<https://youtu.be/aqTb1f9SqF8>

# OOP Concept for Beginners: What is Encapsulation

THORBEN JANSSENNOVEMBER 30, 2017[DEVELOPER TIPS, TRICKS & RESOURCES](https://stackify.com/developers/)

Encapsulation is one of the fundamental [concepts in object-oriented programming](https://stackify.com/oops-concepts-in-java/)(OOP). It describes the idea of bundling data and methods that work on that data within one unit, e.g., a class in Java.

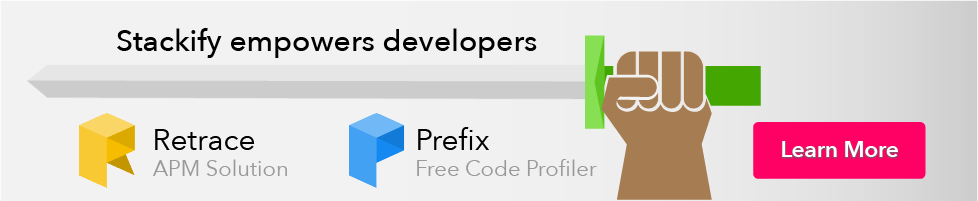
This concept is also often used to hide the internal representation, or state, of an object from the outside. This is called[information hiding](https://en.wikipedia.org/wiki/Encapsulation_(computer_programming)#An_information-hiding_mechanism). The general idea of this mechanism is simple. If you have an attribute that is not visible from the outside of an object, and bundle it with methods that provide read or write access to it, then you can hide specific information and control access to the internal state of the object.

If you’re familiar with any object-oriented programming language, you probably know that these methods as getter and setter methods. As the names indicate, a getter method retrieves an attribute, and a setter method changes it. Depending on the methods that you implement, you can decide if an attribute can be read and changed, or if it’s read-only, or if it is not visible at all. As I will show you later, you can also use the setter method to implement additional validation rules to ensure that your object always has a valid state.

Let’s take a look at an example that shows the concept of encapsulation and how you can use it to implement information hiding and apply additional validation before changing the values of your object attributes.

## ****Encapsulation in Java****

If you’ve read my previous post about [abstraction](https://stackify.com/oop-concept-abstraction/), you already saw several examples for encapsulation. It’s such a basic concept that most Java developers use it without thinking about it. It’s simply how you design a Java class. You bundle a set of attributes that store the current state of the object with a set of methods using these attributes.

[](https://info.stackify.com/cs/c/?cta_guid=eb8ba9dc-2898-4c16-a893-432ae7a086a9&placement_guid=c00f5072-32fb-4332-a4d7-60765e6305d0&portal_id=207384&canon=https%3A%2F%2Fstackify.com%2Foop-concept-for-beginners-what-is-encapsulation%2F&redirect_url=APefjpEtlBPN3N6Y-6jkUJKFrXB_VSRD2Qikva4VBNChzsyQIukwtKaDMZG8qJ7TUr7otnQiwkzuzYV1vo85x9by1ig8ks3wpSZ1tofCkMs2L6NKJIhBImw&click=dac5e7a2-6939-4942-b1e1-3f7d714ba77f&hsutk=c7a0000016961a951ea8016c109c6ccd&utm_referrer=https%3A%2F%2Fwww.google.com%2F)

### ****The****CoffeeMachine****example****

I did that, for example, when I created the CoffeeMachine class. The attributes configMap, beans, grinder, and brewingUnit store the current state of the CoffeeMachine object. The methods brewCoffee, brewEspresso, brewFilterCoffee and addBeans implement a set of operations on these attributes.

You can clone this and all other classes of the CoffeeMachine example project at <https://github.com/thjanssen/Stackify-OopAbstraction>.

import java.util.HashMap;

import java.util.Map;

public class CoffeeMachine {

private Map configMap;

private Map beans;

private Grinder grinder;

private BrewingUnit brewingUnit;

public CoffeeMachine(Map beans) {

this.beans = beans;

this.grinder = new Grinder();

this.brewingUnit = new BrewingUnit();

this.configMap = new HashMap();

this.configMap.put(CoffeeSelection.ESPRESSO, new Configuration(8, 28));

this.configMap.put(CoffeeSelection.FILTER\_COFFEE, new Configuration(30, 480));

}

public Coffee brewCoffee(CoffeeSelection selection) throws CoffeeException {

switch (selection) {

case FILTER\_COFFEE:

return brewFilterCoffee();

case ESPRESSO:

return brewEspresso();

default:

throw new CoffeeException("CoffeeSelection [" + selection + "] not supported!");

