C-1");
        course1.setCname("JAVA ");
        course1.setCcost(50000);

        Course course2 = new Course();
        course2.setCid("C-2");
        course2.setCname("PYTHON");
        course2.setCcost(30000);

        Course course3 = new Course();
        course3.setCid("C-3");
        course3.setCname(".NET ");
        course3.setCcost(20000);

        Course[] courses1 = {course1, course2, course3};
        Course[] courses2 = {course1, course2};
        Course[] courses3 = {course1, course3};

        Student student1 = new Student();
    }
}

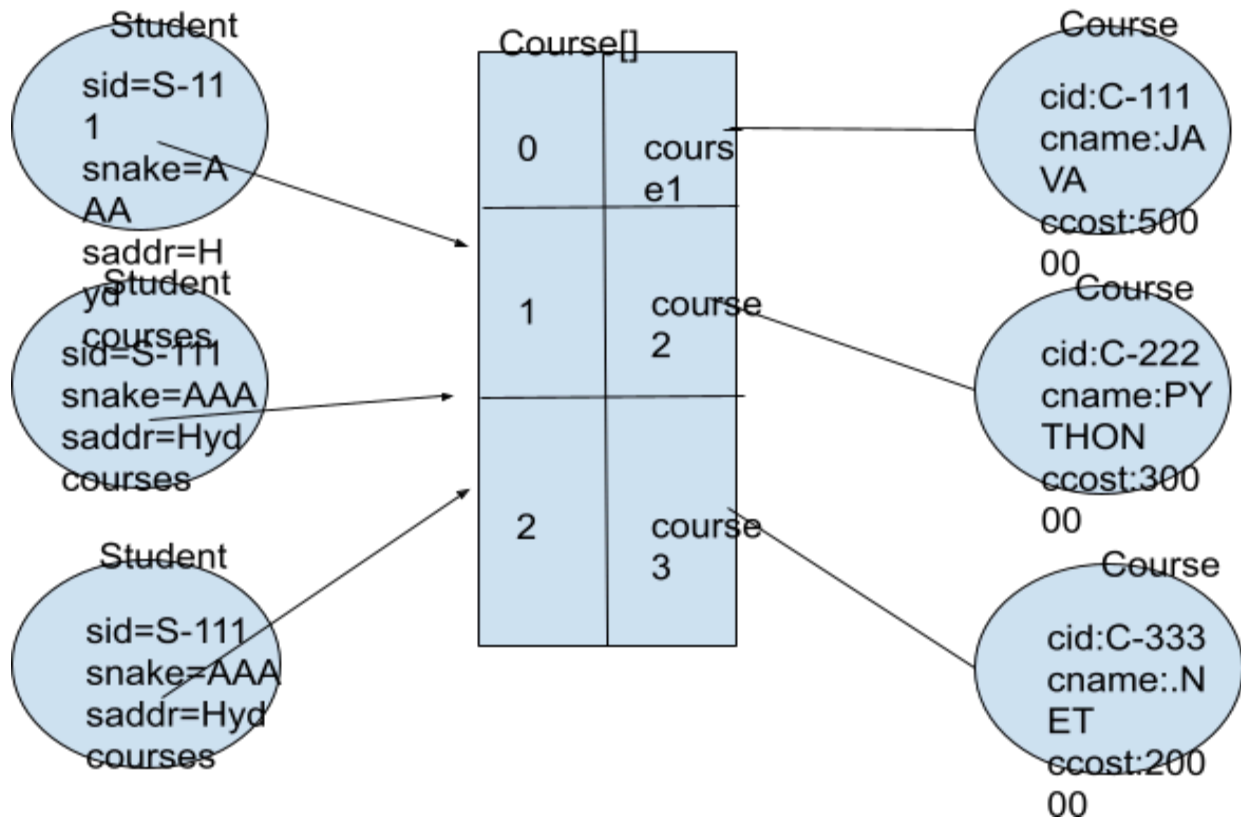
```

```
student1.setSid("S-111");
student1.setSname("AAA");
student1.setSaddr("Hyd");
student1.setCourses(courses1);

Student student2 = new Student();
student2.setSid("S-222");
student2.setSname("BBB");
student2.setSaddr("Hyd");
student2.setCourses(courses2);

Student student3 = new Student();
student3.setSid("S-111");
student3.setSname("AAA");
student3.setSaddr("Hyd");
student3.setCourses(courses3);

student1.getStudentDetails();
System.out.println();
student2.getStudentDetails();
System.out.println();
student3.getStudentDetails();
    }
}
```



In general, there are two ways to implement Associations.

1. Composition
2. Aggregation

Q)What is the difference between Composition and Aggregation?

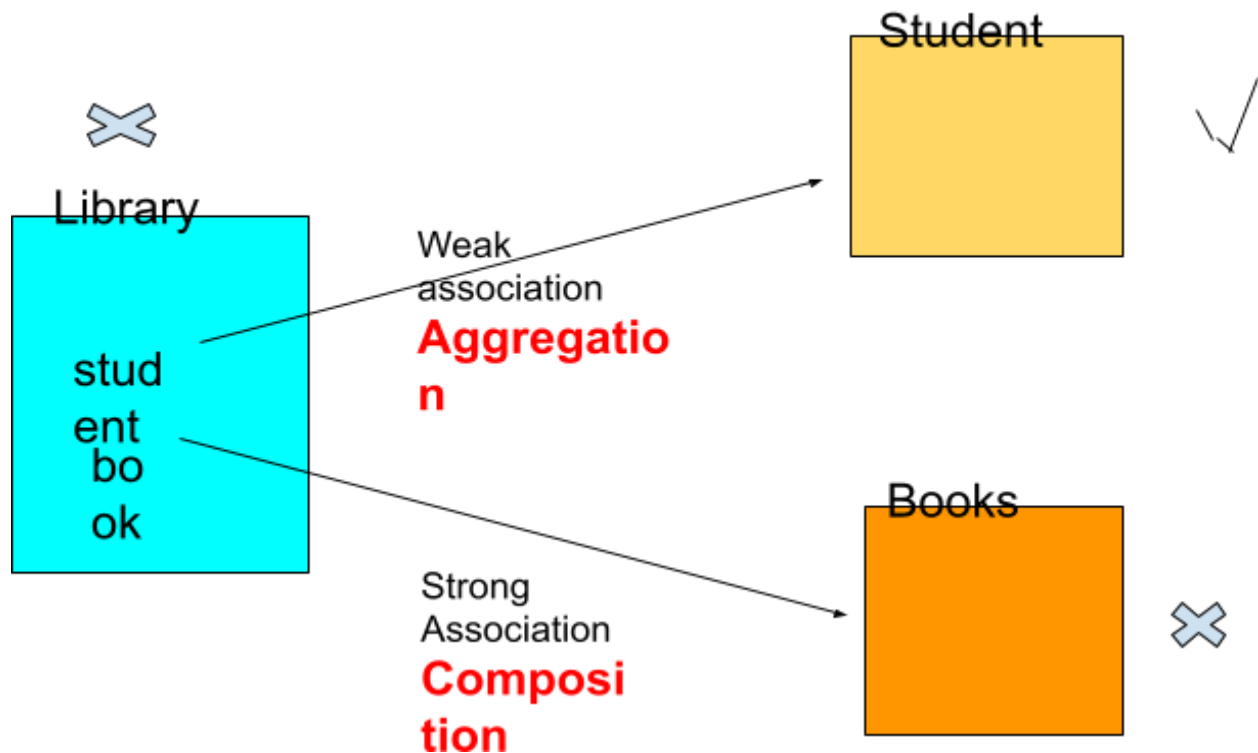
Ans:

Composition is a strong association between entities, if we destroy the Container entity, automatically the Contained entity is also destroyed. The lifetime of the Contained entity is dependent on the lifetime of the Container entity.

EX: If we consider the association between the entities Library and Books then it is Strong association that is Composition, If we close the Library then there is scope for the Books to exist.

Aggregation is a weak association between entities, even if we destroy the Container entity then there is a chance for the Contained entity to exist. The lifetime of the Contained entity is not dependent on the Container entity.

EX: If we consider the association between the entities Library and Student then it is Weak association that is Aggregation, If we close the Library then there is a scope for the Students to exist.



Inheritance in Java:

It is a relation between entities, it will provide variables and methods from one entity to the other entity.

In the above inheritance relation, the entity class which is providing variables and methods is called a Superclass.

In the above inheritance relation, the entity class which is taking variables and methods is called a Subclass.

The main advantage of the inheritance is "Code Reusability".

If we declare variables and methods in a superclass then that variables and methods are accessible in any subclass directly without creating objects for the superclass.

Java has represented inheritance by defining a keyword “extends”.

EX:

