



# OOP Concept for Beginners: What is Abstraction?

THORBEN JANSSEN | NOVEMBER 23, 2017 |

[DEVELOPER TIPS, TRICKS & RESOURCES \(HTTPS://STACKIFY.COM/DEVELOPERS/\)](https://stackify.com/developers/)



— .NET

Java

Azu

AW

Clo

Pop


APM



Sea  
ASP.NET

Topic

ASP

:  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. You can find it everywhere in the real world.

## 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.



:  You 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.



([https://cta-service-cms2.hubspot.com/ctas/v2/public/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-abstraction%2F&redirect\\_url=APefjpGDS38eWX\\_Snj\\_BM9dIELfpaknnCAUJ5RBs6yHA0zrKgBU55i9a9OzBrPwP9653Xs8s8luUPNWp-18-n3K-JSISjPykZP5c2UFIqNV5Y17fvMUzMzAFtqf1O5qsxSiL6JvNTwXH\\_1ZTVxSV2RnxYOQ&click=28da3e96-b48f-4222-](https://cta-service-cms2.hubspot.com/ctas/v2/public/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-abstraction%2F&redirect_url=APefjpGDS38eWX_Snj_BM9dIELfpaknnCAUJ5RBs6yHA0zrKgBU55i9a9OzBrPwP9653Xs8s8luUPNWp-18-n3K-JSISjPykZP5c2UFIqNV5Y17fvMUzMzAFtqf1O5qsxSiL6JvNTwXH_1ZTVxSV2RnxYOQ&click=28da3e96-b48f-4222-)



## 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> (<https://github.com/thjanssen/Stackify-OopAbstraction>).



Stackify

```
import org.thoughts.on.java.coffee.CoffeeException;
import java.util.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;
    }
}
```

*CoffeeSelection* 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.



NE

Java

Azu

AW

Clo

Pop



Topic



```
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 *CoffeeBeans*, instantiate a new *CoffeeMachine* object, and call the *brewCoffee* method with your preferred *CoffeeSelection*.



.NET

Java

Azul

AW

Cloj

Pop





Stackify

```
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
```



NE

Java

Azu

AW

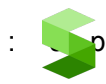
Clo

Pop



Topic





You can see in this example that the abstraction provided by the *CoffeeMachine* class hides all the details of the brewing process. That makes it easy to use and allows each developer to focus on a specific class.

If you implement the *CoffeeMachine*, you don't need to worry about any external tasks, like providing cups, accepting orders or serving the coffee. Someone else will work on that. Your job is to create a *CoffeeMachine* that makes good coffee.

And if you implement a client that uses the *CoffeeMachine*, you don't need to know anything about its internal processes. Someone else already implemented it so that you can rely on its abstraction to use it within your application or system.

That makes the implementation of a complex application a lot easier. And this concept is not limited to the public methods of your class. Each system, component, class, and method provides a different level of abstraction. You can use that on all levels of your system to implement software that's highly reusable and easy to understand.

## Not limited to the client API

Let's dive a little bit deeper into the coffee machine project and take a look at the constructor method of the *CoffeeMachine* class.



.NET

Java

Azu

AW

Clo

Pop



Topic







```
import java.util.Map;

