**Abstract:**

Abstraction is one of the [key concepts](https://stackify.com/oops-concepts-in-java/) of object-oriented programming (OOP) languages. Its main goal is to handle complexity by hiding unnecessary details from the user. That enables the user to implement more complex logic on top of the provided abstraction without understanding or even thinking about all the hidden complexity.

That’s a very generic concept that’s not limited to object-oriented programming. We can find it everywhere in the real world.

It’s main purpose is to hide details, and only expose a high-level mechanism for using it. Implementation details should not be shown.

An example of abstraction, is an [**abstract class**](https://medium.com/@ellehallal/interfaces-abstract-classes-and-concrete-classes-13af02ae96cf). An abstract class can contain abstract methods. This means a class extending from the abstract class, needs to implement the abstract methods. It isn’t concerned with how it is done, as long as the abstract methods are implemented.

**Abstraction in the real world**

I’m a coffee addict. So, when I wake up in the morning, I go into my kitchen, switch on the coffee machine and make coffee. Sounds familiar?

Making coffee with a coffee machine is a good example of abstraction.

You need to know how to use your coffee machine to make coffee. You need to provide water and coffee beans, switch it on and select the kind of coffee you want to get.

The thing you don’t need to know is how the coffee machine is working internally to brew a fresh cup of delicious coffee. You don’t need to know the ideal temperature of the water or the amount of ground coffee you need to use.

Someone else worried about that and created a coffee machine that now acts as an abstraction and hides all these details. You just interact with a simple interface that doesn’t require any knowledge about the internal implementation.

We can use the same concept in object-oriented programming languages like Java.

## Abstraction in OOP

Objects in an OOP language provide an abstraction that hides the internal implementation details. Similar to the coffee machine in your kitchen, you just need to know which methods of the object are available to call and which input parameters are needed to trigger a specific operation. But you don’t need to understand how this method is implemented and which kinds of actions it has to perform to create the expected result.

Let’s implement the coffee machine example in Java. You do the same in any other object-oriented programming language. The syntax might be a little bit different, but the general concept is the same.

### Use abstraction to implement a coffee machine

Modern coffee machines have become pretty complex. Depending on your choice of coffee, they decide which of the available coffee beans to use and how to grind them. They also use the right amount of water and heat it to the required temperature to brew a huge cup of filter coffee or a small and strong espresso.

### Implementing the CoffeeMachine abstraction

Using the concept of abstraction, you can hide all these decisions and processing steps within your CoffeeMachine class. If you want to keep it as simple as possible, you just need a constructor method that takes a Map of CoffeeBean objects to create a new CoffeeMachine object and a brewCoffee method that expects your CoffeeSelection and returns a Coffee object.

You can clone the source of the example project at <https://github.com/thjanssen/Stackify-OopAbstraction>.

import org.thoughts.on.java.coffee.CoffeeException;

import java.utils.Map;

public class CoffeeMachine {

private Map<CoffeeSelection, CoffeeBean> beans;

public CoffeeMachine(Map<CoffeeSelection, CoffeeBean> beans) {

this.beans = beans

}

public Coffee brewCoffee(CoffeeSelection selection) throws CoffeeException {

Coffee coffee = new Coffee();

System.out.println(“Making coffee ...”);

return coffee;

}

}

offeeSelection is a simple enum providing a set of predefined values for the different kinds of coffees.

public enum CoffeeSelection {

FILTER\_COFFEE, ESPRESSO, CAPPUCCINO;

}

And the classes CoffeeBean and Coffee are simple POJOs (plain old Java objects) that only store a set of attributes without providing any logic.

public class CoffeeBean {

private String name;

private double quantity;

public CoffeeBean(String name, double quantity) {

this.name = name;

this.quantity;

}

}

public class Coffee {

private CoffeeSelection selection;

private double quantity;

public Coffee(CoffeeSelection, double quantity) {

this.selection = selection;

this. quantity = quantity;

}

}

Using the *CoffeeMachine* abstraction

Using the *CoffeeMachine* class is almost as easy as making your morning coffee. You just need to prepare a *Map*of the available *CoffeeBean*s, instantiate a new *CoffeeMachine*object, and call the *brewCoffee*method with your preferred *CoffeeSelection*.

import org.thoughts.on.java.coffee.CoffeeException;

import java.util.HashMap;

import java.util.Map;

public class CoffeeApp {

public static void main(String[] args) {

// create a Map of available coffee beans

Map<CoffeeSelection, CoffeeBean> beans = new HashMap<CoffeeSelection, CoffeeBean>();

beans.put(CoffeeSelection.ESPRESSO,

new CoffeeBean("My favorite espresso bean", 1000));

beans.put(CoffeeSelection.FILTER\_COFFEE,

new CoffeeBean("My favorite filter coffee bean", 1000));

// get a new CoffeeMachine object

CoffeeMachine machine = new CoffeeMachine(beans);

// brew a fresh coffee

try {

Coffee espresso = machine.brewCoffee(CoffeeSelection.ESPRESSO);

} catch(CoffeeException e) {

e.printStackTrace();

}

} // end main

} // end CoffeeApp