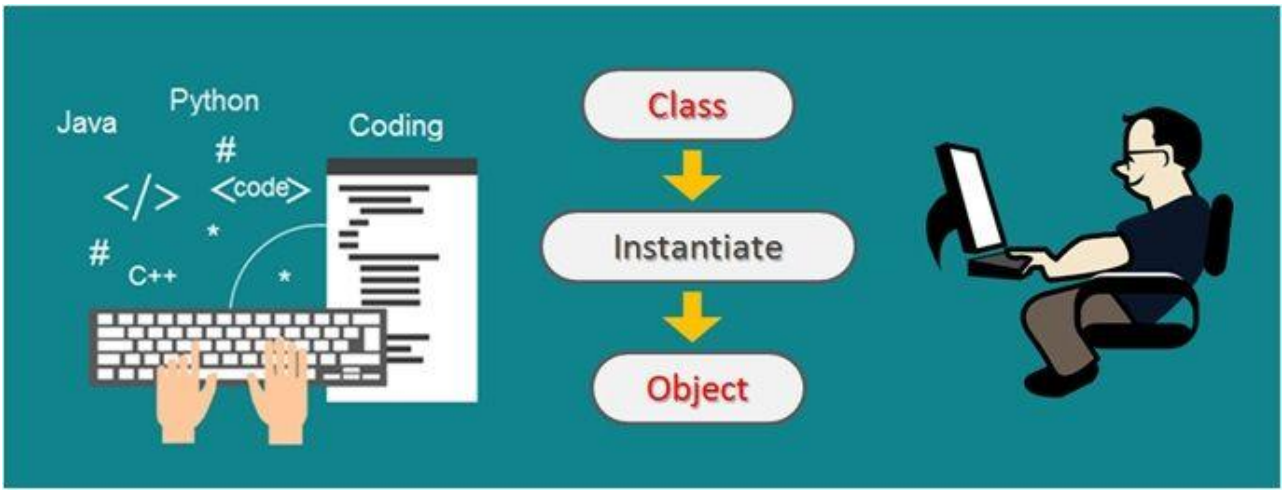
Week 04 - Classes, Objects, Constructors and Encapsulation

Louis Botha, `louis.botha@tuni.fi`



Object Oriented Programming | OOP Principles Explained With Example.

Source: Head First Java, 3rd Edition
Source: https://www.geeksforgeeks.org/classes-objects-java/?ref=lbp
Source: https://www.w3schools.com/java/java_oop.asp

Classes and Objects

1. What is an object?
   An object is a software bundle of related state and behavior. It is a self-contained unit of functionality that can be manipulated and used to represent real-world entities. For example, an object can be a car, a bank account, or a person.

2. What is a class?
   A class is a *blueprint* or template for creating objects. It defines the properties and methods that an object of that class will have. A class in not an object, but a class knows how to create objects. It tells the JVM *how* to make an object of that particular type.

For example, a class called "Car" may have properties like make, model, and year, and methods like start, stop, and accelerate.

3. Creating a class in Java
   To create a class in Java, we use the "class" keyword, followed by the name of the class. The properties and methods of the class are defined within the curly braces {}.

   For example:

```
class Car {
    // properties go here
    // methods go here
}
```

4. Adding properties to a class
   Properties, also known as instance variables, are used to store the state of an object. They are defined within the class and have an access level such as private, protected, and public.

   For example:

```
class Car {
    private String make;
    private String model;
    private int year;
}
```

5. Adding methods to a class
   Methods are used to define the behavior of an object. They are also defined within the class and have an access level.

   For example:

```
class Car {
    // properties
    // methods
```

```
    public void start() {
        // code to start the car
    }
    public void stop() {
        // code to stop the car
    }
    public void accelerate() {
        // code to accelerate the car
    }
}
```

6. Creating an object from a class
   To create an object from a class, we use the "new" keyword, followed by the name of the class, and assign it to a variable.

   For example:

```
Car myCar = new Car();
```

   When an object of a class is created, the class is said to be **instantiated**.