```
class Person{
    -----
}
class Student extends Person{
    -----
}
```

In inheritance, all superclass members are accessible in the subclasses, but all subclass members are not accessible in the superclass.

In Java, 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.

Initially, there are two types of inheritances.

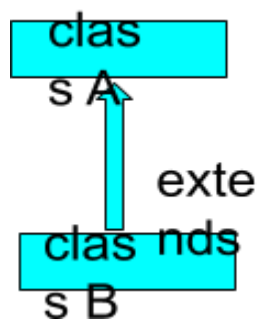
1. Single Inheritance
2. Multiple Inheritance

On the basis of the above two types of inheritances, three more inheritances are defined.

1. Multilevel Inheritance
2. Hierarchical Inheritance
3. Hybrid Inheritance

Single Inheritance:

It is a relation between entity classes, it will provide variables and methods from only one superclass to one or more subclasses.



Java does support Single Inheritance.

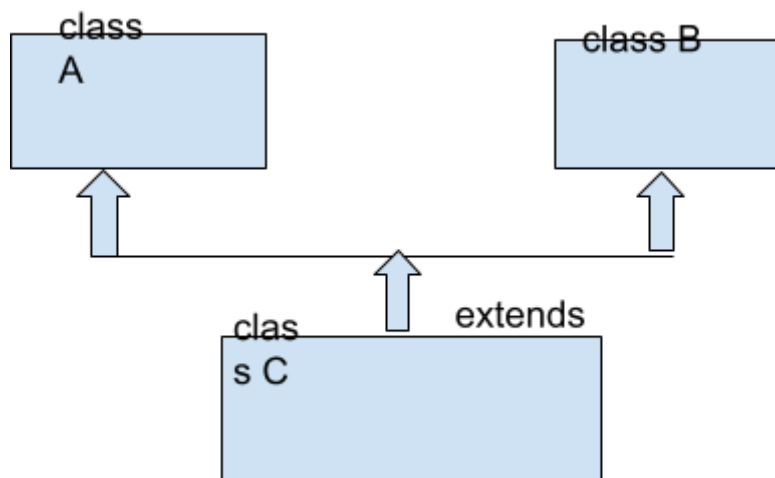
EX:

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

        B b = new B();
        b.m1();
        b.m2();
    }
}
```

Multiple Inheritance:

It is a relation between entity classes, it will provide variables and methods from more than one superclass to one or more subclasses.



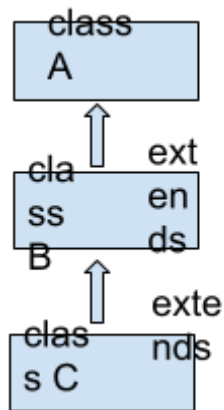
Java does not support Multiple Inheritance, because if we declare the same variable with different values and the same method with different

implementations in both the superclasses and if we access that variable and method in the subclass then which superclass variable would be accessed and which superclass method would be accessed is a confusion state.

Multilevel Inheritance:

It is the combination of the Single inheritances in more than one level.

Java does support Multilevel Inheritance.



EX:

```
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();
        System.out.println();
    }
}
```

```

        B b = new B();
        b.m1();
        b.m2();
        System.out.println();

        C c = new C();
        c.m1();
        c.m2();
        c.m3();
    }
}

```

m1-A

m1-A

m2-B

m1-A

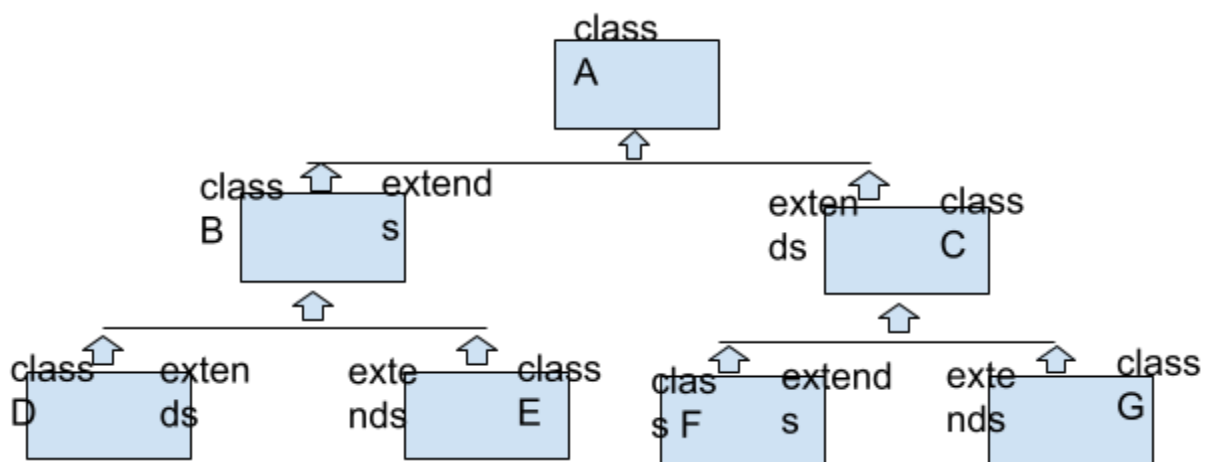
m2-B

m3-C

Hierarchical Inheritance:

It is the combination of the single inheritances in a particular hierarchy.

Java does support Hierarchical Inheritance.



EX:

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

            B b = new B();
            b.m1();
            b.m2();
        }
    }
}
```

```

        System.out.println();

        C c = new C();
        c.m1();
        c.m3();
        System.out.println();

        D d = new D();
        d.m1();
        d.m2();
        d.m4();
        System.out.println();

        E e = new E();
        e.m1();
        e.m2();
        e.m5();
        System.out.println();

        F f = new F();
        f.m1();
        f.m3();
        f.m6();
        System.out.println();

        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

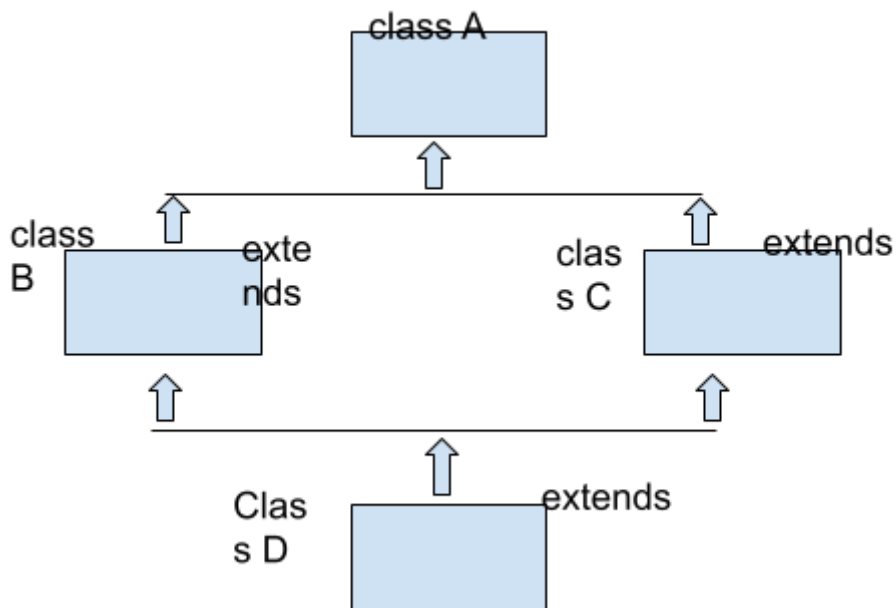
m1-A
m3-C
m6-F

m1-A
m3-C
m7-G

Hybrid Inheritance:

It is the combination of the Single inheritance and multiple inheritance.

Java does not support Hybrid inheritance, because it includes Multiple inheritance, it is not possible in Java.



EX:

```
class Person{  
    String pid;  
    String pname;  
    String paddr;  
    public void getPersonDetails(){
```

```

        System.out.println("Person Id      : "+pid);
        System.out.println("Person Name    : "+pname);
        System.out.println("Person Address : "+paddr);
    }
}
class Employee extends Person{
    int eno;
    float esal;
    public Employee(String pid, String pname,String paddr, int eno, float
esal){
        this.pid = pid;
        this.pname = pname;
        this.paddr = paddr;
        this.eno = eno;
        this.esal = esal;
    }
    public void getEmployeeDetails(){
        System.out.println("Employee Details");
        System.out.println("-----");
        getPersonDetails();
        System.out.println("Employee Number    : "+eno);
        System.out.println("Employee Salary    : "+esal);
    }
}
class Student extends Person{
    String sid;
    String sbranch;
    public Student(String pid, String pname, String paddr, String sid, String
sbranch){
        this.pid = pid;
        this.pname = pname;
        this.paddr = paddr;
        this.sid = sid;
        this.sbranch = sbranch;
    }
    public void getStudentDetails(){
        System.out.println("Student Details");
        System.out.println("-----");
        getPersonDetails();
        System.out.println("Student Id      : "+sid);
        System.out.println("Student Branch : "+sbranch);
    }
}

```



```

public class Main {
    public static void main(String[] args) {
        Employee employee = new Employee("p1", "Durga", "Hyd", 111, 25000);
        employee.getEmployeeDetails();
        System.out.println();

        Student student = new Student("P2", "Anil", "Chennai", "S-111",
"Computers");
        student.getStudentDetails();
    }
}

```

Employee Details

```

-----
Person Id      : p1
Person Name    : Durga
Person Address : Hyd
Employee Number : 111
Employee Salary : 25000.0

```

Student Details

```

-----
Person Id      : P2
Person Name    : Anil
Person Address : Chennai
Student Id     : S-111
Student Branch : Computers

```

Static Context in Inheritance:

In Java there are three elements in the Static Context.

