

Procedural → OOP 1/2

Tamás Ambrus, István Gansperger

Eötvös Loránd University
ambrus.thomas@gmail.com

Procedural or OOP for the win?

The decision is not obvious.

- Procedural code is not specifically readable but is fast, while
- OOP is more readable but comes with numerous functions.

As the applications grow, so do their code, and readability along with maintainability comes to relevance.

That's why we learn OOP, and we do it in Java.

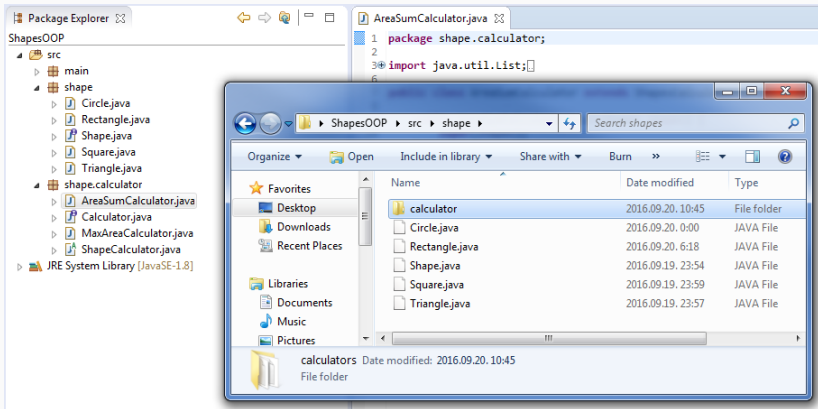
Packages

To make defined types easier to find, Java uses packages. Packages are simple folders in the file system but Java programmers think about them as **groups of related types**.

We have to put the package statement at the top of every Java source file, e.g.

```
package calculator;  
  
// ...
```

Packages



Unnamed packages are provided by the Java platform principally for convenience when developing small or temporary applications or when just beginning development.

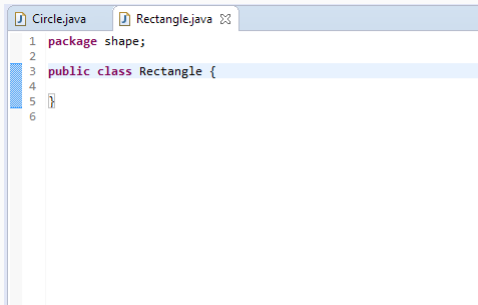
However, it is suggested to use globally unique package names to avoid conflicts.

We define types in packages. Class definitions (or interfaces or enumerations – *to be discussed later, these 3 can be defined types*) have to be put in source files in packages.

Classes mostly contain fields and methods. And by doing so they support one of the advantages of OOP: the **encapsulation**.

Encapsulation means wrapping the data and the code acting together as a single unit. These units are the classes in Java.

For instance, the *Rectangle* class can be found in the *shape* package and additionally, it is *public*.



The screenshot shows a Java IDE with two tabs: 'Circle.java' and 'Rectangle.java'. The 'Rectangle.java' tab is active and displays the following code:

```
1 package shape;  
2  
3 public class Rectangle {  
4  
5 }  
6
```

Elements can be **public** meaning they are accessible everywhere.

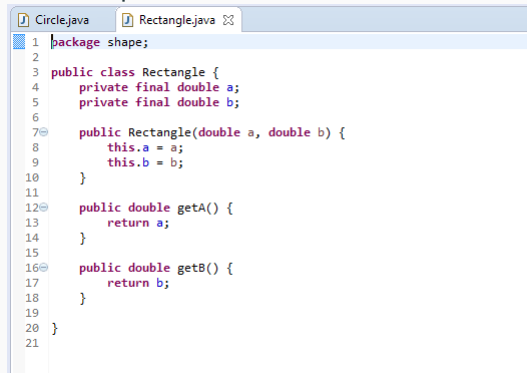
So a public class is visible in the same/different package or even in another Java program.

Its almost opposite accessibility level is the **private**, where the given element can be reached only within its defining type.

