## OOP Concepts in Java

The main ideas behind Java’s Object-Oriented Programming, OOP concepts include [**abstraction**](https://stackify.com/oop-concept-abstraction/)**,**[**encapsulation**](https://stackify.com/oop-concept-for-beginners-what-is-encapsulation/), [**inheritance**](https://stackify.com/oop-concept-inheritance/) and[**polymorphism**](https://stackify.com/oop-concept-polymorphism/). Basically, Java OOP concepts let us create working methods and variables, then re-use all or part of them without compromising security. Grasping OOP concepts is key to understanding how Java works.

Java defines OOP concepts as follows:

* **Abstraction.** Using simple things to represent complexity. We all know how to turn the TV on, but we don’t need to know how it works in order to enjoy it. In Java, abstraction means simple things like **objects**, **classes** and **variables** represent more complex underlying code and data. This is important because it lets you avoid repeating the same work multiple times.
* **Encapsulation.**The practice of keeping fields within a class private, then providing access to those fields via public methods. Encapsulation is a protective barrier that keeps the data and code safe within the class itself. We can then reuse objects like code components or variables without allowing open access to the data system-wide.
* **Inheritance.**A special feature of Object-Oriented Programming in Java, Inheritance lets programmers create new classes that share some of the attributes of existing classes. Using Inheritance lets us build on previous work without reinventing the wheel.
* **Polymorphism.**Allows programmers to use the same word in Java to mean different things in different contexts. One form of polymorphism is **method overloading**. That’s when the code itself implies different meanings. The other form is **method overriding**. That’s when the values of the supplied variables imply different meanings. Let’s delve a little further.

### **How Abstraction Works**

Abstraction lets programmers create useful and reusable tools. For example, a programmer can create several different types of **objects**, which can be variables, functions or data structures. Programmers can also create different **classes** of objects as ways to define the objects.

For instance, a class of variable might be an address. The class might specify that each address object shall have a name, street, city and zip code. The objects, in this case, might be employee addresses, customer addresses or supplier addresses.

### **How Encapsulation Works**

Encapsulation lets us reuse functionality without jeopardizing security. It’s a powerful, time-saving OOP concept in Java. For example, we may create a piece of code that calls specific data from a database. It may be useful to reuse that code with other databases or processes. Encapsulation lets us do that while keeping our original data private. It also lets us alter our original code without breaking it for others who have adopted it in the meantime.

### **How Inheritance Works**

Inheritance is another labor-saving Java OOP concept that works by letting a new class adopt the properties of another. We call the inheriting class a **subclass** or a **child class**. The original class is often called the **parent**. We use the keyword **extends** to define a new class that inherits properties from an old class.

### **How Polymorphism Works**

Polymorphism in Java works by using a reference to a parent class to affect an object in the child class. We might create a class called “horse” by extending the “animal” class. That class might also implement the “professional racing” class. The “horse” class is “polymorphic,” since it inherits attributes of both the “animal” and “professional racing” class.

Two more examples of polymorphism in Java are method overriding and method overloading.

In **method** **overriding**, the child class can use the OOP polymorphism concept to override a method of its parent class. That allows a programmer to use one method in different ways depending on whether it’s invoked by an object of the parent class or an object of the child class.

In **method overloading,**a single method may perform different functions depending on the context in which it’s called. This means a single method name might work in different ways depending on what arguments are passed to it.

**OOP Principal in Java**

* [**S** - Single-responsiblity Principle](https://www.digitalocean.com/community/conceptual_articles/s-o-l-i-d-the-first-five-principles-of-object-oriented-design#single-responsibility-principle)
* [**O** - Open-closed Principle](https://www.digitalocean.com/community/conceptual_articles/s-o-l-i-d-the-first-five-principles-of-object-oriented-design#open-closed-principle)
* [**L** - Liskov Substitution Principle](https://www.digitalocean.com/community/conceptual_articles/s-o-l-i-d-the-first-five-principles-of-object-oriented-design#liskov-substitution-principle)
* [**I** - Interface Segregation Principle](https://www.digitalocean.com/community/conceptual_articles/s-o-l-i-d-the-first-five-principles-of-object-oriented-design#interface-segregation-principle)
* [**D** - Dependency Inversion Principle](https://www.digitalocean.com/community/conceptual_articles/s-o-l-i-d-the-first-five-principles-of-object-oriented-design#dependency-inversion-principle)

**Single-responsibility Principle**

*A class should have one and only one reason to change, meaning that a class should have only one job.*

this principle states that **a class should only have one responsibility. Furthermore, it should only have one reason to change.**

**How does this principle help us to build better software?** Let's see a few of its benefits:

1. **Testing** – A class with one responsibility will have far fewer test cases.
2. **Lower coupling** – Less functionality in a single class will have fewer dependencies.
3. **Organization** – Smaller, well-organized classes are easier to search than monolithic ones.

For example:

**public** **class** **Book** {

**private** String name;

**private** String author;

**private** String text; //constructor, getters and setters

}

This is good

**public** **class** **Book** {

**private** String name;

**private** String author;

**private** String text; //constructor, getters and setters // methods that directly relate to the book properties

**public** String **replaceWordInText**(String word){

**return** text.replaceAll(word, text);

}

**public** **boolean** **isWordInText**(String word){

**return** text.contains(word);

}

}

This is not good.

