OOPs concepts in Java

Object-oriented programming System(OOPs) is a programming paradigm based on the concept of “objects” that contain data and methods.

The primary purpose of object-oriented programming is to increase the flexibility and maintainability of programs. Object oriented programming brings together data and its behaviour(methods) in a single location(object) makes it easier to understand how a program works.

We will cover each and every feature of OOPs in detail so that you won’t face any difficultly understanding **OOPs Concepts**.

OOPs Concepts – Table of Contents

What is an Object

Object: is a bundle of data and its behaviour(often known as methods).

Objects have two characteristics: They have **states** and **behaviors**.

Examples of states and behaviors
Example 1:
Object: House

State: Address, Color, Area
Behavior: Open door, close door

So if I had to write a class based on states and behaviours of House. I can do it like this: States can be represented as instance variables and behaviours as methods of the class. We will see how to create classes in the next section of this guide.

```
class House {
   String address;
   String color;
   double are;
   void openDoor() {
      //Write code here
   }
   void closeDoor() {
      //Write code here
   }
 ...
 ...
}
```

Example 2:
Let's take another example.
Object: Car
State: Color, Brand, Weight, Model
Behavior: Break, Accelerate, Slow Down, Gear change.

Note: As we have seen above, the states and behaviors of an object, can be represented by variables and methods in the class respectively.

Characteristics of Objects:

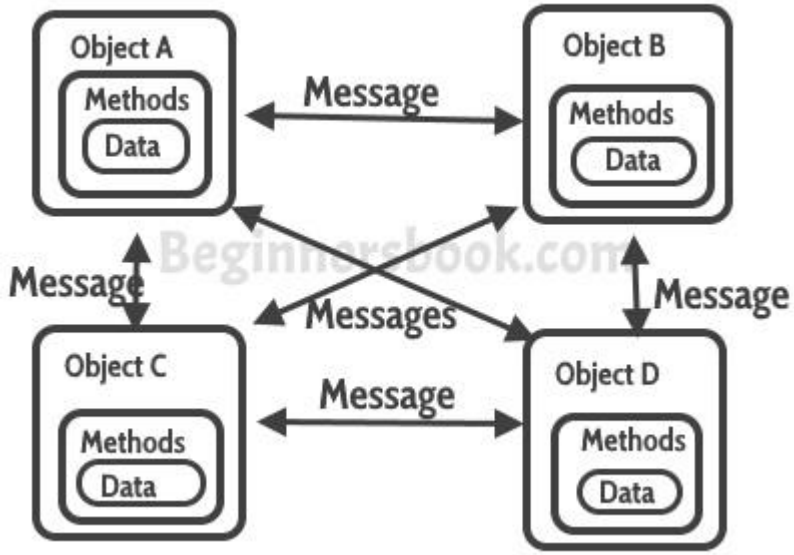
If you find it hard to understand Abstraction and Encapsulation, do not worry as I have covered these topics in detail with examples in the next section of this guide.

1. Abstraction
2. Encapsulation
3. Message passing

Abstraction: Abstraction is a process where you show only "relevant" data and "hide" unnecessary details of an object from the user.

Encapsulation: Encapsulation simply means binding object state(fields) and behaviour(methods) together. If you are creating class, you are doing encapsulation.

Message passing
A single object by itself may not be very useful. An application contains many objects. One object interacts with another object by invoking methods on that object. It is also referred to as **Method Invocation**. See the diagram below.



What is a Class in OOPs Concepts

A class can be considered as a blueprint using which you can create as many objects as you like. For example, here we have a class `Website` that has two data members (also known as fields, instance variables and object states). This is just a blueprint, it does not represent any website, however using this we can create Website objects (or instances) that represents the websites. We have created two objects, while creating objects we provided separate properties to the objects using constructor.

```
public class Website {
   //fields (or instance variable)
   String webName;
   int webAge;

   // constructor
   Website(String name, int age){
      this.webName = name;
```

```
        this.webAge = age;
    }
    public static void main(String args[]){
        //Creating objects
        Website obj1 = new Website("beginnersbook", 5);
        Website obj2 = new Website("google", 18);

      //Accessing object data through reference
      System.out.println(obj1.webName+" "+obj1.webAge);
      System.out.println(obj2.webName+" "+obj2.webAge);
    }
}
```

Output:

```
beginnersbook 5
google 18
```

What is a Constructor

Constructor looks like a method but it is in fact not a method. It's name is same as class name and it does not return any value. You must have seen this statement in almost all the programs I have shared above:

```
MyClass obj = new MyClass();
```

If you look at the right side of this statement, we are calling the default constructor of class `myClass` to create a new object (or instance).

