Ref: **https://beginnersbook.com/**

https://www.geeksforgeeks.org/abstraction-in-java-2/

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

In other words, OOP is a programming model organized around **objects** rather than actions and **data** rather than logic.

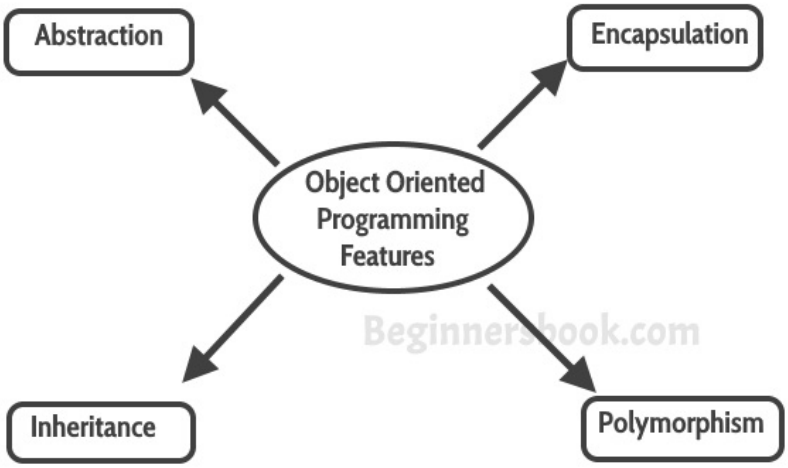
Object is anything that has state, property, behaviour and functionality. Class represents objects.

The primary purpose of **OOPs** is
to increase the flexibility and
to increase maintainability of programs.

OOPs 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.

Features or components of OOP:

Inheritance
Polymorphism
Encapsulation
Abstraction



Inheritance is the technique where one object or class gains the properties of another class or object.

We use extends and implements keywords to inherit properties from one class or object to another.

Method Overloading example: Same method name but with different parameters

```
public class Multiplier {

   public int multiply(int a, int b) {
      return a * b;
   }

   public int multiply(int a, int b, int c) {
      return a * b * c;
   }
}
```

Inheritance is 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.

Advantage of Inheritance:

Code reusability: the code, variables and methods in base (parent) class can be reused in the child class. The code, variables and methods in base (parent) class need not be rewritten in the child class.

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

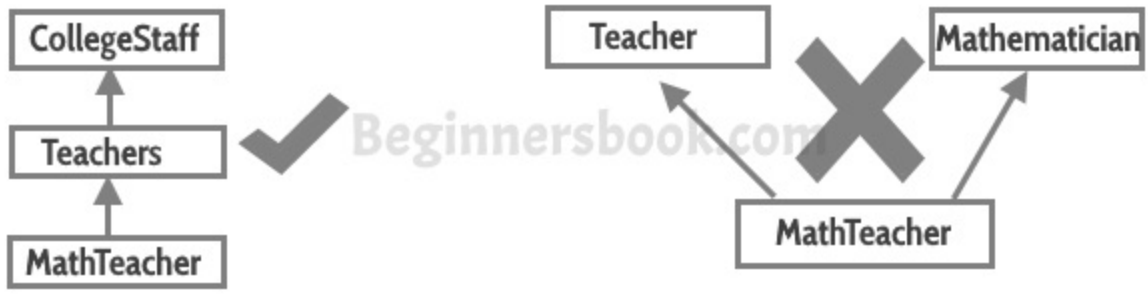
In this example, we have a parent class `Teacher` and a child class `MathTeacher`. In the `MathTeacher`class we need not to write the same code which is already present in the present class. Here we have college name, designation and does() method that is common for all the teachers, thus MathTeacher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

OOPs concept:

OOPs interface vs abstract class

Interface	Abstract class
Interface support multiple implementations.	Abstract class does not support multiple inheritance.
Interface does not contain Data Member	Abstract class contains Data Member
Interface does not contain Constructors	Abstract class contains Constructors
An interface Contains only incomplete member (signature of member)	An abstract class Contains both incomplete (abstract) and complete member
An interface cannot have access modifiers by default everything is assumed as public	An abstract class can contain access modifiers for the subs, functions, properties
Member of interface can not be Static	Only Complete Member of abstract class can be Static

TestInterface.java | TestAbstractClass.java | RegularClass.java | *IntB.java

```
1  package abstractdemo;
2
3  public interface IntB {
4
5      IntB(){
6      Interfaces cannot have constructors
7      Press 'F2' for focus
8
9      }
10     public void getB();
11     public void getBB();
12     public void getB(int b);
13 }
14
```

TestInterface.java | *TestAbstractClass.java | RegularClass.java | *IntB.java

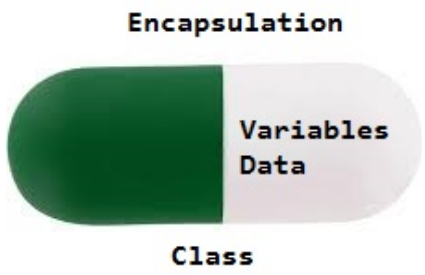
```java
package abstractdemo;

public abstract class TestAbstractClass {


    // constructor of an abstract class
    TestAbstractClass() {

    }

    // constructor of an abstract class
    TestAbstractClass(int c) {

    }

    // constructor of an abstract class
    TestAbstractClass(double pound, double dollar) {

    }

```

public class RegularClass extends TestAbstractClass implements TestInterface, IntB {

// Class to class inheritance: extends
// class to interface: implements
// interface to interface: extends
// once class can inherit JUST one class at a time
// but once class can inherit JUST one class at a time and multiple interface (separated by comma)

}

Java Encapsulation is the technique of providing access on **private variables** or methods through **a public method**.