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

Where Static variables and Static Blocks are executed automatically by JVM at the time of loading the respective class bytecode to the memory.

In inheritance, when we create an object for the subclass JVM must load subclass bytecode to the memory, but in Java there is a conventions like before loading subclass bytecode to the memory first JVM will load superclass bytecode to the memory.

In the case of inheritance, classes loading order will be from superclass to subclass order.

In the case of inheritance, if we provide static context at each and every class then JVM will execute them when the respective class bytecode is loaded.

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

```

        System.out.println("SB-A");
    }
}
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{
        System.out.println("SB-C");
    }
    static int k = m3();
}
public class Main {
    public static void main(String[] args) {
        C c1 = new C();
        C c2 = new C();
    }
}

```

m1-A
 SB-A
 SB-B
 m2-B
 SB-C
 m3-C

Instance Context in Inheritance:

In Java, 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 will be executed automatically just before executing the respective class constructors.

In the case of inheritance, when we create subclass object , JVM must execute subclass constructor, but before executing subclass constructor JVM must execute a 0-arg constructor in the superclass, after executing superclass constructor only JVM will execute subclass constructor.

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");
    }
    {
```

```

        System.out.println("IB-A");
    }
    int m1(){
        System.out.println("m1-A");
        return 10;
    }
    int i = m1();
}
class B extends A{
    {
        System.out.println("IB-B");
    }
    int m2(){
        System.out.println("m2-B");
        return 20;
    }
    int j = m2();
    B(){
        System.out.println("B-Con");
    }
}
class C extends B{
    int m3(){
        System.out.println("m3-C");
        return 30;
    }
    int k = m3();
    C(){
        System.out.println("C-Con");
    }
    {
        System.out.println("IB-C");
    }
}
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");
    }
    int m1(){
        System.out.println("m1-A");
        return 10;
    }
    static{
        System.out.println("SB-A");
    }
    int i = m1();
    {
        System.out.println("IB-A");
    }
    static int m2(){
        System.out.println("m2-A");
        return 20;
    }
    static int j = m2();
}
class B extends A{
    static{
        System.out.println("SB-B");
    }
    static int m3(){
        System.out.println("m3-B");
        return 30;
    }
    static int k = m3();
    B(){
        System.out.println("B-Con");
    }
    {
```

```

        System.out.println("IB-B");
    }
    int m4(){
        System.out.println("m4-B");
        return 40;
    }
    int l = m4();
}
class C extends B{
    static{
        System.out.println("SB-C");
    }
    {
        System.out.println("IB-C");
    }
    static int m5(){
        System.out.println("m5-C");
        return 50;
    }
    int m6(){
        System.out.println("m6-C");
        return 60;
    }
    static int m = m5();
    int n = m6();
    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

m2-A

SB-B

m3-B

SB-C

m5-C

m1-A
IB-A
A-Con
IB-B
m4-B
B-Con
IB-C
m6-C
C-Con

m1-A
IB-A
A-Con
IB-B
m4-B
B-Con
IB-C
m6-C
C-Con

‘super’ keyword:

‘super’ is a java keyword, it can be used to represent a superclass object from the subclasses.

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

1. To refer the superclass variables.
2. To refer the superclass methods.
3. To refer the superclass constructor.

Referring Superclass variables by using ‘super’ keyword:

In Java applications, if we want to refer to a superclass variable by using super keyword then we have to use the following syntax.

super.varName

Note: In Java applications, when we have the same set variables at local, at current class level and at the superclass level then to refer to superclass variables over the local and the current class variables we have to use ‘super’ Keyword.

EX:

```
class A{
    int i = 10;
    int j = 20;
}
class B extends A{
    int i = 30;
    int j = 40;
    B(int i, int j){
        System.out.println(i+" "+j);// 50 60
        System.out.println(this.i+" "+this.j);// 30 40
        System.out.println(super.i+" "+super.j);// 10 20
    }
}
public class Main {
    public static void main(String[] args) {
        B b = new B(50,60);
    }
}
```

```
50    60
30    40
10    20
```

Referring Superclass methods from Subclasses:

If we want to refer to a superclass method from subclasses by using super keyword then we have to use the following syntax.
super.methodName([ParamList]);

In Java applications, when we have the same methods at superclass and at the subclass and when we want to access only superclass methods over the subclass method there we will use 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 from the subclasses:

In general, in inheritance, when we access subclass constructor, JVM will execute a 0-arg constructor in the super class then JVM will execute the subclass constructor, in this context, if we want to execute a particular constructor in the superclass in place of the 0-arg constructor then we have to use “super(--)” in the subclass constructor.

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 Java applications, to access superclass constructor from the subclasses by using super keyword then we have to use the following conditions.

1. In the subclass constructors, super() statement must be the first statement.
2. super() statement must be used in the subclass constructors only, not 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); -----> Compilation Error
    }
}
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); ---> Compilation Error
        System.out.println("m1-A");
    }
}
public class Main {
    public static void main(String[] args) {
        B b = new B();
    }
}

```

Status: Compilation Error

Q)Is it possible to access more than one superclass constructor from a single subclass constructor by using super(--) statements?

 Ans:

 No, it is not possible to access more than one superclass constructor from a subclass constructor by using super() statements, because in the subclass constructors super() statement must be provided as the first statement.

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); -----> Compilation Error
        System.out.println("B-Con");
    }
}
public class Main {
    public static void main(String[] args) {
        B b = new B();
    }
}

```

Status: Compilation Error

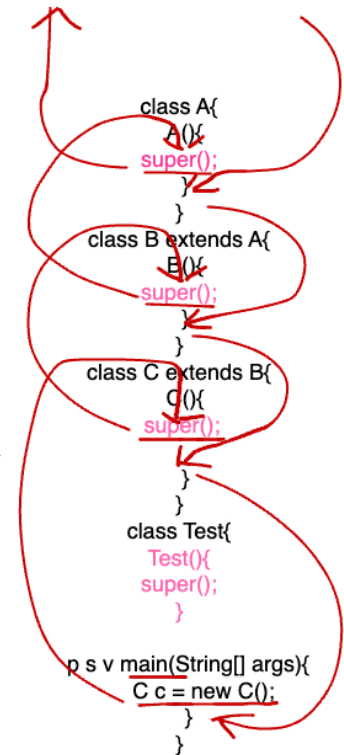
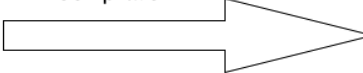
If we prepare a java file with the inheritance relation between classes and if we compile the java file then the compiler will perform the following actions.

1. Compiler will go to each and every class and checks whether any requirement to provide default constructor or not, if no user defined constructor is identified in the class then the compiler will provide the default constructor, if any user defined constructor is identified in the class then the compiler will not provide default constructor.
2. Compiler will go to each and every class and each and every constructor and the compiler checks whether the constructor contains "super()" statement or not to access superclass constructor, if no super() statement exists in the constructor then the compiler will provide "super()" statement in the constructor in order to access a 0-arg constructor in the superclass. If any super() statement exists then the compiler will not provide a "super()" statement in the constructor.
3. Compiler will go to the superclasses and check whether the right constructors are available or not as per the super(--) statements existed in the subclass constructors, if the right constructor is not available in the superclass then the compiler will raise an error.

EX:

```
class A{
  A(){
  }
}
class B extends A{
  B(){
  }
}
class C extends B{
  C(){
  }
}
class Test{
  p s v main(String[] args){
    C c = new C();
  }
}
```

Compilation



```
class A{
  A(){
    super();
  }
}
class B extends A{
  B(){
    super();
  }
}
class C extends B{
  C(){
    super();
  }
}
class Test{
  Test(){
    super();
  }
  p s v main(String[] args){
    C c = new C();
  }
}
```

EX:

```

class A{
    A(){
        Sopln("A-Con");
    }
}
class B extends A{
    B(int i){
        Sopln("B-Con");
    }
}
class C extends B{
    C(){
        Sopln("C-Con");
    }
}


```

```

class Test{
    p s v main(String[] args){
        C c = new C();
    }
}

```

Compilation Compilation Error



```

class Test{
    Test(){
        super();
    }
    p s v main(String[] args){
        C c = new C();
    }
}

```

```

class A{
    A(){
        super();
        Sopln("A-Con");
    }
}
class B extends A{
    B(int i){
        super();
        Sopln("B-Con");
    }
}
class C extends B{
    C(){
        super();
        Sopln("C-Con");
    }
}

```

EX:

```

class A{
    A(){
        Sopln("A-Con");
    }
}
class B extends A{
    B(int i){
        Sopln("B-int-param-Con");
    }
}
class C extends B{
    C(){
        super(10);
        Sopln("C-Con");
    }
}


```

```

class Test{
    p s v main(String[] args){
        C c = new C();
    }
}