We can also have parameters in the constructor, such constructors are known as parametrized constructors.

Example of constructor

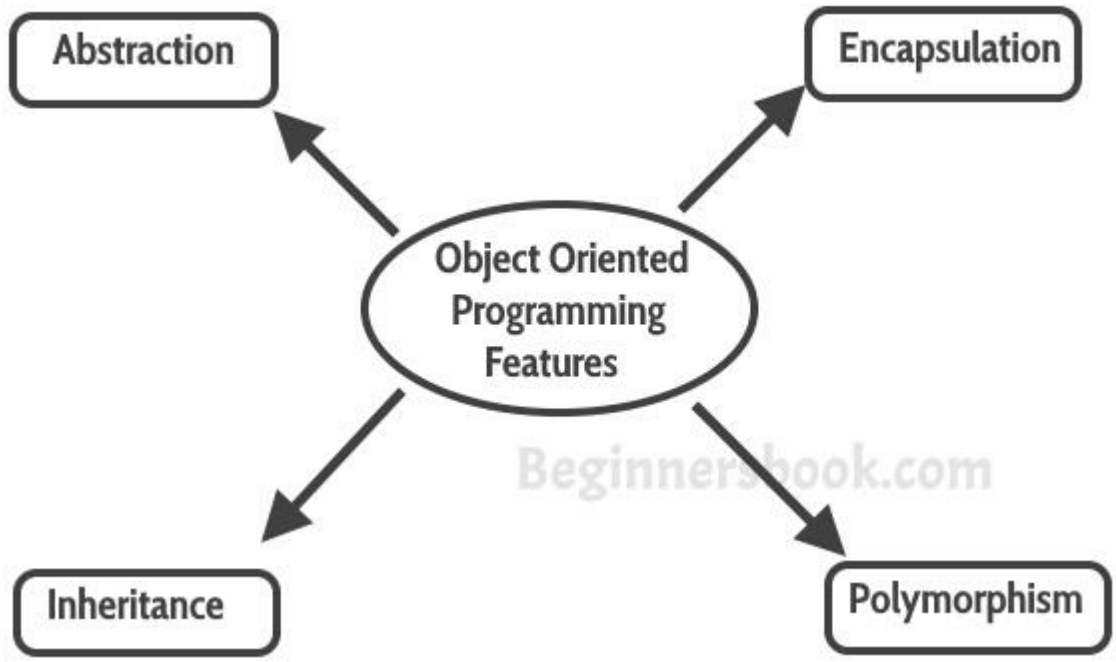
```
public class ConstructorExample {

    int age;
    String name;

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

    //Parameterized constructor
    ConstructorExample(String n,int a){
        this.name=n;
        this.age=a;
    }
```

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

Output:

```
Chaitanya 30
Steve 56
```

Object Oriented Programming features



These four features are the main OOPs Concepts that you must learn to understand the Object Oriented Programming in Java

Abstraction

Abstraction is a process where you show only “relevant” data and “hide” unnecessary details of an object from the user. For example, when you login to your bank account online, you enter your user_id and password and press login, what happens when you press login, how the input data sent to server, how it gets verified is all abstracted away from the you. Read more about it here: Abstraction in Java.

Encapsulation

Encapsulation simply means binding object state(fields) and behavior(methods) together. If you are creating class, you are doing encapsulation.

Encapsulation example in Java

How to

1) Make the instance variables private so that they cannot be accessed directly from outside the class. You can only set and get values of these variables through the methods of the class.

2) Have getter and setter methods in the class to set and get the values of the fields.

```
class EmployeeCount
{
   private int numOfEmployees = 0;
   public void setNoOfEmployees (int count)
   {
       numOfEmployees = count;
   }
   public double getNoOfEmployees ()
   {
       return numOfEmployees;
   }
}
public class EncapsulationExample
{
   public static void main(String args[])
   {
      EmployeeCount obj = new EmployeeCount ();
      obj.setNoOfEmployees(5613);
      System.out.println("No Of Employees:
"+(int)obj.getNoOfEmployees());
    }
}
```

Output:

```
No Of Employees: 5613
```

The class `EncapsulationExample` that is using the Object of class `EmployeeCount` will not able to get the NoOfEmployees directly. It has to use the setter and getter methods of the same class to set and get the value.