Example

Here we have the *blueprint* of a `Dog` class.

```
class Dog {
    // Properties (Attributes)
    String breed;
    String name;
    int age;

    // Methods
    public void bark() {
        System.out.println("Woof, Woof");
```

```
    }
    public void run() {
        System.out.println("Run, Run");
    }
}
```

Here in the Main the class, we create a `Dog Object` from the `Dog Class` *blueprint*.

```
// Application entry point or
class Main {
    public static void main(String[] args) {
        Dog myDog = new Dog();    // Make a dog object
        // Use the Dot operator to set the properties of the o
bject
        myDog.breed = "Golden Retriever";  // Set the breed of
the dog
        myDog.name = "Alma";      // Set the breed of the dog
        myDog.age = 12;           // Set the age of the dog

        // Call the methods of the object
        myDog.bark();
        myDog.run();
    }
}
```

Practice

1. Create a class called "BankAccount" that has properties for
    - account number,
    - account holder name, and
    - balance.

Add methods to

- deposit,
- withdraw, and
- check the balance of the account.

Create an object of the BankAccount class and test the methods.

All the instances share the properties and the behavior of the class. But the values of those properties, i.e. the state are unique for each object.

A single class may have any number of object instances. Say we have 2 `Dog` objects.

```
// Application entry point or a class for testing our actual c
lass
class Main {
    public static void main(String[] args) {
        Dog myDog = new Dog();    // Make a dog object called
myDog
        myDog.breed = "Golden Retriever";
        myDog.name = "Alma";
        myDog.age = 12;
```

```
        myDog.bark();
        myDog.run();

        Dog myOtherDog = new Dog(); // Make a dog object calle
d myOtherDog
        myOtherDog.breed = "French Waterdog";
        myOtherDog.name = "Viima";
        myOtherDog.age = 4;
        myOtherDog.bark();
        myOtherDog.run();
    }
}
```

Here we change the `Main` class to create another `Dog` object with its own values. Both objects are dogs.

:protip: These `Dog` objects can also be added to an array `Dog[] = new Dog[2]` of `Dog` objects.

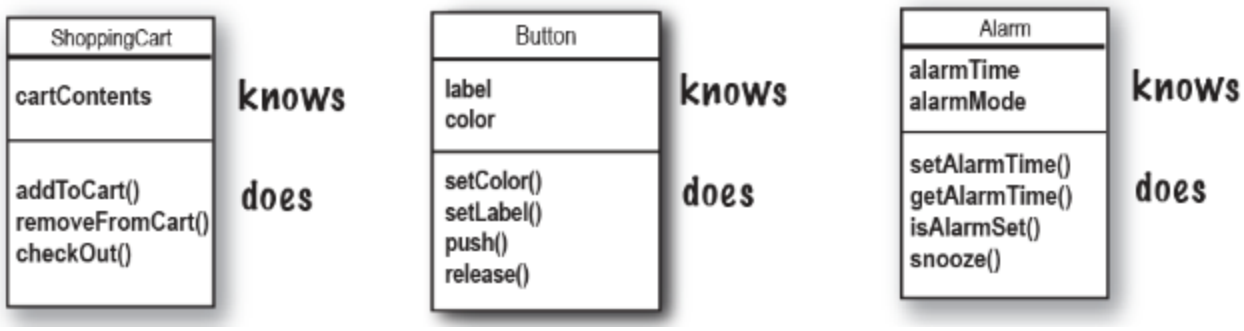
```
Dog[] myDogs = new Dog[2];

myDogs[0] = myDog;
myDogs[1] = myOtherDog;

for(int i=0; i<2; i++) {
    myDogs[i].bark();
}
```

Designing a Class

When you design a class, think about the objects that will be created from that class type.

Think about:

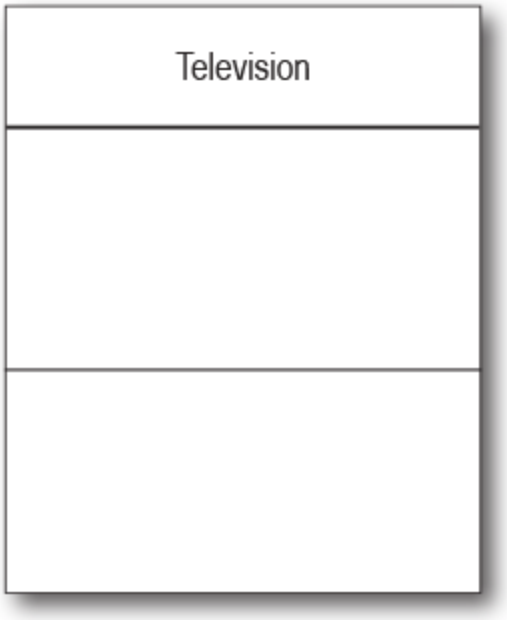
- things the object knows
- things the object does



🔨 Some Practice

Think about a Television.

