4 oops (object oriented programming) concepts in java > Encapsulation, Abstraction, Inheritance, Polymorphism in detail with programs, abstraction via interface and Abstract classes, Inheritance via extends and implements,Compile Runtime polymorphism

Oop stands for **object oriented programming.**

And is often called as Oops

Oop is often pronounced as Oops.

There are **4 basic oops concepts in java** >

- *1. Encapsulation*
- *2. Abstraction*
- *3. Inheritance*
- *4. Polymorphism*

Contents of page :

- ***1. Encapsulation***
  - ***To achieve encapsulation >***
  - ***Advantages of using encapsulation >***
  - ***Encapsulation makes our java code>***
    - maintainable,
    - extensible and
    - flexible.
  - ***Program 1.1 to demonstrate Encapsulation>***

- ***What would happen without encapsulation>***
  - ***Program 1.2 - What would happen without encapsulation >***
- ***What could be impact of not using encapsulation>***
  - ***Program 1.3 - impact of not using encapsulation***
- **program 1.4 -** Now, lets understand how Encapsulation allows to **modify implemented java code without breaking others code** who have implemented the code via **program.**

- ***2. Abstraction***
  - In java abstraction can be achieved by using >
    - Interfaces and
    - Abstract classes

- ***3. Inheritance***
  - **Program 3.1 to show how features of parent/super class are inherited in child/sub class >**
  - ***Inheritance can be achieved by using following keywords>***
    - **extends** and
    - **implements**
  - ***Class*** *can use* ***extend and implements >***
    - ***IS-A relationship with Classes>***
    - ***Program* 3.2** *to demonstrate* ***IS-A relationship with Classes***>
  - ***Interface*** *uses* ***extend*** *only* **>**
    - ***Program* 3.3** *to demonstrate IS-A relationship with* ***Classes and interfaces***>

- ***4. Polymorphism***
  - ***4.1) Compile time polymorphism***

- Compile time polymorphism can be achieved by using Method overloading

- ***4.2) Runtime polymorphism***
  - Runtime polymorphism can be achieved by using Method overriding

1. Encapsulation

In short, encapsulation means **data hiding.**

To achieve encapsulation >

- Make **fields/memberVariables private (*private*** can be accessed only within the same class, hence we are hiding the fields within the class**)**, and
- **access** those **private fields via public methods**. (***Public*** are accessible from everywhere)

Advantages of using encapsulation >

- Encapsulation **prevents other classes to access the class data** (i.e. preventing access to private fields).
- Encapsulation allows to **modify implemented java code without breaking others code** who have implemented the code.

- **Outside users** who are accessing this class **don't know about the private fields of class**,

Example **-** field may be Integer or String type, but user won't have any such information.

So, **class at any time can change data type of a field and users won't know about it**, even they need not to. (This point is related to above point)

- Class fields could be made **read-only** or **write-only**.

- ***Encapsulation makes our java code>***
    - maintainable,
    - extensible and
    - flexible.

Program 1.1 to demonstrate Encapsulation>

```
class Employee{
    private String id; //private field

    public String getId() { //private field accessed inside public method
           return id;
    }

    public void setId(String id) {
           this.id = id;
    }

}

/** JavaMadeSoEasy.com */
public class EncapsulationTest {
    public static void main(String[] args) {
           Employee emp=new Employee();
           emp.setId("1"); //public method can be accessed outside class.
           System.out.println("emp.getId()  >  "+emp.getId());
    }
}

/* OUTPUT

emp.getId()  >  1

*/
```

What would happen without encapsulation>

No encapsulation means fields won't be private and could be used outside class.

Program 1.2 - What would happen without encapsulation >

```
class Employee{
    String id;  //No encapsulation - field isn't private
}
```

```
/** JavaMadeSoEasy.com */
public class EncapsulationTest {
    public static void main(String[] args) {
           Employee emp=new Employee();
           emp.id="1";  //As field isn't private, it could be accessed outside class.

    }
}
```

What could be impact of not using encapsulation>

Let's say data type of id was changed from String to Integer, in that case compilation error will be generated wherever id has been used, because code was written considering id is String not a Integer.