So what is the benefit of encapsulation in java programming

Well, at some point of time, if you want to change the implementation details of the class EmployeeCount, you can freely do so without affecting the classes that are using it.

Inheritance

The process by which one class acquires the properties and functionalities of another class is called inheritance. Inheritance provides the idea of reusability of code and each sub class defines only those features that are unique to it, rest of the features can be inherited from the parent class.

1. Inheritance is a process of defining a new class based on an existing class by extending its common data members and methods.
2. Inheritance allows us to reuse of code, it improves reusability in your java application.
3. The parent class is called the **base class** or **super class**. The child class that extends the base class is called the derived class or **sub class** or **child class**.

Note: The biggest advantage of Inheritance is that the code in base class need not be rewritten in the child class.
The **variables** and **methods** of the base class can be used in the **child class** as well.

Syntax: Inheritance in Java

To inherit a class we use extends keyword. Here class A is child class and class B is parent class.

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

Inheritance Example

In this example, we have a parent class `Teacher` and a child class `MathTeacher`. In the `MathTeacher` class we need not to write the same code which is already present in the present class. Here we have college name, designation and does() method that is common for all the teachers, thus MathTeacher class does not need to write this code, the common data members and methods can inherited from the `Teacher` class.

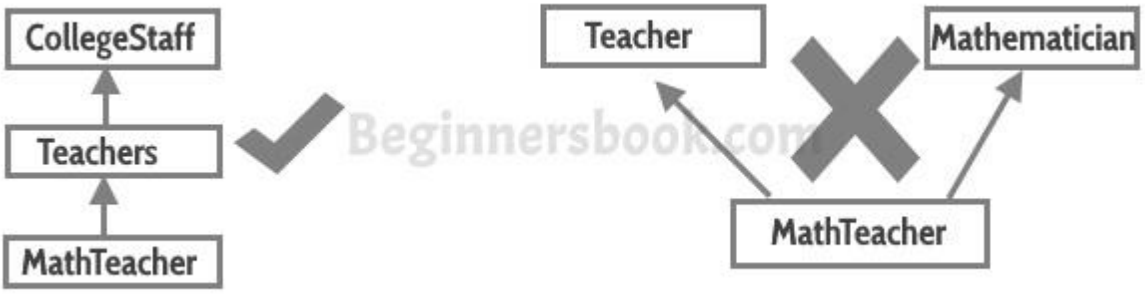
```
class Teacher {
   String designation = "Teacher";
   String college = "Beginnersbook";
   void does(){
       System.out.println("Teaching");
   }
}
public class MathTeacher extends Teacher{
```

```
   String mainSubject = "Maths";
   public static void main(String args[]){
      MathTeacher obj = new MathTeacher();
      System.out.println(obj.college);
      System.out.println(obj.designation);
      System.out.println(obj.mainSubject);
      obj.does();
   }
}
```

Output:

```
Beginnersbook
Teacher
Maths
Teaching
```

Note: Multi-level inheritance is allowed in Java but **not multiple inheritance**



Types of Inheritance:

Single Inheritance: refers to a child and parent class relationship where a class extends the another class.

Multilevel inheritance: refers to a child and parent class relationship where a class extends the child class. For example class A extends class B and class B extends class C.

Hierarchical inheritance: refers to a child and parent class relationship where more than one classes extends the same class. For example, class B extends class A and class C extends class A.

Multiple Inheritance: refers to the concept of one class extending more than one classes, which means a child class has two parent classes. Java doesn't support multiple inheritance, read more about it here.

Most of the new **OO languages** like Small Talk, Java, C# do not support Multiple inheritance. Multiple Inheritance is supported in C++.

Polymorphism

Polymorphism is a object oriented programming feature that allows us to perform a single action in different ways. For example, lets say we have a class `Animal` that has a method `animalSound()`, here we cannot give implementation to this method as we do not know which Animal class would extend `Animal` class. So, we make this method abstract like this:

```
public abstract class Animal{
   ...
   public abstract void animalSound();
}
```

Now suppose we have two Animal classes `Dog` and `Lion` that extends `Animal` class. We can provide the implementation detail there.

```
public class Lion extends Animal{
...
    @Override
    public void animalSound(){
        System.out.println("Roar");
    }
}
```

and

```
public class Dog extends Animal{
...
    @Override
    public void animalSound(){
        System.out.println("Woof");
    }
}
```

As you can see that although we had the common action for all subclasses `animalSound()` but there were different ways to do the same action. This is a perfect example of polymorphism (feature that allows us to perform a single action in different ways).