}

}

private Coffee brewEspresso() {

Configuration config = configMap.get(CoffeeSelection.ESPRESSO);

// grind the coffee beans

GroundCoffee groundCoffee = this.grinder.grind(

this.beans.get(CoffeeSelection.ESPRESSO), config.getQuantityCoffee());

// brew an espresso

return this.brewingUnit.brew(CoffeeSelection.ESPRESSO,

groundCoffee, config.getQuantityWater());

}

private Coffee brewFilterCoffee() {

Configuration config = configMap.get(CoffeeSelection.FILTER\_COFFEE);

// grind the coffee beans

GroundCoffee groundCoffee = this.grinder.grind(

this.beans.get(CoffeeSelection.FILTER\_COFFEE), config.getQuantityCoffee());

// brew a filter coffee

return this.brewingUnit.brew(CoffeeSelection.FILTER\_COFFEE,

groundCoffee, config.getQuantityWater());

}

public void addBeans(CoffeeSelection sel, CoffeeBean newBeans) throws CoffeeException {

CoffeeBean existingBeans = this.beans.get(sel);

if (existingBeans != null) {

if (existingBeans.getName().equals(newBeans.getName())) {

existingBeans.setQuantity(existingBeans.getQuantity() + newBeans.getQuantity());

} else {

throw new CoffeeException("Only one kind of beans supported for each CoffeeSelection.");

}

} else {

this.beans.put(sel, newBeans);

}

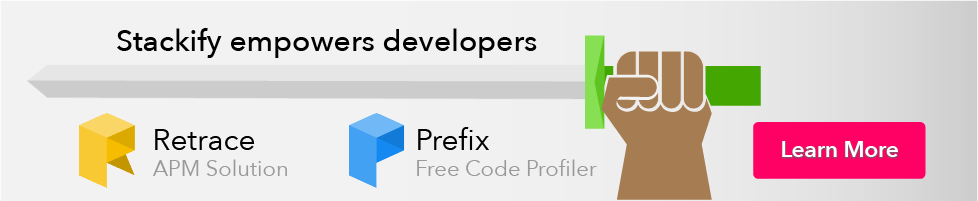
}

}

### ****Information hiding in Java****

As explained at the beginning, you can use the encapsulation concept to implement an information-hiding mechanism. Similar to the abstraction concept, this is one of the most commonly used mechanisms in Java. You can find examples of it in almost all well-implemented Java classes.

You implement this information-hiding mechanism by making your class attributes inaccessible from the outside and by providing getter and/or setter methods for attributes that shall be readable or updatable by other classes.

[](https://info.stackify.com/cs/c/?cta_guid=eb8ba9dc-2898-4c16-a893-432ae7a086a9&placement_guid=c00f5072-32fb-4332-a4d7-60765e6305d0&portal_id=207384&canon=https%3A%2F%2Fstackify.com%2Foop-concept-for-beginners-what-is-encapsulation%2F&redirect_url=APefjpEtlBPN3N6Y-6jkUJKFrXB_VSRD2Qikva4VBNChzsyQIukwtKaDMZG8qJ7TUr7otnQiwkzuzYV1vo85x9by1ig8ks3wpSZ1tofCkMs2L6NKJIhBImw&click=dac5e7a2-6939-4942-b1e1-3f7d714ba77f&hsutk=c7a0000016961a951ea8016c109c6ccd&utm_referrer=https%3A%2F%2Fwww.google.com%2F)

#### ****Access Modifiers****

Java supports four [access modifiers](https://docs.oracle.com/javase/tutorial/java/javaOO/accesscontrol.html) that you can use to define the visibility of classes, methods, and attributes. Each of them specifies a different level of accessibility, and you can only use one modifier per class, method or attribute. As a rule of thumb, you should always use the most restrictive modifier that still allows you to implement your business logic.

These modifiers are, starting from the most to the least restrictive one:

* private
* no modifier
* protected
* public

Let’s take a closer look at each of these modifiers and discuss when you should use them.

##### ****Private****

This is the most restrictive and most commonly used access modifier. If you use the private modifier with an attribute or method, it can only be accessed within the same class. Subclasses or any other classes within the same or a different package can’t access this attribute or method.

As a rule of thumb, the private modifier should be your default choice for all attributes and internal methods that shouldn’t be called from external classes. You might need to make an exception to this rule when you’re using inheritance, and some of the subclasses need direct access to an attribute or internal method. In that case, you should use the protected modifier instead of private.

##### ****No modifier****

When you don’t provide any access modifier for your attribute or method, you can access it within your class and from all classes within the same package.  That’s why it’s often called package-private.