So, by not using encapsulation we will end up **breaking others code**.

You must be thinking that in below program id has been accessed only at one place, we could make necessary adjustments, but what about id being used at thousands of places in other programs.

Program 1.3 - impact of not using encapsulation

```
4  class Employee{
5      Integer id; //No encapsulation - field isn't private
6  }
7
8  /** JavaMadeSoEasy.com */
9  public class EncapsulationTest {
10     public static void main(String[] args) {
11         Employee emp=new Employee();
12         emp.id="1"; //As field isn't private, it could be accessed outside class.
13
14     }
15 }
```

program 1.4 - Now, lets understand how Encapsulation allows to **modify implemented java code without breaking others code** who have implemented the code via **program**.

If id would have been private, other classes would have been accessing id outside class only via public methods of class. So, in that case if we were to change data type of id from String to Integer than to avoid breaking of others code we could make relevant changes in setter and getter methods.

In below program >

As compared to other programs data type of id has been changed from String to Integer,and to avoid breaking of others code we make relevant changes in setter and getter methods. (Please compare setter and getter methods with above program)

```
class Employee{
    Integer id;

    public String getId() {
           return String.valueOf(id);
    }

    public void setId(String id) {
           this.id = Integer.parseInt(id);
    }

}

/** JavaMadeSoEasy.com */
public class EncapsulationTest {
    public static void main(String[] args) {
           Employee emp=new Employee();
           emp.setId("1");
           System.out.println("emp.getId()  >  "+emp.getId());

    }
}

/* OUTPUT

emp.getId()  >  1

*/
```

2. Abstraction

In short, **Abstraction** means **hiding** the **implementation.**

Abstraction means representing only the essential things without including background details.

In java abstraction can be achieved by using >

- Interfaces and
- Abstract classes

Interface	Abstract class
Interface helps in achieving **pure abstraction** in java.	Abstract class aren't **purely abstraction** in java

All Interface are **abstract by default.**
So, it's **not mandatory** to write abstract keyword with interface.
 	
Example-
`interface MyInterface {//compiler will add abstract`
`}`
	It's **mandatory** to write abstract keyword to make class abstract.
	
	Example-
	`abstract class MyAbstractClass{`
	`abstract void m();`

Because of default additions done by compiler, **above code** will be **same as** writing **below code-**

```
abstract interface MyInterface {
}
```

}

Must read : 10 Differences between Interface and abstract class in java - in detail with programs

3. Inheritance

Inheritance is a process where child class acquires the properties of super class.

OR

Inheritance is a process where features of parent/super class are inherited in child/sub class.

Program 3.1 to show how features of parent/super class are inherited in child/sub class >

Animal is **superclass** of **Lion**.
Lion is subclass of **Animal**.

food() method of Animal class is inherited in Lion class.

```
class Animal {
    void food() {
           System.out.println("Animal eat food");
    }
}

class Lion extends Animal {

}


/** JavaMadeSoEasy.com */
public class MyClass{
    public static void main(String[] args) {
           Lion obj = new Lion();
           obj.food();
    }
}
```

```
/*OUTPUT

Animal eat food

*/
```