```

Compilation



```

class Test{
    class Test{
        super();
    }
    p s v main(String[] args){
        C c = new C();
    }
}

```

A-Con
B-Int-Param-Con
C-Con

```

class A{
    A(){
        super();
        Sopln("A-Con");
    }
}
class B extends A{
    B(int i){
        super();
        Sopln("B-int-param-Con");
    }
}
class C extends B{
    C(){
        super(10);
        Sopln("C-Con");
    }
}

```

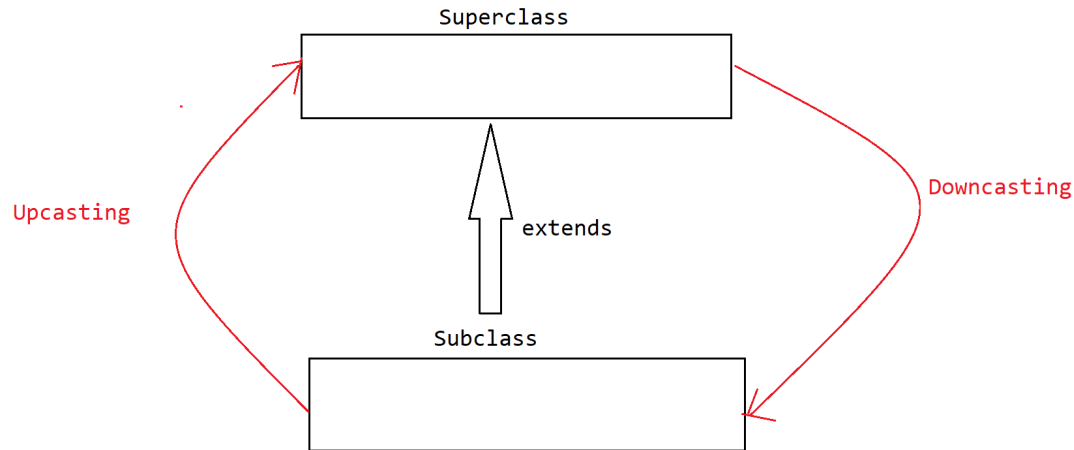
Class Level Type Casting:

The process of converting data from one user defined data type to the another user defined data type is called the User defined data types type casting or Class Level Type Casting.

In Java applications, to perform user defined data types Type casting we have to use either extends relation or implements relation between the two user defined data types.

There are two types of User defined data types type casting.

1. Upcasting
2. Downcasting



Upcasting:

The process of converting data from subclass type to the Superclass type is called Upcasting.

To perform Upcasting we have to assign a subclass reference variable to the superclass reference variable.

Note: In Java applications Upcasting will be used in the Method Overriding.

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();
        A a = b; // Upcasting
        a.m1();

        A a1 = new B(); // Upcasting
        a1.m1();
    }
}

```

Downcasting:

The process of converting data from superclass type to the Subclass type is called Downcasting.

To perform downcasting we have to use the following format.

```
P a = (Q)b;
```

Where Q must be either the same as the P or subclass to P.

To perform Downcasting we have to use the following cases.

```

class A{
}
class B extends A{
}

```

Case#1:

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

```
B b = a;
```

Status: Compilation Error

Reason: In Java , by default all subclass types are compatible with the superclass types, so we can assign subclass reference variables to the superclass reference variables, but superclass types are not compatible with the subclass types, so we are unable to assign superclass reference variables to the subclass reference variable directly.

Note: Still if we want to convert superclass type to the subclass type then we have to use cast operator.

Case#2:

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

```
B b = (B)a;
```

Status: No Compilation Error, but ClassCastException.

Reason: In Java applications, we can keep subclass object reference value in the superclass reference variable, but we are unable to keep superclass object 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, No Exception

When we compile the above code the compiler will perform the following actions.

1. Compiler will recognize the cast operator provided data type and the left side variable data type.
2. Compiler will check whether the cast operator provided data type is compatible with the left side variable data type or not.
3. If the cast operator provided data type is not compatible with the left side variable data type then it will raise an error like "Incompatible Types".
4. If the cast operator provided data type is compatible with the left side variable data type then the compiler will not raise any error and at the same time the compiler will not perform any type casting.

Note: In Java, the compiler is not responsible for the Type casting, it is responsible for the Type checking.

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

1. JVM will recognize the right side variable data type and the cast operator provided data type.
2. JVM will convert the right side variable data type to the cast operator provided data type, it is called Downcasting.
3. 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();
    }
}

```

Consider the following inheritance architecture.

```

class A{
}
class B extends A{
}
class C extends B{
}
class D extends C{
}

```

EX-1:

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

```
B b = a;
```

Status: Compilation Error

EX-2:

```
A a = new A();  
B b = (B)a;  
Status: No Compilation Error, ClassCastException
```

```
EX-3:  
A a = new B();  
B b = (B)a;  
Status: No Compilation Error, No Exception
```

```
EX-4:  
A a = new D();  
B b = (C)a;  
Status: No Compilation Error , No Exception
```

```
EX-5:  
A a = new C();  
B b = (D)a;  
Status: No Compilation Error, ClassCastException
```

```
EX-6:  
A a = new B();  
B b = (D)a;  
Status: No Compilation Error, ClassCastException
```

```
EX-7:  
A a = new D();  
D d = (D)(C)(B)a;  
Status: no Compilation Error, No Exception
```

```
EX-8:  
A a = new C();  
D d = (D)(C)(B)a;  
Status: No Compilation Error, ClassCastException
```

USES-A Relationship:

It is a relation between entity classes, where one entity class uses another entity class up to a particular behavior.

EX:
Account.java

```
package com.durgasoft.entities;

public class Account {
    private String accNo;
    private String accHolderName;
    private String accType;
    private long balance;

    public String getAccNo() {
        return accNo;
    }

    public void setAccNo(String accNo) {
        this.accNo = accNo;
    }

    public String getAccHolderName() {
        return accHolderName;
    }

    public void setAccHolderName(String accHolderName) {
        this.accHolderName = accHolderName;
    }

    public String getAccType() {
        return accType;
    }

    public void setAccType(String accType) {
        this.accType = accType;
    }

    public long getBalance() {
        return balance;
    }

    public void setBalance(long balance) {
        this.balance = balance;
    }
}
```

Transaction.java

```
package com.durgasoft.entities;

public class Transaction {
```

```

private String transactionId;

public Transaction(String transactionId) {
    this.transactionId = transactionId;
}

public void deposit(Account account, int depositAmount){
    account.setBalance(account.getBalance()+depositAmount);
    System.out.println("Transaction Details");
    System.out.println("-----");
    System.out.println("Transaction Id      : "+transactionId);
    System.out.println("Account Number      : 
"+account.getAccNo());
    System.out.println("Account Holder Name : 
"+account.getAccHolderName());
    System.out.println("Account Type        : 
"+account.getAccType());
    System.out.println("Transaction Name     : DEPOSIT");
    System.out.println("Deposit Amount       : "+depositAmount);
    System.out.println("Total Balance        : 
"+account.getBalance());
    System.out.println("Transaction Status   : SUCCESS");
    System.out.println("*****ThanQ, Visit 
Again*****");
}
}

```

Main.java

```

import com.durgasoft.entities.Account;
import com.durgasoft.entities.Transaction;

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

        Account account = new Account();
        account.setAccNo("abc123");
        account.setAccHolderName("Durga");
        account.setAccType("Savings");
        account.setBalance(25000);

        Transaction transaction = new 
Transaction("1234345a786def456");
        transaction.deposit(account, 10000);
    }
}

```

```
}  
}
```

Polymorphism:

Polymorphism is a GREEK word, where Poly means many morphisms means forms or Structures.

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

The main advantage of Polymorphism is “Flexibility”.

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 up to some new functionality is called Method Overloading.

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 with the different method signatures.

In the above context, differences in the method parameter list may be in either of the forms.

1. Difference in the number of parameters.

```
void add(int i, int j){}  
void add(int i, int j, int k){}  
void add(int i, int j, int k, int l){}
```

2. Difference in the parameter data types.

```
void add(int i, int j){}
void add(float f1, float f2){}
```

3. Difference in the order of the parameters:

```
void add(int i, float f){    }
void add(float f, int i){    }
```

In Java applications, in method overloading, it is possible to provide more than one method either in the same class or in the superclass and subclass, that is, it is possible to perform method overloading with or without the inheritance.

EX:

```
class Math{
    void add(int i, int j){
        System.out.println("Result : "+(i+j));
    }
    void add(float f1, float f2){
        System.out.println("Result : "+(f1+f2));
    }
    void add(String str1, String str2){
        System.out.println("Result : "+(str1+str2));
    }
}
public class Main {
    public static void main(String[] args) {

        Math math = new Math();
        math.add(10,20);
        math.add(22.22f, 33.33f);
        math.add("abc", "def");
    }
}
```

Result : 30

Result : 55.550003

Result : abcdef

EX:

```
class Employee{
    public void generateSalary(int basic, float hk, int ta, float pf){
        float sal = basic+(basic*hk/100)+ta-(basic*pf/100);
        System.out.println("Salary : "+sal);
    }
}
```



```

    }
    public void generateSalary(int basic, float hk, int ta, float pf,
int bonus){
        float sal = basic+(basic*hk/100)+ta-(basic*pf/100)+bonus;
        System.out.println("Salary      : "+sal);
    }
}
public class Main {
    public static void main(String[] args) {
        Employee employee = new Employee();
        employee.generateSalary(25000, 50.0f, 1000, 12.5f);
        employee.generateSalary(25000, 50.0f, 1000, 12.5f, 5000);

    }
}

```

```

Salary      : 35375.0
Salary      : 40375.0

```

In the above Method overloading, finding a method among the number of same methods on the basis of the parameter values is called Method Overloading Resolution or Method Resolution.

