Object-oriented programming

Java is an object-oriented language. Except for primitive types *everything* in Java is an object.

Basic concepts:

- Class
  A rule for an object. It is a template (pattern) according to which you can create various objects. It defines a new data type.
- Object
  An instance, an item, a specific example of the class. A programming way of presenting an entity.
- Field
  A feature of a given object describing a given property of an object in terms of its type.
- Method
  An operation performed on the class object.

- The body of the class is enclosed in braces { i }.
- These are fields (status, data) and methods (operations, behavior).
- Java does not allow you to create fields and methods outside of the class!
- The class field can have a default value.
- If you do not specify a default value, it will be set by a compiler.
- Class names are capitalized; field and method names are lowercase.

We have used and created classes before. In the chapter on [data types](../../e-coursebook/java_basicsy/data_types_variables_and_constants) we created instances of simple type wrappers. Now we will learn how to create our own classes that will have fields and provide methods.

** An example of a 'Car' class for describing a car.**

```
public class Car {
```

```
    private String model;
    private int productionYear;
    private String color;
    private long distance = 0;
    private boolean used = false;

    public Car() {

    }

    public String getModel() {
        return model;
    }

    public void setModel(String model) {
        this.model = model;
    }

    public void drive(int km) {
        distance += km;
    }
}
```

- `Car`class declaration; `{` bracket declares the beginning of the class body
- class fields (attributes)
- class constructor declaration; pay attention to the name consistent with the class name and the lack of a return type declaration
- a method (the so-called *getter*) that provides a field value; in this case, the `model` field
- a method (so-called *setter*) that allows to set the value of a field; in this case, the `model` field
- a method that performs an operation on an object (changes the state of the object); by calling it, we pass the number of kilometers to travel, and the method changes the value of the `distance` field by a given value
- `}` bracket ending the body of class `Car`

Classes and objects

The key is to understand the difference between a class and an object. For the record:

Class is a template, formula, recipe for an object. Definition of a new data type. The class has features (fields) and defines methods (activities). ** Object ** is an **instance** of the class.

This is best shown with examples. Classes can be e.g. Car, Human, Ticket, Movie

etc.

- Car
  When we think (of a class) `Car`, we do not mean a *specific* car, but only a vehicle that has the characteristics of a car, e.g., model, brand, registration number, year of manufacture, color, number of doors, body type, mileage etc. Additionally, the car has a behavior, such as moving (forward, backward), opening doors, windows, playing music, and closing and opening. What can the 'Car' class objects be like? Objects are all specific cars that we can clearly identify. It can be, for example, a red Toyota Yaris from 2016 with the registration number GD 98744, a silver Mercedes CLK from 2018 with the registration number WE 66655.
- Human
  When I say: "I see a man", you do not know exactly which person I see. All we know is that it is a creature with human characteristics. These *characteristics* define a person and at the same time distinguish individuals from one another. Thus, the 'Human' class can define such characteristics as: date of birth, first name, surname, address, eye color, hair color, weight, height or gender. Only * giving * these features can clearly indicate a specific person. Any person is an instance of the 'Human' class, eg each of us.
- Film
  In order to clearly indicate a specific film, you can give its title, director, year of production, etc. Examples (objects of the "Film" class) may be, for example, "The Matrix" by the Wachowski brothers from 1999 or "Terminator" by James Cameron from 1984.

We, as programmers, define what features of the objects we want to store and define them in classes – this is called *abstraction*.

When do we create classes?

If several variables often appear close to each other, if several variables together constitute a whole, it is worth grouping them and creating (naming) a class. For example, if we work with *first name*, *surname* and *social security number*, it is worth grouping these data into the class `Human` or `Person` and store this data under one type.