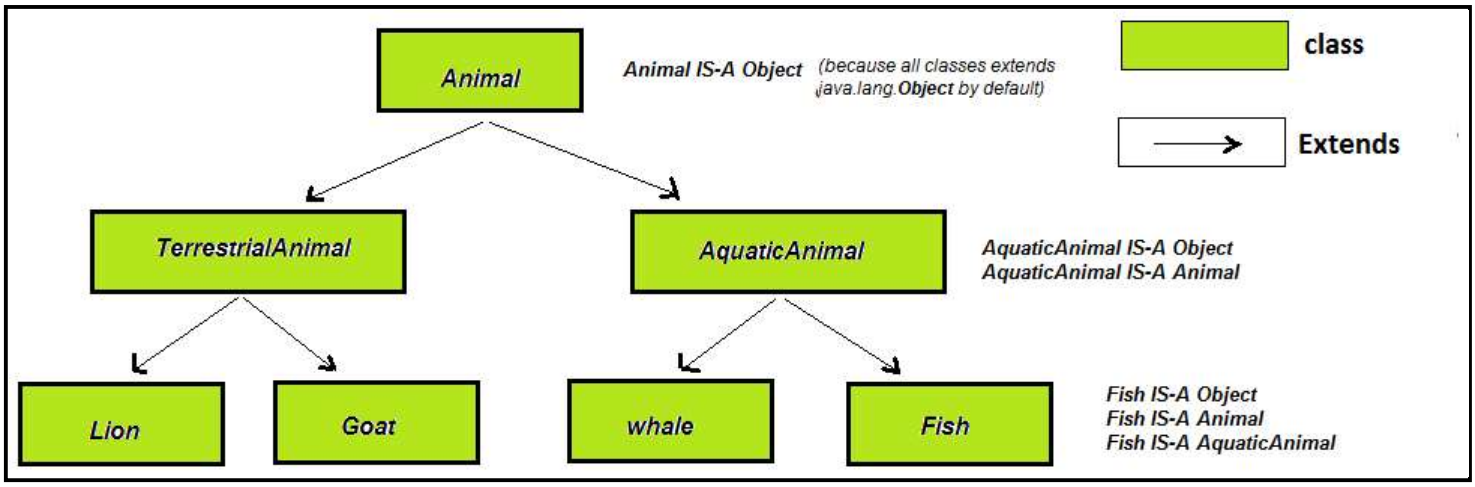
Inheritance can be achieved by using following keywords>

- **extends** and
- **implements**

Class can use extend and implements >

Class always **extends** another class.

Class always **implements** interface.



IS-A relationship with Classes>

Animal IS-A Object (because all classes extends java.lang.**Object** by default)

AquaticAnimal IS-A Object
AquaticAnimal IS-A Animal

Fish IS-A Object
Fish IS-A Animal
Fish IS-A AquaticAnimal

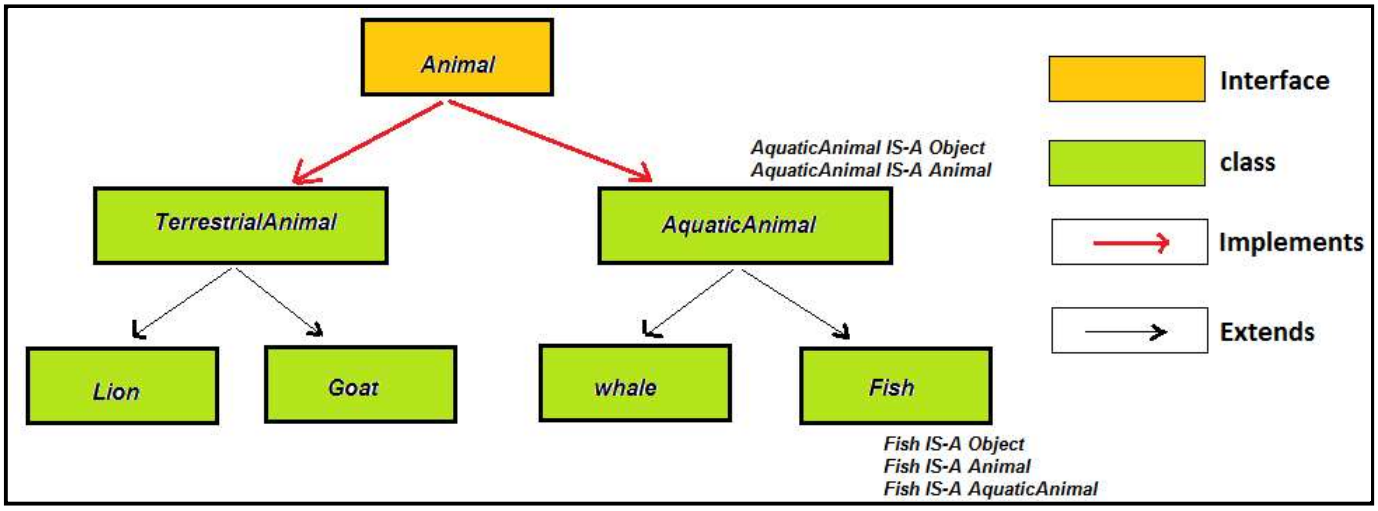
Program 3.2 to demonstrate IS-A relationship with Classes>

```
class Animal {}

class TerrestrialAnimal extends Animal{}
class AquaticAnimal extends Animal{}

class Lion extends TerrestrialAnimal {}
class Goat extends TerrestrialAnimal {}

class Fish extends AquaticAnimal {}
class Whale extends AquaticAnimal {}


/** JavaMadeSoEasy.com */
public class MyClass{
    public static void main(String[] args) {
            Object a = new Animal();  //Animal IS-A Object


            Object aa1 = new AquaticAnimal(); //AquaticAnimal IS-A Object
            Animal aa2 = new AquaticAnimal(); //AquaticAnimal IS-A Animal


            Object f1 = new Fish(); //Fish IS-A Object
            Animal f2 = new Fish(); //Fish IS-A Animal
            AquaticAnimal f3 = new Fish(); //Fish IS-A AquaticAnimal

    }
}
```