Types of Polymorphism

1) Static Polymorphism
2) Dynamic Polymorphism

Static Polymorphism:

Polymorphism that is resolved during compiler time is known as static polymorphism. Method overloading can be considered as static polymorphism example.

Method Overloading: This allows us to have more than one methods with same name in a class that differs in signature.

```
class DisplayOverloading
{
    public void disp(char c)
    {
         System.out.println(c);
    }
    public void disp(char c, int num)
    {
         System.out.println(c + " "+num);
    }
}
public class ExampleOverloading
{
   public static void main(String args[])
   {
       DisplayOverloading obj = new DisplayOverloading();
       obj.disp('a');
       obj.disp('a',10);
   }
}
```

Output:

```
a
a 10
```

When I say method signature I am not talking about return type of the method, for example if two methods have same name, same parameters and have different return type, then this is not a valid method overloading example. This will throw compilation error.

Dynamic Polymorphism

It is also known as Dynamic Method Dispatch. Dynamic polymorphism is a process in which a call to an overridden method is resolved at runtime rather, thats why it is called runtime polymorphism.

Example

```
class Animal{
```

```
    public void animalSound(){
        System.out.println("Default Sound");
    }
}
public class Dog extends Animal{

    public void animalSound(){
        System.out.println("Woof");
    }
    public static void main(String args[]){
        Animal obj = new Dog();
        obj.animalSound();
    }
}
```

Output:

```
Woof
```

Since both the classes, child class and parent class have the same method `animalSound`. Which of the method will be called is determined at runtime by JVM.

Few more overriding examples:

```
Animal obj = new Animal();
obj.animalSound();
// This would call the Animal class method

Dog obj = new Dog();
obj.animalSound();
// This would call the Dog class method

Animal obj = new Dog();
obj.animalSound();
// This would call the Dog class method
```

IS-A & HAS-A Relationships

A Car **IS-A** Vehicle and **HAS-A** License then the code would look like this:

```
public class Vehicle{ }
public class Car extends Vehicle{
    private License myCarLicense;
}
```

Abstract Class and methods in OOPs Concepts

Abstract method:

1) A method that is declared but not defined. Only method signature no body.

2) Declared using the abstract keyword
3) Example :

```
abstract public void playInstrument();
```

5) Used to put some kind of compulsion on the class who inherits the class has abstract methods. The class that inherits must provide the implementation of all the abstract methods of parent class else declare the subclass as abstract.
6) These cannot be abstract

- Constructors
- Static methods
- Private methods
- Methods that are declared “final”

Abstract Class
An abstract class outlines the methods but not necessarily implements all the methods.

```
abstract class A{
   abstract void myMethod();
   void anotherMethod(){
      //Does something
   }
}
```

Note 1: There can be some scenarios where it is difficult to implement all the methods in the base class. In such scenarios one can define the base class as an abstract class which signifies that this base class is a special kind of class which is not complete on its own.

A class derived from the abstract base class must implement those methods that are not implemented(means they are abstract) in the abstract class.

Note 2: Abstract class cannot be instantiated which means you cannot create the object of abstract class. To use this class, you need to create another class that extends this abstract class provides the implementation of abstract methods, then you can use the object of that child class to call non-abstract parent class methods as well as implemented methods(those that were abstract in parent but implemented in child class).

Note 3: If a child does not implement all the abstract methods of parent class(the abstract class), then the child class must need to be declared abstract.

Example of Abstract class and Methods

Here we have an abstract class Animal that has an abstract method animalSound(), since the animal sound differs from one animal to another, there is no point in giving the implementation to this method as every child class must override this method to give its own implementation details. That's why we made it abstract.

Now each animal must have a sound, by making this method abstract we made it compulsory to the child class to give implementation details to this method. This way we ensures that every animal has a sound.

```
//abstract class
abstract class Animal{
   //abstract method
   public abstract void animalSound();
}
public class Dog extends Animal{

   public void animalSound(){
       System.out.println("Woof");
   }
   public static void main(String args[]){
       Animal obj = new Dog();
       obj.animalSound();
   }
}
```

Output:

```
Woof
```

Interfaces in Java

An interface is a blueprint of a class, which can be declared by using **interface** keyword. Interfaces can contain only constants and abstract methods (methods with only signatures no body).Like abstract classes, Interfaces cannot be instantiated, they can only be implemented by classes or extended by other interfaces. Interface is a common way to achieve full abstraction in Java.