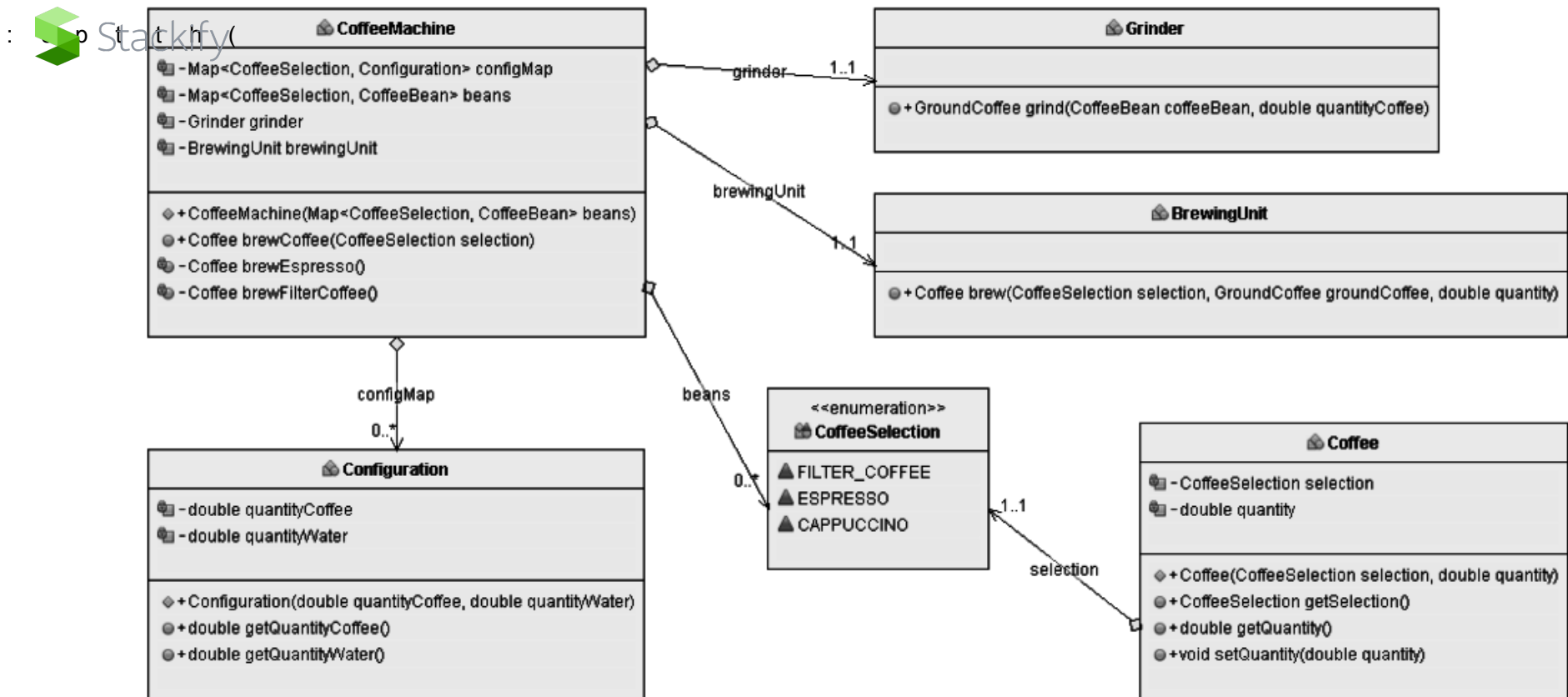
public class CoffeeMachine {
    private Map<CoffeeSelection, Configuration> configMap;
    private Map<CoffeeSelection, CoffeeBean> beans;
    private Grinder grinder;
    private BrewingUnit brewingUnit;

    public CoffeeMachine(Map<CoffeeSelection, CoffeeBean> beans) {
        this.beans = beans;
        this.grinder = new Grinder();
        this.brewingUnit = new BrewingUnit();