When, for example, we work with geometric figures, instead of storing a lot of variables – it is worth creating classes with appropriate fields and methods in them. E.g:

- Wheel
  We have a variable `r` holding the radius. We can create a `Wheel` class with an `r` field and the `getField()` and `getPerimeter()` methods.
- Square
  We have a variable `a` that stores the length of the side. We can

We have a variable `a` that stores the length of the side. We can create a class `Square` with a field `a` and methods `getField()` and `getPerimeter()`

- Rectangle
  We have variables `a` and `b` that store the side lengths. We can create a class `Rectangle` with a field `a` and methods `getField()` and `getPerimeter()`.
- Triangle
  We have variables `a` and `h` which store the length of the side and the height (lowered to that side) respectively. We can create a class `Triangle` with the `a` and `h` fields and the`getField()`method.

Object creation – constructors

A constructor is a special method that creates an instance of a class, or an object. In the example with the `Car` class, the constructor was defined like this:

```
    public Car() {

    }
```

It is a method with **the same name as class**, which **does not declare the return type**! At first glance, it differs from the other methods. We call this constructor *argumentless*. The constructor is a good place to initialize the state of an object.

Defining constructors

Let us consider a class representing a dice that only stores the result of the dice roll. Let's call her 'Dice'.

```
public class Dice {

    private int value;
}
```

If we define a constructor (no arguments) like below

```
public class Dice {

    private int value;

    public Dice() {
```

```
    }
}
```

then it will be possible to create an object of class `Dice`, which (once created) will be in an" undefined "state. Therefore, constructors are often defined in such a way that they allow setting all **required**, **important**, **unambiguous** properties. For the 'Dice' class, it might look like this

```
public class Dice {

    private int value;

    public Dice(int value) {
        this.value = value;
    }
}
```

- the constructor takes a `value` parameter
- we assign the value of the variable `value` to a class field named `value`

> ### Important
>
> The notation `this.value` means that we point to the **field of this** class named `value`.

> ### Note
>
> The `public` keyword before a constructor name means that it is "public", "accessible from everywhere". The access levels (the `public` and `private` modifiers) are beyond the scope of this manual.

Now, when creating an instance of the `Dice` class, we **need to** provide a value (result). This prevents us from creating a cube object with an undefined result.

When designing the constructor, think about what fields should be set right after creating the!

> ### Note

> A class can have several constructors, but they must differ in the number (and / or type) of parameters.

Calling the constructors

Constructor is a special method. To call the constructor, we use the `new` keyword. We have already created instances of classes provided by Java

```
String text = new String("Hello, World!");
Integer i = new Integer(15);
Character c = new Character('a');
```

To create an object (instance) of the `Dice` class, just type

```
Dice dice = new Dice(6);
```

In the `dice` variable we store a reference to an object of the `Dice` class. Additionally (in the constructor) we set the value of the `value` field to `6`. Now we can call methods on the variable `dice`.

Fields and methods

Now let's go back to the fields (attributes, state) and methods (actions) of classes. We will define the (mentioned) class describing the triangle – `Triangle`.

```
public class Triangle {

    private int a;
    private int b;
    private int c;

    public Triangle(int a, int b, int c) {
        this.a = a;
        this.b = b;
        this.c = c;
    }
}
```

We have defined three variables (class fields) of the type `int` that store the lengths of the sides of the triangle.

> **Note**

> Class fields will usually be prefixed with the `private` keyword (modifier). Access levels are beyond the scope of this Coursebook, but let me just mention that the point is that the class members are not directly "visible" to the outside, because we would then have no control over who reads them and when and – most of all – modifies!

When creating a new data type, we declare its components (attributes) that will characterize it. These are the * fields of the * class in which we store the state of the objects.

We can now create several instances of the `Triangle` class.

```
Triangle equilateral = new Triangle(6, 6, 6);   //
equilateral
Triangle isosceles = new Triangle(6, 6, 4);     //
isosceles
Triangle pythagorean = new Triangle(3, 4, 5);   //
pythagorean
Triangle triangle = new Triangle(1, 3, 10);     // a
triangle for sure?
```

Class fields are used to store the state of objects. Methods for performing operations on an object.