I use the private modifier to restrict access to all attributes as well as the brewEspresso and brewFilterCoffee methods in the CoffeeMachine example.  These attributes and methods should only be used within the CoffeeMachine class and are not part of the public API.

That might seem a bit confusing in the beginning, but it’s very useful when the classes in your package implement a well-defined set of logic, and you want to control the API that’s available to classes outside of this package. You can then use package visibility to implement a method that can only be used by classes within this package. That allows you to create a package internal and an external API.

##### ****Protected****

Attributes and methods with the access modifier protected can be accessed within your class, by all classes within the same package, and by all subclasses within the same or other packages.

The protected modifier gets mostly used for internal methods that need to be called or overridden by subclasses. You can also use it to allow subclasses to access internal attributes of a superclass directly.

##### ****Public****

This is the least restrictive access modifier. Methods and attributes that use the publicmodifier can be accessed within your current class and by all other classes.

Public methods and attributes become part of the public API of your class and of any component in which you include them. That is almost never a good idea for any attribute, and you should think twice before you use this modifier on a method.

If a method is publicly available, you need to make sure that it’s well documented and that it robustly handles any input values. Also keep in mind, that sooner or later this method will be used by some part of your application that will make it hard to change or remove it.

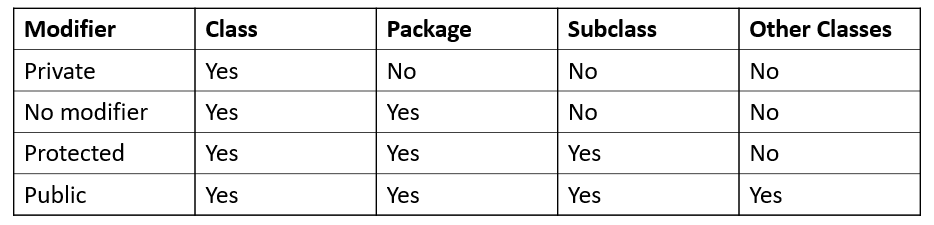
In general, your public API should be as lean as possible and only include the methods which are intended to be used by other parts of the application or by external clients.

That’s the case for the *CoffeeMachine*class, its constructor, and the *brewCoffee*and *addBeans*methods. The *CoffeeMachine*class has to be public because it represents the interface of the coffee machine. It is intended to be used by other classes that don’t have to be part of the same package. The constructor and the *brewCoffee*and *addBeans*methods can be called by other classes to create a new instance of the *CoffeeMachine*and to interact with it by adding coffee beans or by brewing a fresh cup of coffee.

The *brewCoffee*method shows another benefit of the different access modifiers. You can not only use it to hide information, but you can also use to support abstraction. The public *brewCoffee*method abstracts the internal details of the *brewFilterCoffee*and *brewEspresso*methods, which are both private. The access modifiers ensure that an external class can only call the abstraction provided by the *brewCoffee*method, but not the internal methods.

##### ****Accessibility matrix****

Here you can see an overview of the different access modifiers and the accessibility of the attributes or methods.



#### ****The****Coffee****example****

The Coffee class provides a good example of the information-hiding mechanism. It represents a drink that was brewed by the CoffeeMachine.

public class Coffee {

private CoffeeSelection selection;

private double quantity;

public Coffee (CoffeeSelection selection, double quantity) {

this.selection = selection;

this.quantity = quantity;

}

public CoffeeSelection getSelection() {

return selection;

}

public double getQuantity() {

return quantity;

}

public void setQuantity(double quantity) throws CoffeeException {

if (quantity >= 0.0) {

this.quantity = quantity;

} else {

throw new CoffeeException("Quantity has to be >= 0.0.");

}

}

}

The class uses two private attributes to store information about the CoffeeSelectionand the quantity of the drink. The access modifier private makes both attributes inaccessible for other classes within the same or other packages. If you want to get information about the current state of the object, you need to call one of the publicmethods.

The getSelection method provides read access to the selection attribute. It represents the kind of coffee that was brewed by the CoffeeMachine, e.g., a filter coffee or an espresso. As you can see in the code snippet, I didn’t implement a setter method for this attribute because you can’t change the kind of coffee after it was brewed. At least I don’t know of any way to change a boring filter coffee into a strong and tasty espresso.

The available quantity of a drink changes over time. After every sip you take, your cup contains a little bit less. Therefore, I implemented a getter and setter method for the quantity attribute.

If you take a closer look at the setQuantity method, you can see that I also implemented an additional validation. If the coffee is especially delicious, you might drink it until your cup is empty. When you did that, your coffee is gone, and you can’t drink any more of it. So the quantity of the Coffee has to be greater or equal to zero.

## ****Summary****