**Object Oriented Programming (OOPs) Concept in Java**

As the name suggests, **Object-Oriented Programming** or **Java OOPs** concept refers to languages that use objects in programming, they use objects as a primary source to implement what is to happen in the code. Objects are seen by the viewer or user, performing tasks you assign.

**Object-oriented programming** aims to implement real-world entities like **inheritance**, **hiding**, **polymorphism**, etc. in programming. The main aim of OOPs is to bind together the data and the functions that operate on them so that no other part of the code can access this data except that function.

Object-Oriented Programming is the backbone of Java. Mastering OOP concepts like inheritance, encapsulation, and polymorphism is critical for writing scalable Java code. The [**Java Programming Course**](https://gfgcdn.com/tu/S4l/) takes you through these concepts step by step, providing practical examples that reinforce your learning

**Pre-requisites:**

Let us discuss prerequisites by polishing concepts of **method declaration** and **message passing**. Starting with the method declaration consists of six components:

1. [**Access Modifier**](https://www.geeksforgeeks.org/access-modifiers-java): Defines the **access type** of the method i.e. from where it can be accessed in your application. In Java, there are 4 types of access specifiers:   
   * **public:** Accessible in all classes in your application.
   * **protected:** Accessible within the package in which it is defined and in its **subclass(es) (including subclasses declared outside the package)**.
   * **private:** Accessible only within the class in which it is defined.
   * **default (declared/defined without using any modifier):** Accessible within the same class and package within which its class is defined.
2. **The return type**: The data type of the value returned by the method or void if it does not return a value.
3. **Method Name**: The rules for field names apply to method names as well, but the convention is a little different.
4. **Parameter list**: Comma-separated list of the input parameters that are defined, preceded by their data type, within the enclosed parentheses. If there are no parameters, you must use empty parentheses ().
5. **Exception list**: The exceptions you expect the method to throw. You can specify these exception(s).
6. **Method body**: It is the block of code, enclosed between braces, that you need to execute to perform your intended operations.

[**Message Passing**](https://www.geeksforgeeks.org/message-passing-in-java)**:** Objects communicate with one another by sending and receiving information to each other. A message for an object is a request for execution of a procedure and therefore will invoke a function in the receiving object that generates the desired results. Message passing involves specifying the name of the object, the name of the function and the information to be sent.

***Master OOP in Java*** *Write cleaner, more modular, and reusable Java code by building a foundation in object-oriented programming with Educative’s interactive course Learn* [*Object-Oriented Programming in Java*](https://www.educative.io/geeks-oops)*. Sign up at Educative.io with the code* ***GEEKS10*** *to save 10% on your subscription.*

**JAVA OOPs**

Now that we have covered the basic prerequisites, we will move on to the **4 pillars of OOPs** which are as follows. But, let us start by learning about the different characteristics of an Object-Oriented Programming Language.

**OOPS concepts are as follows**:

1. [Class](https://www.geeksforgeeks.org/classes-objects-java)
2. [Object](https://www.geeksforgeeks.org/classes-objects-java)
3. **Pillars of OOPs**
   * [Abstraction](https://www.geeksforgeeks.org/abstraction-in-java-2)
   * [Encapsulation](https://www.geeksforgeeks.org/encapsulation-in-java)
   * [Inheritance](https://www.geeksforgeeks.org/inheritance-in-java)
   * [Polymorphism](https://www.geeksforgeeks.org/polymorphism-in-java)
     + Compile-time polymorphism
     + Runtime polymorphism