Benefits of encapsulation:

The field of a class can be made read-only or write-only
A class can have total control over what is stored in its fields
The users of a class do not know how the class stores its data.

Encapsulation coding example:

```
package
oops.demo
;
```

```java
public class EncapsulationDemo {

    private int ssn;
    private String empName;
    private int empAge;

    public int getSsn() {
        return ssn;
    }

    public void setSsn(int ssn) {
        this.ssn = ssn;
    }

    public String getEmpName() {
        return empName;
    }

    public void setEmpName(String empName) {
        this.empName = empName;
    }

    public int getEmpAge() {
        return empAge;
    }

    public void setEmpAge(int empAge) {
        this.empAge = empAge;
    }

    public static void main(String[] args) {

EncapsulationDemo ed = new EncapsulationDemo();
        ed.setEmpAge(34);
        ed.setEmpName("Ameer");
```

```
            ed.setSsn(123456);


        System.out.println("The name of the employee is :
        " + ed.getEmpName());
            System.out.println("The age of the
        employee is : " + ed.getEmpAge());
            System.out.println("The ssn of the
        employee is : " + ed.getSsn());


        }


        }
```

Encapsulation steps:

1. Declare private variables
2. Generate setters and getters methods
3. Create an object of the class
4. Pass values to the setter methods
5. Read values using public getter methods

Abstraction: is the ability to make a class abstract.

How can we make a class abstract?

abstract keyword makes a class abstract
An abstract class may or may not have an abstract method

But to make a functional **abstract class** there must be at least one abstract method

An example of a regular class:

```
public class MyTestClass {



}
```

An example of an abstract class:

```
public abstract class TestAbstractClass {

}
```

Indication of an abstract method:

```
// an abstract method: is a method that has no method body - {   }
// (i) no body (ii) abstract keyword
public abstract void getPrinted();
```

Abstract class:

- is a class that has **abstract** keyword in its class declaration:

```
public abstract class TestAbst {

}
```

- may or may not contain abstract method
- cannot be instantiated (object of an abstract class cannot be created)

```java
package abstr;

public abstract class TestAbst { // TestAbst -> is a regular class

    // a regular class can't have an abstract method

    // regular method:
    public static void main(String[] args) {
        TestAbst ta = new TestAbst();  // object of an abstract class can't be created

    }
```

Cannot instantiate the type TestAbst

- to use an abstract class (or abstract methods from an Interface), it must be inherited by another **regular** class to provide implementation of all the abstract methods in it
- **extends** keyword is required to inherit an abstract class and **implements** keyword to inherit from an Interface

```java
package abstr;

public class ActualClass extends TestAbst implements InterfaceA{

    public static void main(String[] args) {

    }

    @Override
    public void add(int a) {
        // TODO Auto-generated method stub

    }

    @Override
    public void add(int b, int c) {
        // TODO Auto-generated method stub

    }
```

abstract class

interface

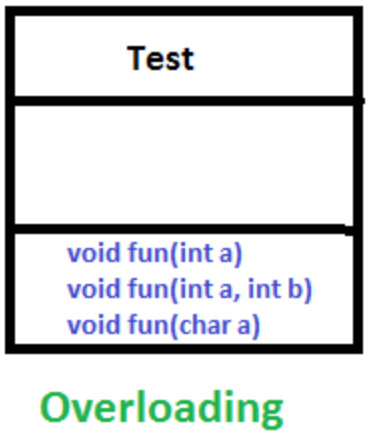
Abstract method is

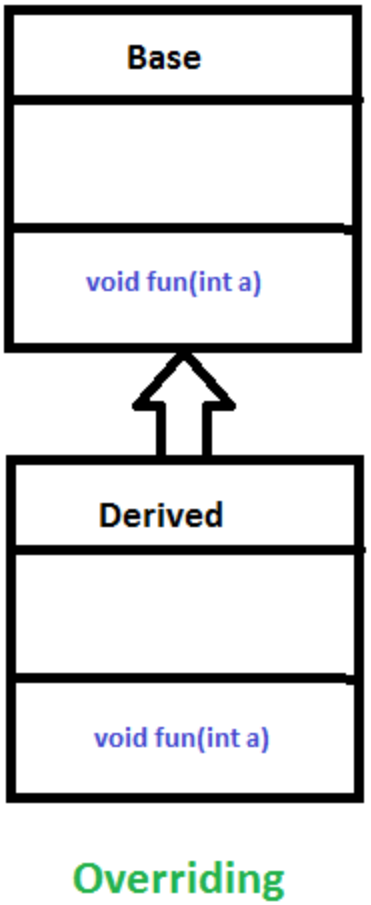
- any method that has only method declaration without implementation details (no method body and algorithms / codes / logics). { }

- must need **abstract** keyword in its declaration unless its a method inside an interface - methods inside an interface are by-born (default) abstract
- has no body or implementation details
- must be implemented in child class
- Needs method **overriding** to implement abstract methods in child class

Interface is a group of related methods with no method body.