Method Overriding:

Method Overriding is the process of providing replacement for the existing method functionality with a new method functionality.

Note: Method overloading is the extension of the existing method functionality. Method Overriding is the replacement of the existing method functionality.

In Java applications, to perform method overriding we must have inheritance relations between the classes.

To perform method overriding we have to take more than one method with the same method prototypes.

To perform method overriding in Java applications we have to use the following steps.

1. Declare a superclass with a method which we want to override.
2. Declare a subclass for the above superclass with the same superclass method prototype and with the new functionality.

3. Declare the main class and main() method, access the superclass method and get output from the subclass method.

In method overriding, when we access a superclass method VM must execute the respective subclass method , to achieve this we have to use the following cases.

```
class A{
    void m1(){
        -----
    }
}
class B extends A{
    void m1(){
        -----
    }
}
```

Case#1:

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

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

Status: No Overriding here, method overriding requires Subclass object, not superclass object.

Case#2:

```
B b = new B();
```

```
b.m1();
```

Status: Method overriding happened here, but to prove method overriding we have to access superclass method , not subclass method.

Case#3:

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

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

Status: Method Overriding is Successful and proof is also be successful.

EX:

```
class A{
    void m1() {
        System.out.println("m1(): Old Method Implementation");
    }
}
class B extends A{
    void m1() {
```

```

        System.out.println("m1(): New Method Implementation");
    }
}
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();

    }
}

```

m1(): New Method Implementation

Q)What are the differences between Method Overloading and Method Overriding?

Ans:

1. Method Overloading is an extension process of an existing method functionality to a new method functionality.

Method Overriding is a replacement process of an existing method functionality to a new functionality.

2. To perform Method overloading inheritance is not mandatory.

To perform Method overriding Inheritance is mandatory.

3. In method overloading both the method signatures must be different.

In Method Overriding both the method prototypes must be the same.

Rules and regulations to perform method Overriding:

1. In method Overriding, the superclass method must not be declared as private, because we must access the superclass method outside of the superclass that is in the main class and in the main() method.

EX:

```
class A{
```

```

    private void m1(){
        System.out.println("m1(): Old Functionality");
    }
}
class B extends A{
    void m1(){
        System.out.println("m1(): New Functionality");
    }
}
class Test{
    public static void main(String[] args) {
        A a = new B();
        a.m1();
    }
}

```

nagoorn@Nagoors-MacBook-Pro apps % javac Test.java

Test.java:14: error: m1() has private access in A

 a.m1();

 ^

2. In Method Overriding, the subclass method return type must be the same as the respective superclass method return type.

EX:

```

class A{
    int m1(){
        System.out.println("m1(): Old Functionality");
        return 10;
    }
}
class B extends A{
    void m1(){
        System.out.println("m1(): New Functionality");
    }
}
class Test{

```

```

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

```

nagoorn@Nagoors-MacBook-Pro apps % javac Test.java
Test.java:8: error: m1() in B cannot override m1() in A
 void m1(){
 ^
 return type void is not compatible with int
1 error

EX:

```

class A{
    int m1(){
        System.out.println("m1(): Old Functionality");
        return 10;
    }
}

class B extends A{
    int m1(){
        System.out.println("m1(): New Functionality");
        return 20;
    }
}

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

```

nagoorn@Nagoors-MacBook-Pro apps % javac Test.java
nagoorn@Nagoors-MacBook-Pro apps % java Test
m1(): New Functionality

3. In Method Overriding, the superclass method must not be declared as final irrespective of the subclass method final declaration.

EX:

```
class A{
    final void m1(){
        System.out.println("m1(): Old Functionality");
    }
}
class B extends A{
    void m1(){
        System.out.println("m1(): New Functionality");
    }
}
class Test{
    public static void main(String[] args) {
        A a = new B();
        a.m1();
    }
}
```

```
nagoorn@Nagoors-MacBook-Pro apps % javac Test.java
Test.java:7: error: m1() in B cannot override m1() in A
    void m1(){
        ^
    overridden method is final
```

EX:

```
class A{
    void m1(){
        System.out.println("m1(): Old Functionality");
    }
}
class B extends A{
    final void m1(){
        System.out.println("m1(): New Functionality");
    }
}
```

```

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

```

nagoorn@Nagoors-MacBook-Pro apps % javac Test.java

nagoorn@Nagoors-MacBook-Pro apps % java Test

m1(): New Functionality

4. In method Overriding, either superclass method or subclass method or both superclass method and subclass method must not be declared with static, if we declare either superclass method or subclass method 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, in this context if we create subclass object then the superclass method overhides subclass method, here if we access superclass method then JVM will execute only superclass method, this feature of method overriding is called Method Over hiding.

EX:

```

class A{
    static void m1(){
        System.out.println("m1(): Old Functionality");
    }
}
class B extends A{
    void m1(){
        System.out.println("m1(): New Functionality");
    }
}
class Test{
    public static void main(String[] args) {
        A a = new B();
        a.m1();
    }
}

```

```
nagoorn@Nagoors-MacBook-Pro apps % javac Test.java
Test.java:7: error: m1() in B cannot override m1() in A
```

```
    void m1(){
```

```
        ^
```

```
    overridden method is static
```

EX:

```
class A{
    void m1(){
        System.out.println("m1(): Old Functionality");
    }
}
class B extends A{
    static void m1(){
        System.out.println("m1(): New Functionality");
    }
}
class Test{
    public static void main(String[] args) {
        A a = new B();
        a.m1();
    }
}
```

```
nagoorn@Nagoors-MacBook-Pro apps % javac Test.java
Test.java:7: error: m1() in B cannot override m1() in A
```

```
    static void m1(){
```

```
        ^
```

```
    overriding method is static
```

EX:

```
class A{
    static void m1(){
        System.out.println("m1(): Old Functionality");
    }
}
class B extends A{
```



```

    static void m1(){
        System.out.println("m1(): New Functionality");
    }
}
class Test{
    public static void main(String[] args) {
        A a = new B();
        a.m1();
    }
}

```

nagoorn@Nagoors-MacBook-Pro apps % javac Test.java

nagoorn@Nagoors-MacBook-Pro apps % java Test

m1(): Old Functionality

5. In method overriding, subclass method scope must be either the same as the superclass method scope or wider than the superclass method scope.

EX:

```

class A{
    public void m1(){
        System.out.println("m1(): Old Functionality");
    }
}
class B extends A{
    protected void m1(){
        System.out.println("m1(): New Functionality");
    }
}
class Test{
    public static void main(String[] args) {
        A a = new B();
        a.m1();
    }
}

```

```
nagoorn@Nagoors-MacBook-Pro apps % javac Test.java
Test.java:7: error: m1() in B cannot override m1() in A
    protected void m1(){
                  ^
    attempting to assign weaker access privileges; was public
1 error
```

EX:

```
class A{
    protected void m1(){
        System.out.println("m1(): Old Functionality");
    }
}
class B extends A{
    public void m1(){
        System.out.println("m1(): New Functionality");
    }
}
class Test{
    public static void main(String[] args) {
        A a = new B();
        a.m1();
    }
}
```

```
nagoorn@Nagoors-MacBook-Pro apps % javac Test.java
nagoorn@Nagoors-MacBook-Pro apps % java Test
m1(): New Functionality
```

6. In Method Overriding, subclass method access privileges must be either the same as the superclass method access privileges or weaker access privileges than the superclass method access privileges.

Consider the following program.

```

class A{
    void m1(){
        System.out.println("m1(): Old Functionality");
    }
}
class B extends A{
    void m1(){
        System.out.println("m1(): New Functionality");
    }
}
public class Main {
    public static void main(String[] args) {
        A a = new B();
        a.m1();
    }
}

```

In the above program, when we access the superclass method automatically JVM recognizes the method overriding and JVM will execute the respective subclass method, where JVM will not execute superclass method body, so in this context providing implementation for the superclass method is unnecessary.

To overcome the above problem we have to declare a method in the superclass without providing implementation, in JAVA / J2EE applications if we want to declare a method without its implementation then we have to declare that method as an abstract method.

In Java applications, to declare a method as an abstract method then the respective class must be the abstract class.

EX:

```

abstract class A{
    abstract void m1();
}
class B extends A{
    void m1(){
        System.out.println("m1(): New Functionality");
    }
}
public class Main {
    public static void main(String[] args) {
        A a = new B();
        a.m1();
    }
}

```

```
    }  
}
```

m1(): New Functionality

Q)What are the differences between concrete methods and abstract methods?