        // create coffee configuration
        this.configMap = new HashMap<CoffeeSelection, Configuration>();
        this.configMap.put(CoffeeSelection.ESPRESSO, new Configuration(8, 28));
        this.configMap.put(CoffeeSelection.FILTER_COFFEE, new Configuration(30, 480));
    }
}
```

As you can see in the code snippet, the constructor not only stores the provided *Map* of available *CoffeeBeans* in an internal property, it also initializes an internal *Map* that stores the configuration required to brew the different kinds of coffees and instantiates a *Grinder* and a *BrewingUnit* object.

[.NET](#)[Java](#)[Azul](#)[AW](#)[Cloj](#)[Pop](#)[Topic](#)



All these steps are not visible to the caller of the constructor method. The developer most likely doesn't even know that the *Grinder* or *BrewingUnit* class exists. That's another example of the abstraction that the *CoffeeMachine* class provides.

## Each class provides its own abstraction

The classes *Grinder* and *BrewingUnit* provide abstractions on their own. The *Grinder* abstracts the complexity of grinding the coffee and *BrewingUnit* hides the details of the brewing process.

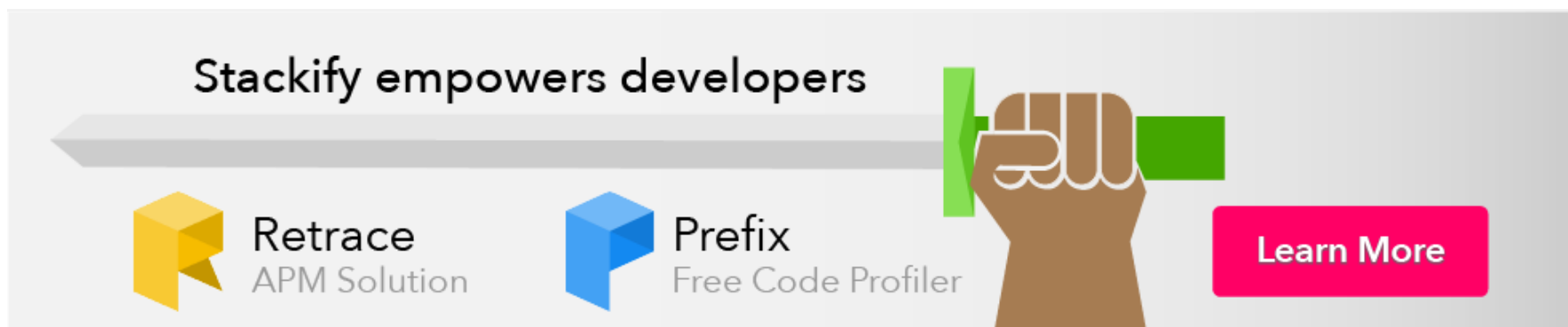
Stackify

```
public class Grinder {  
    public GroundCoffee grind(CoffeeBean coffeeBean, double quantityCoffee) {  
        // ...  
    }  
}
```

```
public class BrewingUnit {  
    public Coffee brew(CoffeeSelection selection, GroundCoffee groundCoffee, double quantity) {  
        // ...  
    }  
}
```

That makes the implementation of the *CoffeeMachine* class a lot easier. You can implement the *brewCoffee* method without knowing any details about the grinding or brewing process. You just need to know how to instantiate the 2 classes and call the *grind* and *brew* methods.

Stackify empowers developers



The banner features a central illustration of a brown fist holding a green rectangular block. To the left of the fist is a long, light gray arrow pointing left. Below the fist are two logos: a yellow cube logo for 'Retrace APM Solution' and a blue cube logo for 'Prefix Free Code Profiler'. To the right of the fist is a pink button with the text 'Learn More'.

Retrace APM Solution

Prefix Free Code Profiler

Learn More

([https://cta-service-cms2.hubspot.com/ctas/v2/public/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%2Ffoop-concept-](https://cta-service-cms2.hubspot.com/ctas/v2/public/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%2Ffoop-concept-)

## Different abstraction levels within the same class

In this example, I took the abstraction one step further and implemented 3 methods to brew the different kinds of coffee. The *brewCoffee* method, which gets called by the client, just evaluates the provided *CoffeeSelection* and calls another method that brews the specified kind of coffee.

The *brewFilterCoffee* and *brewEspresso* methods abstract the specific operations required to brew the coffee.

