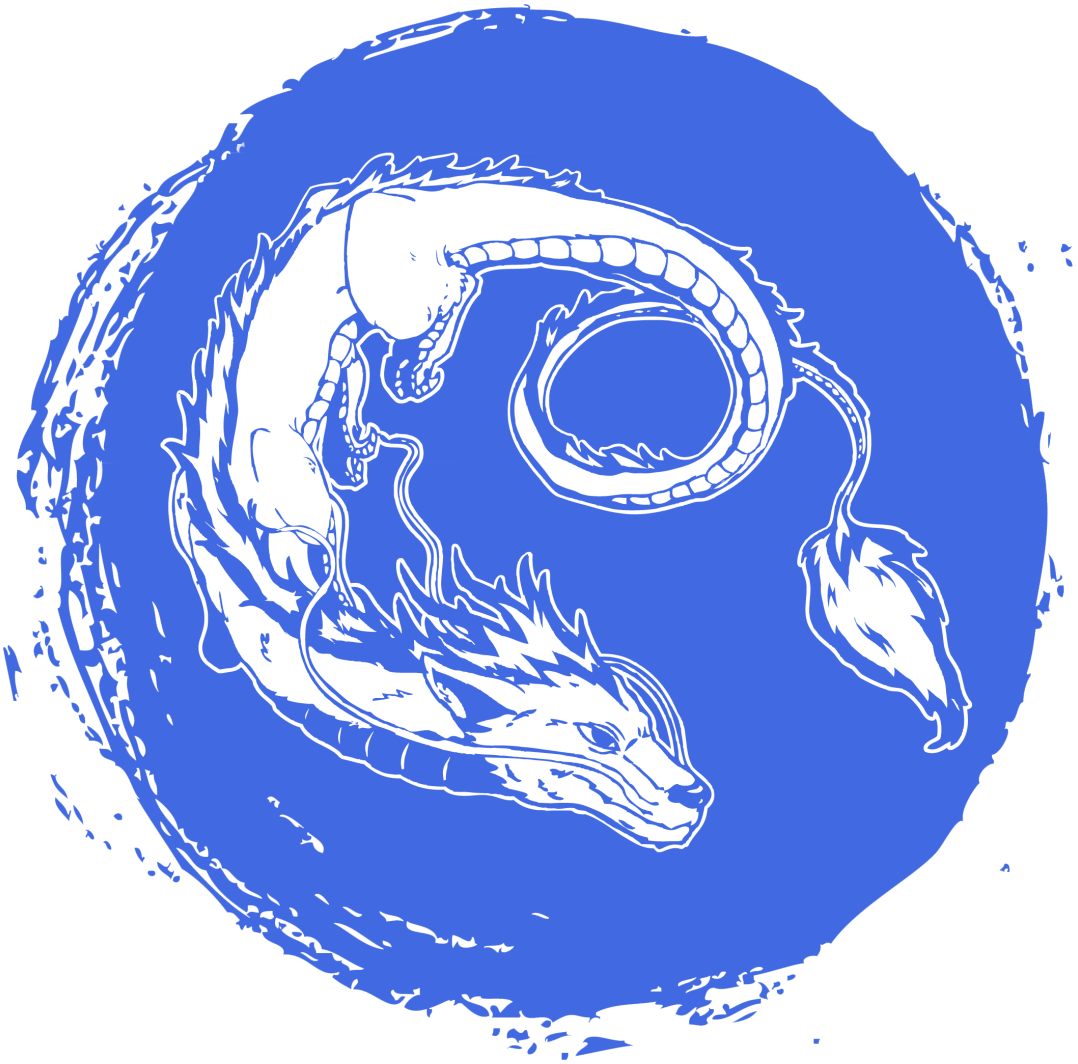
Java Workshop — Tutorial D3

version **#v1.2.3**



Yet Another Kind of Assistants 2022 <yaka@tickets.assistants.epita.fr>

Copyright

Contents

*https://intra.assistants.epita.fr

1 Files input/output

1.1 Read and Write

The most straightforward method to open and read or write into a file is:

```
public class Main {
    public static void main(String[] args) {
        Path file = Paths.get("mystery.txt");

        try {
            // Read mystery.txt
            List<String> lines = Files.readAllLines(file);

            List<String> writeLines = Arrays.asList("This", "is", "a", "message");
            Files.write(file, writeLines, Charset.forName("UTF-8"));

        } catch (IOException e) {
            e.printStackTrace();
        }

    }
}
```

This method has the advantage to automatically close the file after reading or writing into it, but this means every time you want to interact with it, it will be opened again. Moreover, it is not very memory-efficient because the whole file will be loaded into memory. We really discourage you to use this method.