**public** **class** **Book** {

//...

**void** **printTextToConsole**(){

// our code for formatting and printing the text

}

}

**Open-Closed Principle:**

*Objects or entities should be open for extension but closed for modification.*

*For example:*

**public** **class** **Guitar** {

**private** String make;

**private** String model;

**private** **int** volume;

//Constructors, getters & setters

}

**public** **class** **SuperCoolGuitarWithFlames** **extends** **Guitar**{

**private** String flameColor;

//constructor, getters + setters

}

**Liskov Substitution Principle:**

***if class*A*is a subtype of class*B*, we should be able to replace*B *with*A *without disrupting the behavior of our program.***

for example:

**public** **interface** **Car** {

**void** **turnOnEngine**();

**void** **accelerate**();

}

**public** **class** **MotorCar** **implements** **Car** {

**private** Engine engine; //Constructors, getters + setters

**public** **void** **turnOnEngine**() { //turn on the engine!

engine.on();

}

**public** **void** **accelerate**() { //move forward!

engine.powerOn(1000);

}

}

**public** **class** **ElectricCar** **implements** **Car** {

**public** **void** **turnOnEngine**() {

**throw** **new** **AssertionError**("I don't have an engine!");

}

**public** **void** **accelerate**() {

//this acceleration is crazy!

}

}

***Interface Segregation Principle:***

**larger interfaces should be split into smaller ones. By doing so, we can ensure that implementing classes only need to be concerned about the methods that are of interest to them.**

**Gang of Four Design Patterns**

### **Creational Design Patterns**

* [Abstract Factory](https://springframework.guru/gang-of-four-design-patterns/abstract-factory-design-pattern/). Allows the creation of objects without specifying their concrete type.
* [Builder](https://springframework.guru/gang-of-four-design-patterns/builder-pattern/). Uses to create complex objects.
* [Factory Method](https://springframework.guru/gang-of-four-design-patterns/factory-method-design-pattern/). Creates objects without specifying the exact class to create.
* [Prototype](https://springframework.guru/gang-of-four-design-patterns/prototype-pattern/). Creates a new object from an existing object.
* [Singleton](https://springframework.guru/gang-of-four-design-patterns/singleton-design-pattern/). Ensures only one instance of an object is created.

### **Structural Design Patterns**

* [Adapter](https://springframework.guru/gang-of-four-design-patterns/adapter-pattern/). Allows for two incompatible classes to work together by wrapping an interface around one of the existing classes.
* [Bridge](https://springframework.guru/gang-of-four-design-patterns/bridge-pattern/). Decouples an abstraction so two classes can vary independently.
* [Composite](https://springframework.guru/gang-of-four-design-patterns/composite-pattern/). Takes a group of objects into a single object.
* [Decorator](https://springframework.guru/gang-of-four-design-patterns/decorator-pattern/). Allows for an object’s behaviour to be extended dynamically at run time.
* [Facade](https://springframework.guru/gang-of-four-design-patterns/facade-pattern/). Provides a simple interface to a more complex underlying object.
* [Flyweight](https://springframework.guru/gang-of-four-design-patterns/flyweight-pattern/). Reduces the cost of complex object models.
* [Proxy](https://springframework.guru/gang-of-four-design-patterns/proxy-pattern/). Provides a placeholder interface to an underlying object to control access, reduce cost, or reduce complexity.

### **Behavior Design Patterns**

* [Chain of Responsibility](https://springframework.guru/gang-of-four-design-patterns/chain-of-responsibility-pattern/). Delegates commands to a chain of processing objects.
* [Command](https://springframework.guru/gang-of-four-design-patterns/command-pattern/). Creates objects which encapsulate actions and parameters.
* [Interpreter](https://springframework.guru/gang-of-four-design-patterns/interpreter-pattern/). Implements a specialized language.
* [Iterator](https://springframework.guru/gang-of-four-design-patterns/iterator-pattern/). Accesses the elements of an object sequentially without exposing its underlying representation.
* [Mediator](https://springframework.guru/gang-of-four-design-patterns/mediator-pattern/). Allows loose coupling between classes by being the only class that has detailed knowledge of their methods.
* [Memento](https://springframework.guru/gang-of-four-design-patterns/memento-pattern/). Provides the ability to restore an object to its previous state.
* [Observer](https://springframework.guru/gang-of-four-design-patterns/observer-pattern/). Is a publish/subscribe pattern which allows a number of observer objects to see an event.
* [State](https://springframework.guru/gang-of-four-design-patterns/state-pattern/). Allows an object to alter its behavior when its internal state changes.
* [Strategy](https://springframework.guru/gang-of-four-design-patterns/strategy-pattern/). Allows one of a family of algorithms to be selected on-the-fly at run-time.
* [Template Method](https://springframework.guru/gang-of-four-design-patterns/template-method-pattern/). Defines the skeleton of an algorithm as an abstract class, allowing its sub-classes to provide concrete behavior.
* [Visitor](https://springframework.guru/gang-of-four-design-patterns/visitor-pattern/). Separates an algorithm from an object structure by moving the hierarchy of methods into one object.