If you design a class that would need to represent a television, what would it look like? You don't have to write code but think what does the a `Television` object need to know and what does the `Television` object need to be able to do.



In summary:

- All Java code is defined in a **class**.
- A class describes how to make an object of that class type. **A class is like a blueprint.**
- An object **knows** things and **does** things.
- Things an object knows about itself are called **properties/fields** (effectively, it's just variables). They represent the *state* of an object.
- Things an object does are called **methods**. They represent the *behavior* of an object.

Low-effort meme - 9GAG

Constructors

A *constructor* is a block of code that is called when an instance of an object is created.

In many ways, a constructor is similar to a method, but a few differences exist:

- A constructor doesn't have a return type.
- The name of the constructor must be the same as the name of the class.
- Unlike methods, constructors are not considered to be members of a class (That's important only when it comes to inheritance),
- A constructor is called when a new instance of an object is created. In fact, it's the `new` keyword that calls the constructor.
- Used to give an initial state to the object when it is created.
- After creating the object, you can't call the constructor again.

A class can have *multiple* constructors, but each one of them should have a different set of parameters. The different set of parameters is called constructor overloading.

Each constructor can perform different initialisations based on the passed parameters.

Here is an example of a class with a constructor:

```
class Student {
    private String name;
    private int id;
    private String major;

    public Student(String name, int id, String major) {
        this.name = name;
        this.id = id;
        this.major = major;
    }
}
```

In this example, the class "Student" has a constructor that takes three parameters:

- a string for the name,
- an int for the id, and
- a string for the major.

The constructor uses the `this` keyword to refer to the instance variables of the class, and assigns the passed parameter values to them.

You can also have a constructor with no parameters, this is called a default constructor, and it's used to create an object with default values for the properties.

Every class has a right to a constructor. If you don’t provide a constructor, Java appoints one for you, free of charge. This free constructor is called the *default constructor.* It doesn’t accept any parameters and doesn’t do anything, but it does allow your class to be instantiated.

```
class Student {
    private String name;
    private int id;
```

```
    private String major;

    public Student() {
        this.name = "";
        this.id = 0;
        this.major = "";
    }
}
```

When you create a new instance of the class, you use the new keyword and call the constructor, like this:

```
Student student1 = new Student("John Doe", 123456, "Computer Science");
Student student2 = new Student();
```

Here we have a new blueprint of a `Dog` class where the class has a constructor.

```
class Dog {
    String breed;
    String name;
    int age;

    // The constructer
    Dog(String breed, String name, int age) {
        this.breed = breed;
        this.name = name;
        this.age = age;
    }

    public void bark() {
        System.out.println("Woof, I am " + this.name);
    }
```

```
    public void run() {
        System.out.println("Run, Run");
    }
}
```

Now we can change the `Main` class to create the objects and setting the variable when calling the constructor.

```
class Main {
    public static void main(String[] args) {
        Dog myDog = new Dog("Golden Retriever", "Alma", 12);
        myDog.bark();
        myDog.run();
        Dog myOtherDog;
        myOtherDog = new Dog("French Waterdog", "Viima", 4);
        myOtherDog.bark();
        myOtherDog.run();
    }
}
```

Like normal methods, the constructor can also be overloaded. In other words, we can create the `Dog` object when we know only some of the

```
class Dog {
    private String breed;
    private String name;
    private int age;

    Dog(String breed, String name, int age) {
        this.breed = breed;
        this.name = name;
        this.age = age;
```

```
    }

    Dog(String name, int age) {
        this.name = name;
        this.age = age;
    }

    Dog(String breed) {
        this.breed = breed;
    }
}
```

Encapsulation

Encapsulation is one of the four fundamental principles of object-oriented programming, along with inheritance, polymorphism, and abstraction. It refers to the practice of hiding the implementation details of an object and exposing only the necessary information to the outside world.