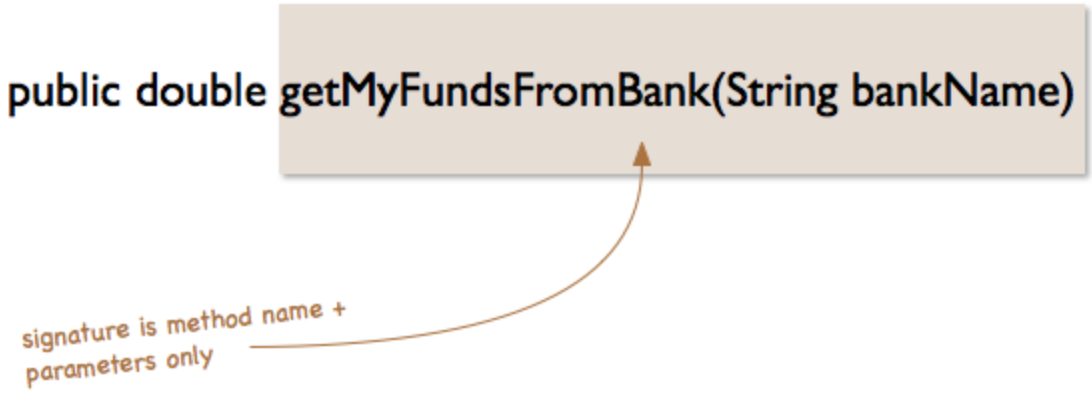
Further study: https://www.geeksforgeeks.org/abstraction-in-java-2/

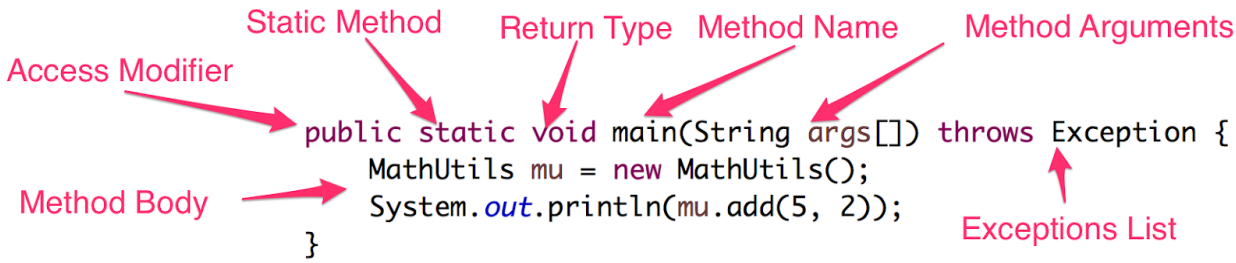




In **Java**, a **method signature** is part of the **method** declaration. It's the combination of the **method** name and the parameter list. The reason for the emphasis on just the

method name and parameter list is because of overloading. It's the ability to write **methods** that have the same name but accept different parameters.





In Java polymorphism is mainly divided into two types:

- Compile time Polymorphism

Runtime Polymorphism

// Java program for Method overridding

```
class Parent {
   void Print()    {
      System.out.println("parent class");
   }
}
```

```
class subclass1 extends Parent {

   void Print()    {
      System.out.println("subclass1");
   }
}

class subclass2 extends Parent {

   void Print() {
      System.out.println("subclass2");
   }
}

class TestPolymorphism3 {
   public static void main(String[] args){

      Parent a;

      a = new subclass1();
      a.Print();

      a = new subclass2();
      a.Print();
   }
}
```

Output:
subclass1
subclass2

Method overriding (run-time polymorphism):
Relationship between parent and child class having the same method names

No.	Method Overloading	Method Overriding
1)	Method overloading is used *to increase the readability* of the program.	Method overriding is used *to provide the specific implementation* of the method

		that is already provided by its super class.
2)	Method overloading is performed *within class*.	Method overriding occurs *in two classes* that have IS-A (inheritance) relationship.
3)	In case of method overloading, *parameter must be different*.	In case of method overriding, *parameter must be same*.
4)	Method overloading is the example of *compile time polymorphism*.	Method overriding is the example of *run time polymorphism*.
5)	In java, method overloading can't be performed by changing return type of the method only. *Return type can be same or different* in method overloading. But you must have to change the parameter.	*Return type must be same or covariant* in method overriding.