All objects of a class have the same fields and methods, but the values of object fields are usually different. At the same time, a class may also have fields and methods which are common for all objects. Such fields and methods are known as **static members,** which are declared with the `static` keyword.

In this topic, you will learn how to use static class members. As a bonus, you will finally understand the part that the `static` keyword plays in the declaration of the `main` method.

§1. Class variables

A **class variable (static field)** is a field declared with the `static` keyword. It can have any primitive or reference type, just like a regular instance field. A static field has the same value for all instances of the class. It belongs to the class, rather than to an instance of the class.

If we want all instances of a class to share a common value, for example, a global variable, it's better to declare it as static. This can save us some memory because a single copy of a static variable is shared by all created objects.

Static variables can be accessed directly by the class name. To access a static field, you should write

```
ClassName.fieldName;
```

Let's look at an example. Here is a class with two public static variables:

```
class SomeClass {

    public static String staticStringField;

    public static int staticIntField;
}
```

We can set their values and get them:

```
SomeClass.staticIntField = 10;
SomeClass.staticStringField = "it's a static member";

System.out.println(SomeClass.staticIntField); // It prints "10"
System.out.println(SomeClass.staticStringField); // It prints "it's a static member"
```

> Generally, it's not a good idea to declare **non-final public static fields**, here we just used them as an example.

We can also access the value of a static field through an instance of the class.

```
SomeClass.staticIntField = 30;

SomeClass instance = new SomeClass();

System.out.println(instance.staticIntField); // It prints "30"
```

Let's see a more complex example. Here is a class with a static field named `lastCreated`. The field stores the date of the last created instance.

```
public class SomeClass {

    public static Date lastCreated;

    public SomeClass() {
        lastCreated = new Date();
    }
}
```

The value of the static field is changed in the class constructor every time a new object is created.

The code below creates two instances and outputs intermediate results:

```
System.out.println(SomeClass.lastCreated);

SomeClass instance1 = new SomeClass();
System.out.println(SomeClass.lastCreated);

SomeClass instance2 = new SomeClass();
System.out.println(SomeClass.lastCreated);
```

In my case, the results were the following:

```
null
Sun Aug 20 17:49:24 YEKT 2017
Sun Aug 20 17:49:25 YEKT 2017
```

§2. Class constants

Static fields with the `final` keyword are **class constants**, which means they cannot be changed. According to the naming convention, constant fields should always be written in uppercase with an underscore (`_`) to separate parts of the name.

The standard class `Math` , for example, contains two static constants:

```
public static final double E = 2.7182818284590452354;

public static final double PI = 3.14159265358979323846;
```

Constants are often public, but it's not a rule.

To see how they work in an example, let's declare a class named `Physics` with two static constants:

```
class Physics {

    /**
     * The speed of light in a vacuum (m/s)
     */
    public static final long SPEED_OF_LIGHT = 299_792_458;

    /**
     * Electron mass (kg)
     */
    public static final double ELECTRON_MASS = 9.10938356e-31;
}
```

To use the constants, let's write the following code:

```
System.out.println(Physics.ELECTRON_MASS); // 9.10938356E-31
System.out.println(Physics.SPEED_OF_LIGHT); // 299792458
```

Since those fields are constants, we cannot change their values. If we try to do it, we'll get an error:

```
Physics.ELECTRON_MASS = 10; // compile-time error
```

§3. Class methods

A class may have **static methods** as well as static fields. Such methods are also known as **class methods**. A static method can be accessed by the class name and doesn't require an object of the class.

Static methods can be called directly with the class name. To access a method, you should write

```
ClassName.staticMethodName(args);
```

A static method may have arguments like a regular instance method or it may well have no arguments. But, unlike instance methods, static methods have several special features:

- a static method can access only static fields and cannot access non-static fields;
- a static method can invoke another static method, but it cannot invoke an instance method;
- a static method cannot refer to `this` keyword because there is no instance in the static context.

Instance methods, however, can access static fields and methods.

> Static methods are often used as **utility methods** that are the same for the whole project. As an example, you can create a class with only static methods for performing typical math operations.

The Java class library provides a lot of static methods for different classes. Here are just a few of them:

- the `Math` class has a lot of static methods, such as `Math.min(a, b)`, `Math.abs(val)`, `Math.pow(x, y)` and so on;
- the `Arrays` class has a lot of static methods for processing arrays such as `toString(...)`;
- `Long.valueOf(...)`, `Integer.parseInt(...)`, `String.valueOf(...)` are static methods too.

Here is a class with one constructor, a static method and an instance method.

```
public class SomeClass {

    public SomeClass() {
        invokeAnInstanceMethod(); // this is possible here
        invokeAStaticMethod();    // this is possible here too
    }

    public static void invokeAStaticMethod() {
        // it's impossible to invoke invokeAnInstanceMethod() here
    }

    public void invokeAnInstanceMethod() {
        invokeAStaticMethod();  // this is possible too
    }
}
```

This example shows that you can invoke a static method from the instance context (constructors and instance methods), but you can't invoke an instance method from a static context.

The only way to call an instance method from a static one is to provide a reference to this instance as an argument. You can also create objects of other classes and call their methods in a similar way. Here's an example:

```
public static void invokeAStaticMethod(SomeClass someClassInstance) {

    // calling instance method from static context by passing instance as an argument
    someClassInstance.invokeAnInstanceMethod();

    // calling instance and static methods of AnotherClass instance
    AnotherClass anotherClassInstance = new AnotherClass();
    anotherClassInstance.invokeAnotherClassInstanceMethod();
    anotherClassInstance.invokeAnotherClassStaticMethod();
}
```

An example of a static method is the `main` method. It always should be static.

§4. Conclusion

In this lesson, we discussed static fields and methods and some situations where we can use them. It is important to remember that static members cannot access the values of object fields since there is no instance context (`this`). Nonetheless, they are a good option for providing a set of common constants (together with `final`) and utility methods for the whole project. We will consider other helpful ways of using static members in the next topics.