Therefore, the usual method to open and read (resp. write) a file is to use the `BufferedReader` (resp. `BufferedWriter`) class. It allows buffering for efficient reading and writing of characters, arrays, and lines. It also keeps the file open until you close it yourself. These classes are wrappers around the primitive `FileReader` and `FileWriter` classes, which allow only basic interactions with files.

```
public class Main {
    public static void main(String args[]) {
        String fileName = "another-mistery.txt";
        BufferedReader bufferedReader = null;
        BufferedWriter bufferedWriter = null;

        try {
            try {
                bufferedReader = new BufferedReader(new FileReader(fileName));
                bufferedWriter = new BufferedWriter(new FileWriter(fileName));

                String line;
                while ((line = bufferedReader.readLine()) != null) {
                    System.out.println(line);
                }
                bufferedWriter.write("Another message");
            }
            finally {
                bufferedReader.close();
```

(continues on next page)

(continued from previous page)

```
                bufferedWriter.close();
            }
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}
```

1.2 Try-with-resources

As you have seen in the example, unlike the first method, `BufferedReader` and `BufferWriter` need to be closed manually after using it. Because it is redundant and error prone, the `try-with-resources` statement was introduced. It is basically a try statement that declares one or more resources and ensures that each resource is closed at the end of the statement execution.

```
public class Main {
    public static void main(String args[]) {
        String fileName = "toto.txt";

        try (BufferedReader br = new BufferedReader(new FileReader(fileName))) {
            String line;
            while ((line = br.readLine()) != null) {
                System.out.println(line);
            }
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}
```

Finally, you saw yesterday in class how to use streams and their power. Good news! You can use it to read files line by line.

```
public class Main {
    public static void main(String[] args) {
        String fileName = "file-file-file.txt";

        //read file into stream, try-with-resources
        try (Stream<String> stream = Files.lines(Paths.get(fileName))) {
            stream.forEach(System.out::println);
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}
```

1.3 File class

Java provides a very convenient way to interact with the file system: the `File` class. Here are some examples to use it.

```
// Get current working directory path
String cwd = new File("").getAbsolutePath();
System.out.println("Current location : " + cwd);

// Create an empty file in current working directory
File file = new File("file-at-bad-location.txt");
if (!(file.createNewFile()))
    System.out.println("Can't create new file at " + cwd);

// Create a new directory in current working directory
File newDir = new File("newDir");
if (!newDir.mkdirs())
    System.out.println("Can't create directory at " + cwd);
System.out.println(newDir.getAbsolutePath());

// Move file-at-bad-location.txt in the new directory and rename it
if (!(file.renameTo(new File(newDir.getAbsolutePath() + File.separator + "file-at-good-
↪location.txt"))))
    System.out.println("Can't move file to " + newDir.getAbsolutePath());

// Create several nested directories
if (!new File(newDir.getAbsolutePath() + File.separator + "subDir/subSubDir/subSubSubDir").
↪mkdirs())
    System.out.println("Can't create directories");

// Unfortunately, you cannot remove a non empty directory in java in one
// shot. You must remove all its contents first and then remove it.
 boolean deleteDirectory(File file) {
     File[] contents = file.listFiles();
     if (contents != null) {
         for (File f : contents) {
             deleteDirectory(f);
         }
     }
     return file.delete();
 }
 deleteDirectory(newDir);
```

1.4 Exercise

1.4.1 Objectives

In this exercise you will be able to practice Streams and IO in *Java*. We will test your static method *miaou* following this prototype:

```
public static void miaou(String srcPath, String dstPath, String wordToReplace);
```

This method will read the content of the source file and for each line:

1. Add the line number followed by a space
2. Replace all the occurrences of `wordToReplace` by “miaou”

The processed result will be written in the given destination file.