Interface *uses* ***extend*** *only* **>**

Interface always **extends another interface**.

Interface **can extend more than one interface**.



IS-A relationship with **Classes and interfaces**>

AquaticAnimal IS-A Object

AquaticAnimal IS-A Animal

Fish IS-A Object
Fish IS-A Animal
Fish IS-A AquaticAnimal

Program 3.3 to demonstrate IS-A relationship with Classes and interfaces>

```
interface Animal {}

class TerrestrialAnimal implements Animal{}
class AquaticAnimal implements Animal{}

class Lion extends TerrestrialAnimal {}
class Goat extends TerrestrialAnimal {}

class Fish extends AquaticAnimal {}
class Whale extends AquaticAnimal {}


/** JavaMadeSoEasy.com */
public class MyClass{
    public static void main(String[] args) {

            Object aa1 = new AquaticAnimal(); //AquaticAnimal IS-A Object
            Animal aa2 = new AquaticAnimal(); //AquaticAnimal IS-A Animal


            Object f1 = new Fish(); //Fish IS-A Object
            Animal f2 = new Fish(); //Fish IS-A Animal
            AquaticAnimal f3 = new Fish(); //Fish IS-A AquaticAnimal

    }
}
```

4. Polymorphism

Java allows >

- ***4.1) Compile time polymorphism &***
- ***4.2) Runtime polymorphism***

4.1) Compile time polymorphism

Compile time polymorphism can be achieved by using **Method overloading**
When a class have same method name with different argument, than it is called method overloading.

- Call to overloaded method is **bonded** at **compile time**.
- Method overloading concept is also known as **compile time polymorphism** in java.
- Java does **not allow overloading by changing the return type**, though overloaded methods can change the return type.
- Method overloading is **generally done in same class** but **can also be done in SubClass**
- Method name - Overriding method same name as of superclass method in java,

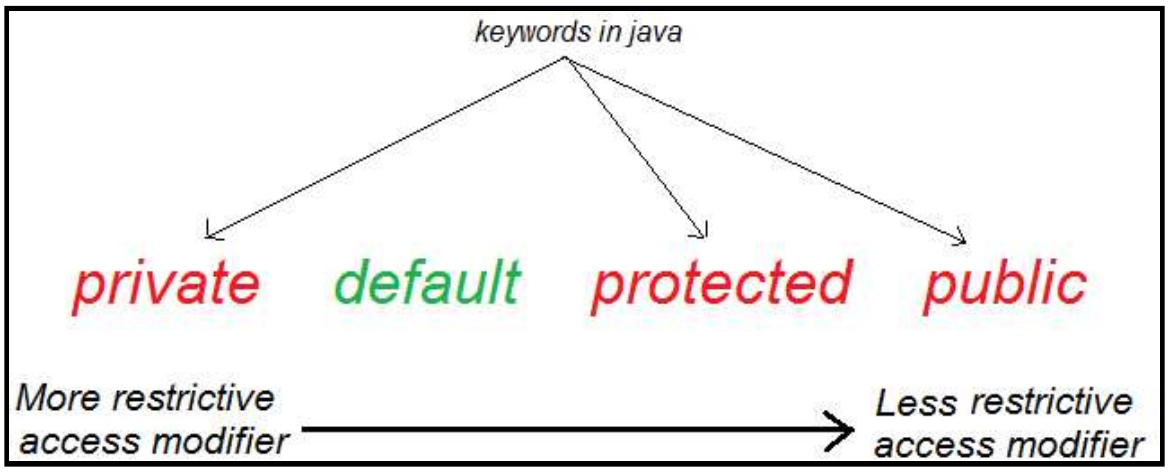
Advantage of method overloading -
Method overloading enables consistency in the naming of methods which logically perform almost similar tasks and the only difference is in number of arguments.
Method overloading enables same method name to be **reused** in program.
Method overloading is done to make program logically more readable and understandable.