We can define a method that checks whether the given values are correct, i.e. whether the given side lengths can form a triangle.

```
    public boolean isValid() {
        if ((a + b) > c && (a + c) > b && (b + c) > a)
{
            return true;
        } else {
            return false;
        }
    }
```

A triangle has the property that the sum of any two sides must be greater than the remaining side.

Let's call this method on the created objects. We will create a new class with a `main` method in which we will create objects and call methods.

```
public class TriangleApp {

    public static void main(String[] args) {
        Triangle equilateral = new Triangle(6, 6, 6);
        Triangle isosceles = new Triangle(6, 6, 4);
        Triangle pythagorean = new Triangle(3, 4, 5);
        Triangle triangle = new Triangle(1, 3, 10);

        System.out.println(equilateral.isValid());
        System.out.println(isosceles.isValid());
        System.out.println(pythagorean.isValid());
        System.out.println(triangle.isValid());
    }
}
```

Result

```
true
true
true
false
```

So in the variable `triangle` we are storing an object of class `Triangle`, which is "degenerate", invalid. We will continue to use only the remaining, valid objects.

> **Note**
>
> We do not cover the exception mechanism in this coursebook, which would be useful in this case to prevent the creation of an invalid object. I'm just mentioning it so you know that Java is prepared for this type of situation.

Now let's define another methods for calculating the area and perimeter of the triangle.

```
    public double getField() {
        // Heron's formula
        double p = (a + b + c) / 2;
        return Math.sqrt(p * (p - a) * (p - b) * (p -
c));
    }

    public int getPerimeter() {
```

```
    public int getPerimeter() {
        return a + b + c;
    }
```

By running these methods we get

```
18
16
12
```

Getters and setters

So far, we can define fields, methods, constructors and create objects. There are also special methods, which we call `getter` (*accessor*) and `setter` (*mutator*).

When we define a class field, we can also create a method *to return* the value of that field and/or a method * to *set* the value of that field. The first method is `getter` and the second is `setter`.

Getters

For example, if we wanted to get and display the lengths of the sides of a triangle, then `getter` methods would be useful.

Methods of type `getter` are named by adding "**get**"to the field name, and the first letter of the field name is capitalized. The method returns the value stored in the field, and the return type matches the field type. Methods of the type `getter` are *argumentless*.

For example, for a field

```
private String name;
```

a `getter` method looks as follows

```
public String getName() {
    return name;
}
```

> **Note**
>
> It has been assumed that for `boolean` fields, in the `getter` methods, the prefix "**" is "**" is added

the "getter" methods, the prefix "is" is added instead of `get`.

```
private boolean active;

public boolean isActive() {
    return active;
}
```

** An example of `getter` methods for the `Triangle` class.**

```
public int getA() {
    return a;
}

public int getB() {
    return b;
}

public int getC() {
    return c;
}
```

Let's use the following methods:

```
Triangle pythagorean = new Triangle(3, 4, 5);

System.out.println("side  A: " + pythagorean.getA());
System.out.println("side B: " + pythagorean.getB());
System.out.println("side C: " + pythagorean.getC());
System.out.println("correct: " + (pythagorean.isValid()
? "yes" : "no"));
if (pythagorean.isValid()) {
    System.out.println("field: " +
pythagorean.getField());
    System.out.println("perimeter: " +
pythagorean.getPerimeter());
}

Triangle triangle = new Triangle(1, 3, 10);

System.out.println("side A: " + triangle.getA());
System.out.println("side B: " + triangle.getB());
System.out.println("side C: " + triangle.getC());
```

```
System.out.println("correct: " + (triangle.isValid() ?
"yes" : "no"));
if (triangle.isValid()) {
    System.out.println("field: " +
triangle.getField());
    System.out.println("perimeter: " +
triangle.getPerimeter());
}
```

Result:

```
Side  A: 3
side B: 4
side C: 5
correct: yes
field: 6.0
perimeter: 12
side A: 1
side B: 3
side C: 10
correct: no
```

Setters

For example, if we wanted to change the values of the side lengths of a triangle, then `setter` methods would be useful.