Ans:

1. Concrete methods will have both method declaration and method implementation.

Abstract methods will have only method declaration without the method implementation.

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

Abstract methods are possible in both abstract classes and interfaces.

3. To declare concrete methods no need to use any special keyword.

To declare abstract methods we need to use the 'abstract' keyword.

4. Concrete methods will provide less shareability.

Abstract methods will provide more shareability.

Q)What are the differences between Concrete class and abstract class?

Ans:

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

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

2. To declare a concrete class we have to use the 'class' keyword.

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

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

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

4. Concrete classes are able to provide less shareability.

Abstract classes are able to provide more shareability.

In Java applications, when we declare an abstract class with abstract methods then it is a convention to declare a subclass for the abstract class and to provide implementation for all the abstract methods inside the subclass.

EX:

```
abstract class A{
    void m1() {
        System.out.println("m1-A");
    }
    abstract void m2();
    abstract void m3();
}
class B extends A{
    void m2() {
        System.out.println("m2-B");
    }
    void m3() {
        System.out.println("m3-B");
    }
    void m4() {
        System.out.println("m4-B");
    }
}
public class Main {
    public static void main(String[] args) {
        A a = new B();
        a.m1();
        a.m2();
    }
}
```

```

        a.m3();
        //a.m4(); ----> Error

        B b = new B();
        b.m1();
        b.m2();
        b.m3();
        b.m4();
    }
}

```

```

m1-A
m2-B
m3-B
m1-A
m2-B
m3-B
m4-B

```

In Java applications, to declare a method as an abstract method then the respective class must be an abstract class, but to declare a class as an abstract class then it is not mandatory to declare at least one abstract method. In Java applications, we can declare abstract classes with or without the abstract methods.

EX:

```

abstract class A{
    void m1() {
        System.out.println("m1-A");
    }
    void m2() {
        System.out.println("m2-A");
    }
}
class B extends A{

}
public class Main {
    public static void main(String[] args) {
        A a = new B();
        a.m1();
        a.m2();
    }
}

```

m1-A

m2-A

In Java applications, we can extend an abstract class to a concrete class, we can extend a concrete class to an abstract class.

EX:

```
class A{
    void m1(){
        System.out.println("m1()-A");
    }
}
abstract class B extends A{
    abstract void m2();
    abstract void m3();
}
class C extends B{
    void m2(){
        System.out.println("m2()-C");
    }
    void m3(){
        System.out.println("m3()-C");
    }
}
class Test{
    public static void main(String[] args) {
        A a = new C();
        a.m1();

        B b = new C();
        b.m1();
        b.m2();
        b.m3();
    }
}
```

```
C c = new C();  
c.m1();  
c.m2();  
c.m3();  
}  
}
```

nagoorn@Nagoors-MacBook-Pro apps % javac Test.java

nagoorn@Nagoors-MacBook-Pro apps % java Test

m1()-A

m1()-A

m2()-C

m3()-C

m1()-A

m2()-C

m3()-C

nagoorn@Nagoors-MacBook-Pro apps %

In Java applications, if we declare an abstract class with some abstract methods and if we take a subclass for the abstract class then it is mandatory to provide implementation for all the abstract methods in the subclass, in this case if we provide implementation for some of the abstract methods, not for all the abstract methods in the subclass then the compiler will raise an error, in this context, to remove compilation errors we have to declare the respective subclass as an abstract class and take a subclass for the abstract class and provide implementation for all the remaining abstract methods.

EX:

```
abstract class A{  
    abstract void m1();  
    abstract void m2();  
    abstract void m3();  
}  
abstract class B extends A{
```



```

    void m1() {
        System.out.println("m1-B");
    }
}
class C extends B{
    void m2() {
        System.out.println("m2-C");
    }
    void m3() {
        System.out.println("m3-C");
    }
}
class Test{
    public static void main(String[] args) {
        A a = new C();
        a.m1();
        a.m2();
        a.m3();

    }
}

```

nagoorn@Nagoors-MacBook-Pro apps % javac Test.java

nagoorn@Nagoors-MacBook-Pro apps % java Test

m1-B

m2-C

m3-C

Interfaces:

Interfaces is a Java feature, it is able to allow only abstract methods.

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

In Java applications, for interfaces we are able to declare the reference variables, but we are unable to create the objects.

Inside the interfaces, by default, all variables are “public static final”, no need to declare explicitly.

Inside the interfaces, by default, all methods are “public abstract”, no need to declare explicitly.

When compared with the concrete classes and abstract classes , interfaces are able to provide more shareability.

In Java applications, if we declare an interface with abstract methods then it is a convention to provide an implementation class and provide implementation for all the abstract methods.

EX:

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

        A a = new A();
        a.m1();
        a.m2();
        a.m3();
    }
}
```

```

        a.m4 ();

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

```

```

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 some abstract methods and if we take an implementation class for the interface then it is mandatory to provide implementation for all the methods of the interface in the implementation class, if we miss implementation for any abstract method then the compiler will raise an error, in this context to come out from the compilation errors we have to declare the respective implementation class as an abstract class and we have to provide implementation to all the remaining abstract methods by taking a subclass for the abstract class.

EX:

```

interface I{
    void m1();
    void m2();
    void m3();
}

abstract class A implements I{
    public void m1(){
        System.out.println("m1-A");
    }
}

class B extends A{
    public void m2(){
        System.out.println("m2-B");
    }
}

```

```

        public void m3(){
            System.out.println("m3-B");
        }
    }
    public class Main {
        public static void main(String[] args) {
            I i = new B();
            i.m1();
            i.m2();
            i.m3();
        }
    }
}
m1-A
m2-B
m3-B

```

In Java , it is possible to implement more than one interface in a single implementation class.

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();
    }
}

```

m1-A

m2-A

m3-A

In Java applications, it is not possible to extend more than one class to a single class , 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 Java applications, we will use interfaces to declare services [abstract methods] and we will give an option to some other developers in the same project or in different modules or to the third party users to provide implementations for all the services.

EX:

In JDBC, Driver is an interface provided by SUN Microsystems and it has given an option to all the database vendors to provide their own implementations for the Driver interface in order to connect with their own databases.

As per the above notation, almost all the database vendors have provided their own implementation classes for the Driver interface.

EX:

```

interface Driver{// SUN Microsystems
    void registerDriver();
    void connect();
}

class OracleDriver implements Driver{// Oracle Vendor
    public void registerDriver(){
        System.out.println("OracleDriver Registered.....");
    }
    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.....");
    }
    public void connect(){
        System.out.println("Connection Established between Java
application and the MySQL Database....");
    }
}
class DB2Driver implements Driver{// DB2 Vendor
    public void registerDriver(){
        System.out.println("DB2Driver Registered.....");
    }
    public void connect(){
        System.out.println("Connection Established between Java
application and the DB2 Database....");
    }
}
class Test{// JDBC Application
    public static void main(String[] args) {
        // If we use Oracle DB
        Driver oracleDriver = new OracleDriver();
        oracleDriver.registerDriver();
        oracleDriver.connect();
        System.out.println();

        // If we use MySQL DB
        Driver mysqlDriver = new MySQLDriver();
        mysqlDriver.registerDriver();
        mysqlDriver.connect();
        System.out.println();

        // If we use DB2 DB
        Driver db2Driver = new DB2Driver();
        db2Driver.registerDriver();
        db2Driver.connect();
    }
}

```

```
}  
}
```

```
nagoorn@Nagoors-MacBook-Pro apps % javac Test.java  
nagoorn@Nagoors-MacBook-Pro apps % java Test  
OracleDriver Registered.....  
Connection Established between Java application and the Oracle Database.....
```

```
MySQLDriver Registered.....  
Connection Established between Java application and the MySQL Database....
```