```
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());  
}
```

```
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());  
}
```

I defined both methods as private because I just want to provide an additional, internal level of abstraction. That not only makes the implementation of the *brewCoffee* method a lot easier, it also improves the reusability of the code.

You could, for example, reuse the *brewEspresso* method when you want to support the *CoffeeSelection.CAPPUCCINO*. You would then just need to implement the required operations to heat the milk, call the *brewEspresso* method to get an espresso, and add it to the milk.

## Summary



.NET

Java

Azu

AW

Cloj

Pop




Topic



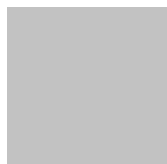


Abstraction is a general concept which you can find in the real world as well as in OOP languages. Any objects in the real world, like your coffee machine, or classes in your current software project, that hide internal details provide an abstraction.

These abstractions make it a lot easier to handle complexity by splitting them into smaller parts. In the best case, you can use them without understanding how they provide the functionality. And that not only helps you to split the complexity of your next software project into manageable parts, it also enables you every morning to brew a fresh cup of amazing coffee while you're still half asleep.

 About the Author

 Latest Posts



### About Thorben Janssen

Thorben is an independent trainer and author of the Amazon bestselling book *Hibernate Tips - More than 70 solutions to common Hibernate problems*. He writes about Java EE related topics on his blog [Thoughts on Java](https://www.thoughts-on-java.org) (<https://www.thoughts-on-java.org>).



(<https://plus.google.com/u/0/b/109323639830862412239/109323639830862412239>)



(<http://thjanssen123>)



(<https://www.linkedin.com/in/thorbenjanssen/>)



(<https://plus.google.com/u/0/b/109323639830862412239/109323639830862412239?rel=author>)

Improve Your Code with Retrace APM



.NET

Java

Azu

AW

Clo

Pop



Stackify's APM tools are used by thousands of .NET, Java, and PHP developers all over the world.  
Explore Retrace's product features to learn more.



(/retrace-application-performance-management/)

App Performance Monitoring (<https://stackify.com/retrace-application-performance-management/>)



(/retrace-code-profiling/)

Code Profiling (<https://stackify.com/retrace-code-profiling/>)



.NET

Java

Azu

AW

Clo

Pop





(/retrace-error-monitoring/)

Error Tracking (<https://stackify.com/retrace-error-monitoring/>)



(/retrace-log-management/)

Centralized Logging (<https://stackify.com/retrace-log-management/>)



(/retrace-app-metrics/)

App & Server Metrics (<https://stackify.com/retrace-app-metrics/>)





## Get In Touch

[Contact Us](#)

[Request a Demo](#)

8900 State Line Rd  
#100  
Leawood, KS 66206

[816-888-5055](tel:816-888-5055)



## Product

[Retrace APM](#)

[Prefix](#)

[.NET Monitoring](#)

[Java Monitoring](#)

[PHP Monitoring](#)

[Node.js Monitoring](#)

[Ruby Monitoring](#)

[Retrace vs New Relic](#)

## Solutions

[Log Management](#)

[Application  
Performance  
Management](#)

[Application Metrics](#)

[Azure Monitoring](#)

[Error Tracking](#)

[Code Profiling](#)

## Resources

[Pricing](#)

[Case Studies](#)

[Blog](#)

[Podcast](#)

[Documentation](#)

[Free eBooks](#)

[Videos](#)

[Ideas Portal](#)

## Company

[About Us](#)

[GDPR](#)

[Careers](#)

[Security Information](#)

[Terms & Conditions](#)

[Privacy Policy](#)

[Retrace vs Application Insights](#)

\_\_\_\_\_

\_\_\_\_\_

\_\_\_\_\_

[What is APM?](#)

\_\_\_\_\_

\_\_\_\_\_

\_\_\_\_\_

[APM ROI](#)

\_\_\_\_\_

\_\_\_\_\_

© 2018 Stackify



[.NET](#)

[Java](#)

[Azure](#)

[AWS](#)

[Cloud](#)

Pop