4.2) Runtime polymorphism

Runtime polymorphism can be achieved by using **Method overriding**
Method of superclass is **overridden** in subclass **to provide more specific implementation**.

1. Access modifier - Overriding method must not have more restrictive access modifier in java.

- Example 1 - public method cannot be overridden by private method in java.
- Example 2 - default method can be overridden by default, protected or public method in java.



1. Return type - Java allow method overriding by changing the return type, but only Covariant return type are allowed. (see Program 4)

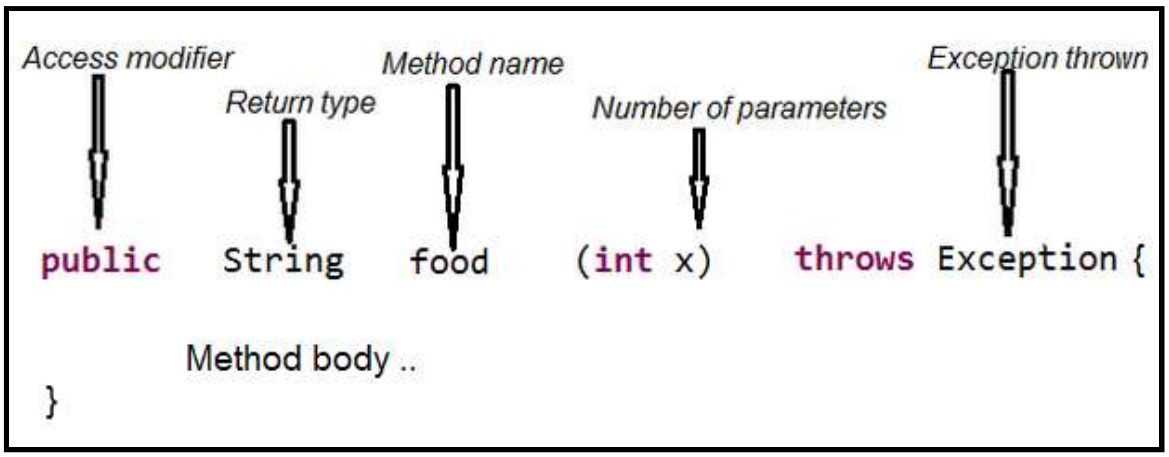
1. Number of parameters - Overriding method must have same number of parameters in java.

1. Exception thrown -
    - Overriding method must not throw new or broader checked exception in java,
    - though Overriding method may throw new narrower(subclass) of checked exception or
    - Overriding method can throw any runtime exception in java.

    For more detail on 5th point please Read : Throw/declare checked and unchecked exception while overriding superclass method in java

(Method definition formation) Let's make above 5 terms easy to remember by diagram / understand method overriding by diagram -

Method definition is formed by using following 5 terms -



Note : Return type may be void, in that case method doesn't return anything in java.

Must read for detailed illustration with programs >

[Method overloading in java - in detail with programs,10 Features,need of method overloading, overloading main method, Diagram and tabular form of Implicit casting/promotion of primitive Data type]

[Method overriding in java - in detail with programs, 10 Features, need of method overriding, understanding @Override annotation, Covariant return, diagram to understand access modifiers]