```
DB2Driver Registered.....  
Connection Established between Java application and the DB2 Database....  
nagoorn@Nagoors-MacBook-Pro apps %
```

Q) Find the valid syntaxes between classes, abstract classes and interfaces from the following list?

-
1. class extends class -----> Valid
 2. class extends class,class -----> Invalid
 3. Class extends abstract class -----> Valid
 4. class extends abstract class, abstract class -----> Invalid
 5. class extends class, abstract class -----> Invalid
 6. class extends interface -----> Invalid
 7. class implements class -----> Invalid
 8. class implements abstract class -----> Invalid
 9. Class implements interface -----> Valid
 - 10.class implements interface, interface -----> Valid
 - 11.class implements interface extends class -----> Invalid
 - 12.class extends class implements interface -----> Valid
 - 13.class extends class implements interface, interface -----> Valid
 - 14.class extends abstract class implements interface -----> Valid
 - 15.class extends abstract class implements interface, interface -----> Valid

 - 16.abstract class extends class -----> Valid
 - 17.abstract class extends class,class -----> Invalid
 - 18.abstract Class extends abstract class -----> Valid
 - 19.abstract class extends abstract class, abstract class -----> Invalid
 - 20.abstract class extends class, abstract class -----> Invalid
 - 21.abstract class extends interface -----> Invalid

22.abstract class implements class -----> Invalid
 23.abstract class implements abstract class -----> Invalid
 24.abstract Class implements interface -----> Valid
 25.abstract class implements interface, interface -----> Valid
 26.abstract class implements interface extends class -----> Invalid
 27.abstract class extends class implements interface -----> Valid
 28.abstract class extends class implements interface, interface -----> Valid
 29.abstract class extends abstract class implements interface -----> Valid
 30.abstract class extends abstract class implements interface, interface
 ----> Valid

 31.interface extends class -----> Invalid
 32.interface extends abstract class -----> Invalid
 33.interface extends interface -----> Valid
 34.interface extends interface, interface -----> Valid
 35.interface implements class -----> Invalid
 36.interface implements abstract class -----> Invalid
 37.interface implements interface -----> Invalid
 38.interface implements interface, interface -----> Invalid

Marker Interfaces:

If any interface is declared without the abstract methods then that interface is called a Marker interface.

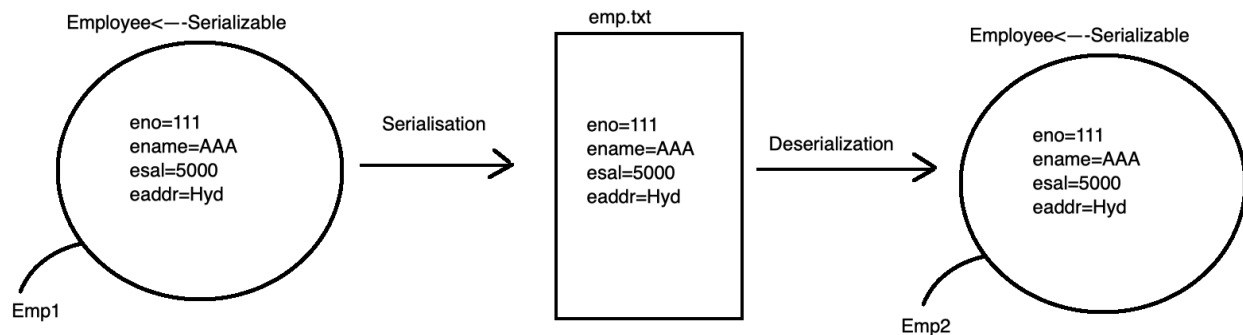
In Java applications, Marker interfaces are not for implementing business logic, where the marker interfaces are able to provide some abilities to the objects at runtime of the applications.

EX: java.io.Serializable
 java.lang.Cloneable

java.io.Serializable:

The process of separating data from an object is called Serialization.
 The process of reconstructing an object on the basis of the data is called Deserialization.

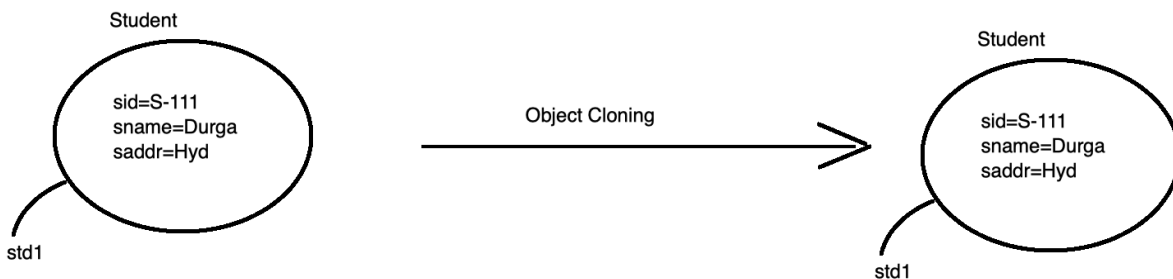
In Java applications, by default all java objects are not eligible for the Serialization and Deserialization, only the objects whose classes are implementing java.io.Serializable marker interface are eligible for the Serialization and Deserialization.



`java.lang.Cloneable`:

The process of generating a duplicate object on the basis of an 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 the Object cloning.



Adapter Classes:

In general, in Java applications, we will declare interfaces with the abstract methods as per the requirement, as per the java conventions we have to provide implementation for all the interface methods by taking an implementation class. In this context, once we implement an interface in an implementation class then it is mandatory to provide implementation for all the abstract methods with or without the application requirement.

The above approach may increase the number of unnecessary methods in the java applications. To overcome this problem we have to use the Adapter Design pattern.

In Java applications, Adapter Design Pattern has provided the following solution for the above design problem.

1. Declare a mediator class between interface and the implementation classes.
2. Implement the interface and provide empty implementation to all the methods.
3. Each and every implementation class must extend the Mediator class in place of implementing the interface.
4. In the implementation classes, override the required methods and remove the unnecessary methods.
5. In this solution , developers must create objects for the implementation classes , not for the mediator class, so we must declare the mediator class as an abstract class.
6. In this solution, if we need any method common in all the implementation classes with the variable implementation then we have to declare that method as an abstract method in the mediator class.

In the above solution, the mediator class is called a Generic class or an Adapter class.

```
Interface I{
    void m1();
    -----
    void m50();
}
```

```
Abstract class M implements I{
    p v m1() { }
    -----
    abstract p v m25() ;
    -----
    p v m50() { }
}
```

Generic Class
Adapter class

```
extends M
class C1 implements I{
    p v m1() { ---- }
    p v m25() { ---- }
}
```

```
extends M
class C2 implements I{
    p v m2() { ---- }
    p v m25() { ---- }
}
```

— — — — —

```
extends M
class C10 implements I{
    p v m10() { ---- }
    p v m25() { ---- }
}
```

Note: The main intention of the Adapter classes is to avoid the unnecessary methods implementation while implementing interfaces in the implementation classes.

EX: WindowAdapter.
 MouseListener
 GenericServlet

EX:

```
public interface WindowListener{
    public void windowOpened(WindowEvent we);
    public void windowClosed(WindowEvent we);
    public void windowClosing(WindowEvent we);
    public void windowIconified(WindowEvent we);
    public void windowDeiconified(WindowEvent we);
    public void windowActivated(WindowEvent we);
    public void windowDeactivated(WindowEvent we);
}

public class WindowAdapter implements WindowListener{
    public void windowOpened(WindowEvent we){    }
    public void windowClosed(WindowEvent we){    }
    public void windowClosing(WindowEvent we){    }
    public void windowIconified(WindowEvent we){    }
    public void windowDeiconified(WindowEvent we){    }
    public void windowActivated(WindowEvent we){    }
    public void windowDeactivated(WindowEvent we){    }
}

class WindowHandler extends WindowAdapter{
    public void windowClosing(WindowEvent we){ ----- }
}
```

Object Cloning:

The process of generating a duplicate object from an object is called the Object cloning.

In Java applications, if we want to perform Object Cloning, we have to use the following steps.

1. Declare a class and implement java.lang.Cloneable marker interface.
In java applications, by default all java objects are not eligible for the Object cloning, only the objects whose classes are implementing java.lang.Cloneable marker interface are eligible for the Object cloning.
2. Provide the variables and methods in the user defined class as per the requirement.
3. Override the Object class provided clone method in the user defined class and access the Object class provided clone method by using super keyword.
4. In the Main class and main() method , create an object for the User defined class and access the user defined class provided clone() method in order to get the duplicate object.

EX:

```
package com.durgasoft.test;
class Student implements Cloneable{

    private String sid;
    private String sname;
    private String saddr;

    public Student(String sid, String sname, String saddr) {
        super();
        this.sid = sid;
        this.sname = sname;
        this.saddr = saddr;
    }

    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);
    }

    @Override
    public Object clone() throws CloneNotSupportedException {
        Object obj = super.clone();
        return obj;
    }
}
```

```

    }

}

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

        Student originalStudent = new Student("S-111", "Durga",
"Hyderabad");
        System.out.println("Student Details from Original Student
Object");
        originalStudent.getStudentDetails();
        System.out.println("Original Student Object Ref    :
"+originalStudent);
        System.out.println();

        Student duplicateStudent = (Student)originalStudent.clone();

        System.out.println("Student Details from the Duplicate Student
Object");
        duplicateStudent.getStudentDetails();
        System.out.println("Duplicate Student Object Ref    :
"+duplicateStudent);
    }
}

```