Methods of type `setter` are named by adding the prefix "**set**"to the field name, and the first letter of the field name is capitalized. The method returns the type of `void`, and its (only) argument type matches the type of the field.

For example, for a field

```
private String name;
```

A `setter` method looks as follows

```
public void setName(String name) {
    this.name = name;
}
```

> ## Note

> Again, I emphasize the difference between: `this.name` and `name`.
> `this.name` points to the `name` field of the current (`this`) class.
> `name` points to the name of the parameter of the `setName (String name)` method.
>
> This provision is used to prevent name collisions.

** An example of `setter` methods for the `Triangle` class.**

```
public void setA(int a) {
    this.a = a;
}

public void setB(int b) {
    this.b = b;
}

public void setC(int c) {
    this.c = c;
}
```

With these methods, we can "fix" the triangle defined in the variable `triangle`.

```
Triangle triangle = new Triangle(1, 3, 10);
System.out.println("correct: " + (triangle.isValid() ?
"yes" : "no"));
triangle.setA(8);
System.out.println("correct: " + (triangle.isValid() ?
"yes" : "no"));
```

Result

```
correct: no
correct: yes
```

> **Note**
>
> As a programmer, you decide which fields you want

> to make *getters* available to and which *setters*. If you don't want an outsider to change the value of the field – don't generate a *setter*. If you don't want an outsider to be able to read the value of a given field – don't generate a *getter*.

`toString()` method

The `toString()` method is another "special" method. Its task is *textual representation of the state of the object*. All relevant information should be included in this representation. We can call the `toString()` method explicitly (just like the other methods), but we often pass a variable to the `System.out.println()` method, which – on a reference variable – automatically calls its `toString() method`.

Coming back to the `Dice` class for a moment, you could define the following `toString()` method for it:

```
public class Dice {

    private int value;

    public Dice(int value) {
        this.value = value;
    }

    public String toString() {
        return "[" + value + "]";
    }
}
```

Creating several instances and calling `toString()`

```
public class DiceApp {

    public static void main(String[] args) {
        Dice one = new Dice(1);
        Dice six = new Dice(6);

        System.out.println(one.toString());
        System.out.println(one);
        System.out.println(six.toString());
        System.out.println(six);
    }
}
```

Result:

```
[1]
[1]
[6]
[6]
```

Note

Every class in Java has a `toString ()` method declared. What we did above is to * override * this default method. Try not to override the `toString ()` method in the `Dice` class and call it; something similar to

```
Dice@71318ec4
```

where:

- `Dice` – is the class name
- `71318ec4` – stands for `hash code` (numeric value) written in hexadecimal form

Overriding or overriding methods and the `hashCode ()` and `equals ()` methods are beyond the scope of this manual.

Summary

What have we learned in this chapter?

Q: What is a class?

A: It's a recipe for an object. A template, a pattern according to which we can create various objects.

Q: What is an object?

A: Instance, concrete example of a given class.

Q: What is the class field?

A: A feature of a given object describing the type of a given property of the object.

Q: What is the classroom method?

A: This is an operation performed on the class object.

Q: What's the naming convention for classes, methods and fields?

A: The names of the classes are written with a capital letter; methods and fields – small. If the name consists of several words, we write it together, but we start each word with a capital letter.

Q: When do we create classes?

A: When we want to name some entity; when several variables together constitute a whole.

Q: What is a constructor?

A: This is a special method that creates an instance of the class, ie an object. It has the same name as the class and does not declare a return type.

Q: What does the word `this` mean (in the context of constructors)?

A: You can often see `this.value = value` in constructors. This means that the `value` (constructor parameter) is assigned to the `value` variable (class field). To distinguish a constructor parameter name from a class field name, the word `this` is used to indicate that it is a **field of that** class.