Encapsulation allows you to control the access to the internal state of an object, you can use it to limit the ways in which other objects can interact with it. The main benefits of encapsulation are:

- Data hiding
  Encapsulation allows you to hide the implementation details of a class, such as the private fields and methods, and expose only the necessary information to the outside world. This helps to protect the internal state of the object and prevents other objects from modifying it in an unintended way.
- Improved maintainability
  Encapsulation makes it easier to modify the implementation of a class without affecting the code that uses it. This allows you to make changes to the class without breaking existing code.
- Modularity

Encapsulation allows you to create self-contained objects that can be reused in other parts of your code.

- Abstraction
  Encapsulation allows you to create abstractions of the real-world entities and operations.

In Java, encapsulation is achieved by using the access modifiers (public, private, protected) to control the access to the fields and methods of a class. For example, you can use the private access modifier to make a field or method only accessible within the class, and the public access modifier to make it accessible to other classes.

It's worth noting that encapsulation is not the same thing as data protection, encapsulation provides a way to *hide the implementation details* and control the access to the class's internal state, while data protection is the technique of secure the data from *unauthorised access*.

Getters and Setters

Encapsulation – About coding

As a general rule, you should avoid creating public properties. Instead, you can make all your properties private. Then you can selectively grant access to the data those properties contain by adding to the class special methods called ***accessors.***

There are two types of accessors.

- A `get` *accessor* (also called a ***getter***) is a method that retrieves an attribute value,
- A `set` *accessor (*also called a ***setter****)* is a method that sets an attribute value.

Adding getters and setters and making sure that *sensitive* data is hidden from users, is called **Encapsulation**.

Encapsulation in short, is to:

- declare class variables/properties as `private`
- provide public **get** and **set** methods to access and update the value of a `private` variable

A good practice is to validate the parameter that are passed into the setter methods. Here is an example of a class with getters and setters and validation in the set methods.

```
class Student {
    private int id;
    private String name;
    private String major;

    public void setId(int id) {
        if (id > 0) {
            this.id = id;
        } else {
            throw new IllegalArgumentException("Invalid id val
ue.");
        }
    }
```

```
    public int getId() {
        return id;
    }

    public void setName(String name) {
        if (name != null && !name.isEmpty()) {
            this.name = name;
        } else {
            throw new IllegalArgumentException("Invalid name v
alue.");
        }
    }

    public String getName() {
        return name;
    }

    public void setMajor(String major) {
        if (major != null && !major.isEmpty()) {
            this.major = major;
        } else {
            throw new IllegalArgumentException("Invalid major
value.");
        }
    }

    public String getMajor() {
        return major;
    }
}
```

In this example, for each property, there is a corresponding get method that returns the value of the property.
The get methods are public and does not take any parameter.
This way, other objects can access the properties of a Student object by calling the corresponding get method.

In this example, the `main` method creates an object of the `Student` class and sets its properties using the set methods. Then, it prints the properties of the student object by calling the corresponding get methods.

```
public class Main {
    public static void main(String[] args) {
        Student student = new Student();
        student.setId(123);
        student.setName("John Doe");
        student.setMajor("Computer Science");

        System.out.println("Student ID: " + student.getId());
        System.out.println("Student Name: " + student.getName
());
        System.out.println("Student Major: " + student.getMajo
r());
    }
}
```

It's a good practice to include try-catch blocks around the code that can throw an exception, this way you can handle it in a controlled way, and prevent the program from crashing.
In case of the set methods of the `Student` class, you can use a try-catch block to handle the `IllegalArgumentException` that is thrown when the validation fails.

Here's an example of how you can use a try-catch block to handle the exception in the `main` method:

```
public class Main {
```

```
    public static void main(String[] args) {
        Student student = new Student();
        try {
            student.setId(-123);
            student.setName("");
            student.setMajor(null);
        } catch (IllegalArgumentException e) {
            System.out.println("Invalid value: " + e.getMessag
e());
        }
    }
}
```

In this case, if any of the set methods throws an exception, the catch block will catch it and print the message of the exception.
This way you can handle the exception, and give a proper feedback to the user, instead of letting the program to crash.

:protip: Some notes:

- You can have properties that are never set from outside the class which are **read-only** and only provide a getter.
- You can have getters that return a calculated value.