```

Student Details from Original Student Object
Student Details
-----
Student Id      : S-111
Student Name    : Durga
Student Address : Hyderabad
Original Student Object Ref    : com.durgasoft.test.Student@7344699f
Student Details from the Duplicate Student Object
Student Details
-----
Student Id      : S-111
Student Name    : Durga
Student Address : Hyderabad
Duplicate Student Object Ref    : com.durgasoft.test.Student@6b95977

```

In Java , there are two types of Object Clonings.

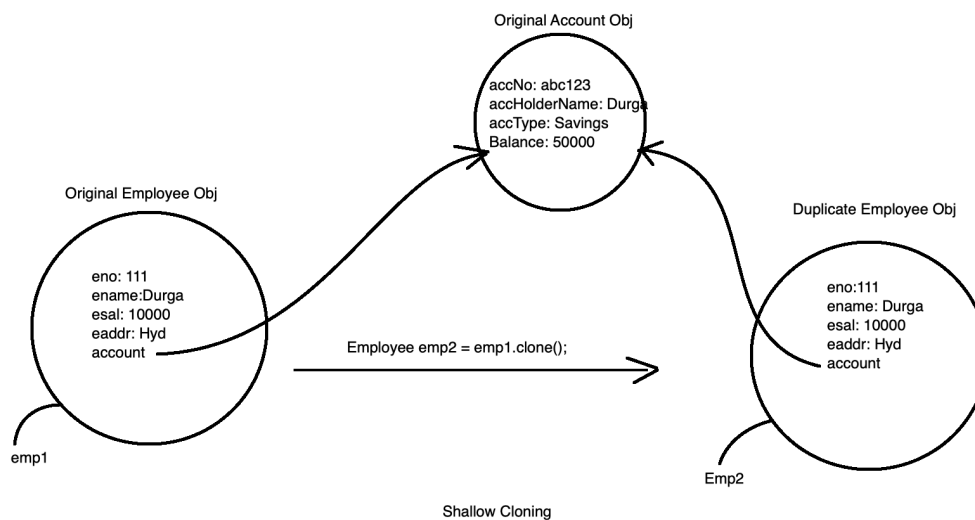
1. Shallow Cloning

2. Deep Cloning

Shallow Cloning:

It is a default cloning mechanism, where the developers are not required to define the cloning mechanism, it was provided in the Object class provided clone() method.

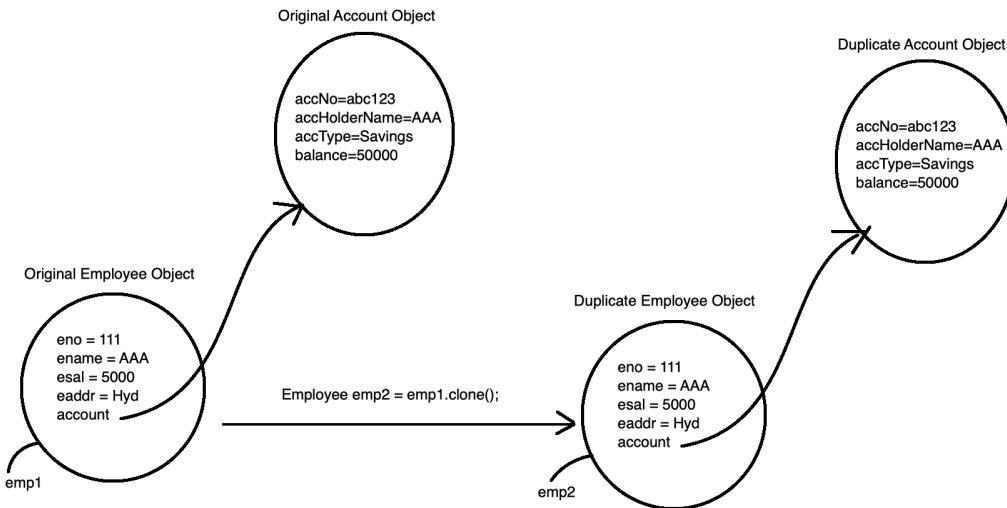
In the case of Shallow Cloning, when we perform cloning over a container object, JVM will perform cloning over the Container object only, JVM will not perform cloning over the Contained object. In the Shallow Cloning, both Original Container and Duplicate Container objects are referring to the same Contained Object.



Deep Cloning:

It is not a default cloning mechanism, it is an user defined cloning mechanism, where the developers must define the cloning logic explicitly.

In the case of Deep cloning, When we perform cloning over the Container object , JVM will perform Cloning over the container object and its contained object, in this case, Original Container object will refer the original contained object and the Duplicate container object will refer the duplicate contained object.



EX:

```

package com.durgasoft.test;
class Account {
    private String accNo;
    private String accHolderName;
    private String accType;
    private int balance;
    public Account(String accNo, String accHolderName, String accType, int
balance) {
        this.accNo = accNo;
        this.accHolderName = accHolderName;
        this.accType = accType;
        this.balance = 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    : " + balance);
    }
}
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) {
        super();
        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.getAccountDetails();
    }
    @Override
    public Object clone() throws CloneNotSupportedException {
        Object obj = super.clone();
        return obj;
    }
    public Account getAccount() {
        return account;
    }
}

public class Test {
    public static void main(String[] args) throws
CloneNotSupportedException {
        Account originalAccount = new Account("abc123", "Durga",
"Savings", 5000);
        Employee originalEmployee = new Employee(111, "Durga", 5000,
"Hyd", originalAccount);
        System.out.println("Employee Details from the Original Object");
        originalEmployee.getEmployeeDetails();
        System.out.println("Original Employee Object Ref :
"+originalEmployee);
    }
}

```

```

        System.out.println("Original Account Object Ref    :
"+originalAccount);
        System.out.println();

        Employee duplicateEmployee = (Employee) originalEmployee.clone();
        System.out.println("Employee Details from Duplicate Object");
        duplicateEmployee.getEmployeeDetails();
        System.out.println("Duplicate Employee Object Ref    :
"+duplicateEmployee);
        System.out.println("Duplicate Account Object Ref    :
"+duplicateEmployee.getAccount());
    }
}

```

Employee Details from the Original Object

Employee Details

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

Account Details

Account Number : abc123
Account Holder Name : Durga
Account Type : Savings
Account Balance : 5000

Original Employee Object Ref : com.durgasoft.test.Employee@70dea4e

Original Account Object Ref : com.durgasoft.test.Account@5c647e05

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 Holder Name : Durga
Account Type : Savings
Account Balance : 5000

Duplicate Employee Object Ref : com.durgasoft.test.Employee@33909752

Duplicate Account Object Ref : com.durgasoft.test.Account@5c647e05

EX:

```
package com.durgasoft.test;
class Account {
    private String accNo;
    private String accHolderName;
    private String accType;
    private int balance;
    public Account(String accNo, String accHolderName, String accType, int
balance) {
        this.accNo = accNo;
        this.accHolderName = accHolderName;
        this.accType = accType;
        this.balance = balance;
    }

    public String getAccNo() {
        return accNo;
    }
    public String getAccHolderName() {
        return accHolderName;
    }
    public String getAccType() {
        return accType;
    }
    public int getBalance() {
        return 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    : " + balance);
    }
}
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) {
        super();
        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.getAccountDetails();
    }
    @Override
    public Object clone() throws CloneNotSupportedException {
        Account duplicateAccount = new Account(account.getAccNo(),
account.getAccHolderName(), account.getAccType(), account.getBalance());
        Employee duplicateEmployee = new Employee(enno, ename, esal, eaddr,
duplicateAccount);
        return duplicateEmployee;
    }
    public Account getAccount() {
        return account;
    }
}
public class Test {
    public static void main(String[] args) throws CloneNotSupportedException {
        Account originalAccount = new Account("abc123", "Durga", "Savings",
5000);
        Employee originalEmployee = new Employee(111, "Durga", 5000, "Hyd",
originalAccount);
        System.out.println("Employee Details from the Original Object");
        originalEmployee.getEmployeeDetails();
        System.out.println("Original Employee Object Ref : "+originalEmployee);
        System.out.println("Original Account Object Ref  : "+originalAccount);
        System.out.println();

        Employee duplicateEmployee = (Employee) originalEmployee.clone();
        System.out.println("Employee Details from Duplicate Object");
        duplicateEmployee.getEmployeeDetails();
        System.out.println("Duplicate Employee Object Ref :
"+duplicateEmployee);
        System.out.println("Duplicate Account Object Ref  :
"+duplicateEmployee.getAccount());
    }
}

```

Employee Details from the Original Object

Employee Details

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

Account Details

```
-----  
Account Number       : abc123  
Account Holder Name  : Durga  
Account Type         : Savings  
Account Balance      : 5000  
Original Employee Object Ref : com.durgasoft.test.Employee@70dea4e  
Original Account Object Ref  : com.durgasoft.test.Account@5c647e05  
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 Holder Name  : Durga  
Account Type         : Savings  
Account Balance      : 5000  
Duplicate Employee Object Ref : com.durgasoft.test.Employee@33909752  
Duplicate Account Object Ref  : com.durgasoft.test.Account@55f96302
```

'instanceof' operator:

It is a boolean operator, it will check whether a reference variable is representing an instance of the provided class or not.

Syntax:

refVar instanceof ClassName

1. If the refVar class type is same as the provided ClassName then the instanceof operator will return true value.
2. If the refVar class type is subclass to the provided class then the instanceof operator will return true value.
3. If the refVar class type is superclass to the provided class then the instanceof operator is false value.
4. If no relationship exists between the refVar class and the provided className then the compiler will raise an error.

EX:

```
package com.durgasoft.test;
class A{
}
class B extends A{

}
class C{

}

public class Test {
    public static void main(String[] args){
        A a = new A();
        System.out.println(a instanceof A);
        System.out.println(a instanceof Object);

        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 B); ----> Error

    }
}
```

true
true
true
true
false
true