Note:

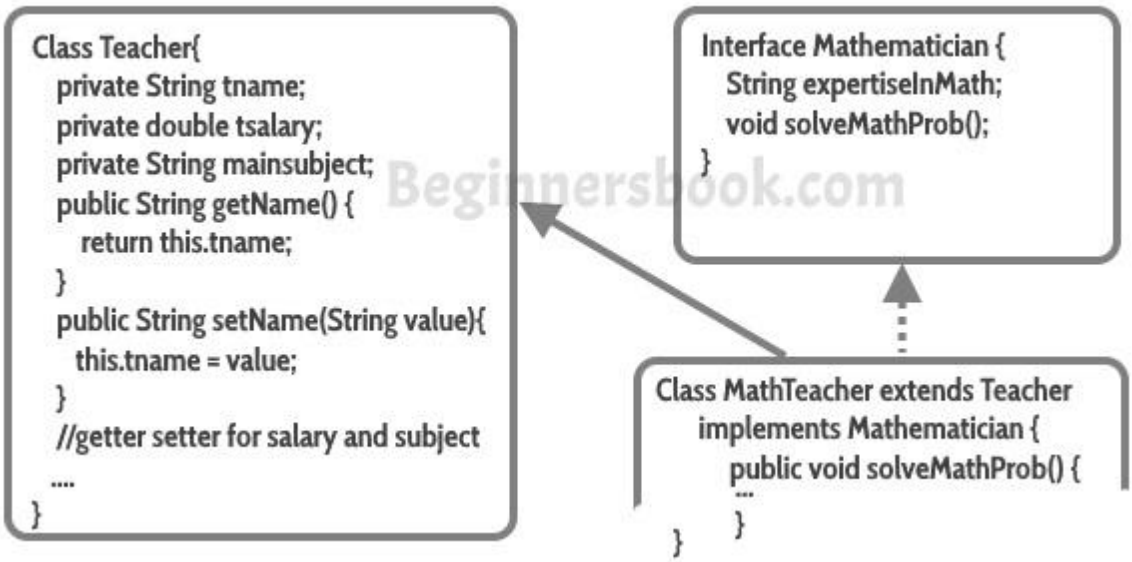
1. Java does not support Multiple Inheritance, however a class can implement more than one interfaces
2. Interface is similar to an abstract class but it contains only abstract methods.

3. Interfaces are created by using interface keyword instead of the keyword class
4. We use `implements` keyword while implementing an interface(similar to extending a class with extends keyword)

Interface: Syntax

```
class ClassName extends Superclass implements Interface1, Interface2, ....
```

Example of Interface:



Note:

1. All **methods in an interface** are implicitly public and abstract. Using the keyword `abstract` before each method is optional.
2. An **interface** may contain final variables.
3. A class can **extend only one other class**, but it can **implement any number of interfaces.**
4. When a class implements an interface it has to give the definition of all the abstract methods of interface, else it can be declared as abstract class
5. An interface reference can point to **objects** of its implementing classes.

Generalization and Specialization:
In order to implement the concept of inheritance in an OOPs, one has to first

identify the similarities among different classes so as to come up with the base class.

This process of identifying the similarities among different classes is called **Generalization**. Generalization is the process of extracting shared characteristics from two or more classes, and combining them into a generalized superclass. Shared characteristics can be attributes or methods.

In contrast to generalization, specialization means creating new subclasses from an existing class. If it turns out that certain attributes or methods only apply to some of the objects of the class, a subclass can be created.

Access Specifiers

Well, you must have seen public, private keyword in the examples I have shared above. They are called access specifiers as they decide the scope of a data member, method or class.

There are **four types** of access specifiers in java:
public: Accessible to all. Other objects can also access this member variable or function.
private: Not accessible by other objects. Private members can be accessed only by the methods in the same class. **Object accessible only in class in which they are declared.**
protected: The scope of a protected variable is within the class which declares it and in the class which inherits from the class (Scope is class and subclass).
Default: Scope is Package Level. We do not need to explicitly mention default as when we do not mention any access specifier it is considered as default.

Source : https://beginnersbook.com/2013/04/oops-concepts/