Fields, methods

So in OOP we aim at putting the related data and functions together. In Java data can be stored in **class fields** and functions are called **methods**.

For example, this is a humble definition of *Rectangle* class:

A screenshot of a Java IDE with two tabs: 'Circle.java' and 'Rectangle.java'. The 'Rectangle.java' tab is active, showing the following code:

```
1 package shape;
2
3 public class Rectangle {
4     private final double a;
5     private final double b;
6
7     public Rectangle(double a, double b) {
8         this.a = a;
9         this.b = b;
10    }
11
12    public double getA() {
13        return a;
14    }
15
16    public double getB() {
17        return b;
18    }
19
20 }
21
```

How do we read this class?

This *Rectangle* class has two fields that can be initialized only once by the constructor, and this class has two getter functions for the fields.

Yeah, it's clear, but what does that mean?

The declared fields

```
private final double a;  
private final double b;
```

We have two fields (named *a* and *b*) that store double values. These fields are private meaning they can be reached only from this class.

The fields are also **final**, which means, that

- they must be initialized (assigned to a value) in the declaration line or in the constructor
- they cannot be reinitialized

Constructors are used to initialize objects from the given type. Constructor declarations look like method declarations - except that they **use the name of the class** and have **no return type**.

Initialization

```
Rectangle r = new Rectangle(10.0, 15.0);
```

allocates space for a new *Rectangle* object in the memory **and initializes its fields** given by the rules in the constructor. We also store an *r* named *Rectangle* reference referring to this created object.

A class can have:

- only one constructor - as *Rectangle* has in this example
- more constructors - constructor overloading
- zero constructor

Default constructor

You do not have to provide any constructors for your class. The compiler automatically provides a public, no-argument one named default constructor. Once you define a constructor on your own, the default one is going to be lost.

Here is an example to a default constructor:

```
public class Rectangle {  
  
    private double a;  
    private double b;  
  
    public double getA() {  
        return a;  
    }  
  
    public double getB() {  
        return b;  
    }  
}
```

That's nice! But what does that **this** mean in the constructor?

```
public Rectangle(double a, double b) {  
    this.a = a;  
    this.b = b;  
}
```

This constructor means that we assign the value of the *parameter* *a* to class field **a** belonging to this object. And the same for **b**.

The keyword this refers to the object on which the function is called.

In this case we build the object by assigning values to its fields.

Do we always need the keyword this in the constructor?

Getters, setters

Before we write an example for practice, let's introduce the getters and setters.

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

```
public double getB() {  
    return b;  
}
```

In Java getters and setters are completely ordinary functions. The only thing that makes them getters or setters is convention. A getter for **a** is called **getA** and the setter is called **setA**. In the case of a boolean, the getter is called **isA**.

Implementation - first version

```
public class Circle {  
    private float radius;  
  
    public float getRadius() {  
        return radius;  
    }  
}
```

Implementation - second version

```
public class Circle {  
    private float diameter;  
  
    public float getRadius() {  
        return diameter / 2;  
    }  
}
```

Public fields **vs** private fields with getters and setters:

- even the visibility can be controlled more precisely
- if after setting something we have to send an event, for instance, a method is better place
- without a setter method one can only read the field - customizable
- no one wants to see "public" in a multithreaded environment

The main method

```
public static void main(String[] args) {  
    Rectangle r = new Rectangle(2, 4);  
    System.out.println(r.getA());  
}
```

This is required by the compiler, since this is the entry point of your program. It must be:

- public - to be reachable by the compiler
- static - meaning compiler does not have to instantiate the class in which this method is
- void - because it does not return a value

It must have:

- String array argument - in which command line arguments are

References

- <http://docs.oracle.com/javase/specs/#7.4.2>
- <https://docs.oracle.com/javase/tutorial/java/java00/accesscontrol.html>
- <http://stackoverflow.com/questions/2036970/how-do-getters-and-setters-work>
- <https://docs.oracle.com/javase/tutorial/java/java00/constructors.html>