Q: How do we call constructors in Java?

A: Constructors in Java are special methods that we call in a special way. The word 'new' is used for this. For example

```
String text = new String("Hello, World!");
Date today = new Date();
```

Q: What are getters?

A: Getters (*accessors*) are methods that return the value of a given field. They are created by appending the ** get ** prefix to the field name, and the first letter of the field name is capitalized. The method returns the value stored in the field and the return type matches the field type. Methods of the type `getter` are * argumentless *. It has been assumed that for `boolean` fields in the `getter` methods, the prefix ** is ** is added instead of `get`.

Q: What are setters?

Q: What are setters?

A: Setters (*mutators*) are methods that set the value of a given field. They are created by appending the "**set**" prefix to the field name, and the first letter of the field name is capitalized. The method returns the type of `void`, and its (only) argument type matches the type of the field.

Q: What is the `toString()` method?

A: This is a "special" method. Its task is *textual representation of the state of the object*. All relevant information should be included in this representation. Every class in Java has a `toString()` method declared.

Exercises

- Create a `Person` class
  - add fields: name, surname, gender, age, pesel
  - add a method to check whether a given person has reached retirement age (e.g. `hasReachedRetirementAge()`); for women, let's assume the retirement age is >=60 years, and for men >=65 years
  - add a method that returns the age difference from one person to another
    - let the method take a parameter of type `Person`
    - it should not return negative values as the difference of years
  - add a method that calculates and returns how many years are left before retirement
- Create a `Computer`class`public class Computer { // mandatory fields private String motherboard; // motherboard private String processor; // "i5", "i7", "intel", "amd" private int cores; // number of cores private int ram; // amount of RAM // additional fields private int hd; // disk size in GB private String monitor; // producer name private String printer; // printer name public Computer(String motherboard, String processor, int cores, int ram) { } }`
  - complete the constructor with value assignment
  - add a `toString()`method
  - add methods *getter* and *setter*
  - test how it works
  - the `processor` and `cores` fields merge with each other – let's create a separate class for them`public class Processor { private String name; // procesor name private int cores; // number of cores }`
    - add a proper constructor
    - add methods *getter* and *setter*

add methods getter and setter
        - add a `toString()` method
        - change the `Computer` class to use the `Processor` class now `public class Computer { // mandatory fields private String motherboard; // motherboard private Processor processor; private int ram; // amount of RAM // additional fields private int hd; // disk size in GB private String monitor; // producer name private String printer; // printer name public Computer(String motherboard, Processor processor, int ram) { } }`
    - RAM chips can come in various configurations, create a class `RAM` and put the appropriate fields in it (e.g. `name` and `size`)
        - suppose there can be 4 RAM bones on the board – create a 4-element array for elements of the `RAM` class`public class Computer { // mandatory fields private String motherboard; // motherboard private Processor processor; private RAM[] ramSlots; // RAM chips // additional fields private int hd; // disk size in GB private String monitor; // producer name private String printer; // printer name public Computer(String motherboard, Processor processor, Ram ram) { }`
        - in the constructor put a variable `Ram ram` that specifies a single RAM chip given when creating the computer; remember to initialize the `ramSlots` array
        - add a method that allows you to add more RAM dice – watch out for slot restrictions; in case of insufficient space, you can e.g. remove the smallest bone and replace it with the given one (if it is not smaller)
    - on the basis of RAM, perform a similar change with hard disks – there may also be several of them in the computer; have a different capacity and manufacturer (and e.g. type and speed)
    - other variables (`monitor`, `printer`) can also be converted into classes – a computer can have several monitors and printers, each of them has its own parameters
    - add more peripherals to the computer
    - run and test every time

- run and test every time
- customize the `toString()` method to display the required data
- consider a method that compares two computer sets
    - for example, assign some weights to the parts and calculate a score based on them

Complete Lesson