To do so, you will have to implement first the constructors and methods of the `MyKitten` class. Afterwards, the static method `miaou` will be straightforward.

Be careful!

You must follow thoroughly the given architecture since it will be tested. Do not add any method or attribute. Those we provide are sufficient.

1.4.2 Prototypes

You will need to implement the following methods of the `MyKitten` class:

```java
/**
 * Initializer.
 *
 * @param srcPath Source file path.
 */
public MyKitten(String srcPath);

/**
 * Use the streamContent to replace `wordToReplace` by "miaou". Don't forget
 * to add the line number before hand for each line. Store the new
 * result directly in the streamContent field.
 *
 * @param wordToReplace The word to replace
 */
public void replaceByMiaou(String wordToReplace);

/**
 * Use the streamContent to write the content into the destination file.
 *
 * @param destPath Destination file path.
 */
public void toFile(String destPath);

/**
 * Creates an instance of MyKitten and calls the above methods to do it
 * straightforward.
 *
 * @param srcPath       Source file path
 * @param destPath      Destination file path
 * @param wordToReplace Word to replace
 */
public static void miaou(String srcPath, String destPath,
        String wordToReplace);
```

1.4.3 Example

Consider an input file (`input.txt`) containing:

```
This is a test text
The purpose of this test
Is to do some replacement testing
Testing is a crucial step
They attest that you've done well
```

Calling the static method `miaou` like below:

```
MyKitten.miaou("./input.txt",
            "./output.txt", "test");
```

Will create an output file (`output.txt`) containing:

```
1 This is a miaou text
2 The purpose of this miaou
3 Is to do some replacement miaouing
4 Testing is a crucial step
5 They atmiaou that you've done well
```

2 Default methods

2.1 Quick Reminder

In Java, when you want to represent a class that cannot be instantiated, you can define it as *abstract*. An abstract class can be used to organize your inheritance hierarchy by implementing a concept as a parent class. For example, if you want to have a parent class `Animal` and children classes `Platypus` and `Otter`, you can define `Animal` as abstract because animal is just a concept.

An interface allows you to ensure that your class implements a specific behavior. The main use of interfaces is to allow *multiple inheritance* because Java does not allow *multiple inheritance* of **classes**.

Use abstract class when you want to represent an abstract concept defined by children classes, and interface when you want to define a behavior that other classes must implement or if you want to use multiple inheritance.

2.2 Definition

Since Java 8, we can specify a default behaviour for a method of an interface. It allows to avoid duplicating code between two different classes that implement the same interface similarly. Of course, we can override our default behaviour. Using default methods, there is no need to update each class that implements our interface to add a default behavior.

To create a *default*, the keyword `default` must be placed before the declaration of a method in the interface. Be careful, unlike basic methods in interfaces, once you define a method as *default*, you must implement a body.

```java
public class Main {
    public interface MyInterface {
        default public void myDefault() {
            System.out.println("Burn all the bridges then fire in the hole");
        }
    }
}
```

This interface contains a default method `myDefault` which will print `Burn all the bridges then fire in the hole`.

You can use the following test class:

```java
public class TestClass implements MyInterface {
    public static void main(String[] args) {
        TestClass test = new TestClass();
        test.myDefault();
    }
}
```

You already know that a class can implement one **or more** interfaces, conflicts between default methods may therefore appear.

```java
public interface DefOne {
    default public void myMethod() {
        System.out.println("Rust in Peace");
    }
}

public interface DefTwo {
    default public void myMethod() {
        System.out.println("Hardwired... To Self Destruct");
    }
}
```

If you create a class `Diamondo` by making it implement both interfaces, compiler will complain because it does not know which `myMethod` to choose. This problem is called **diamond inheritance problem**. You will be forced to override the method in your class.

```java
public class Diamondo implements DefOne, DefTwo{
    public void myMethod() {
        // If you want Diamondo.myMethod() to have the same behavior
        // of DefOne.myMethod(), you can use :
        DefOne.super.myMethod();
    }
}
```

2.3 Difference between abstract classes and interfaces with default methods

Watch out! Abstract classes and interfaces with default methods are very different and have two different usage. An interface with a default method is just an interface and you have to use it as an interface.

If you need to represent an abstract concept, and want private attributes, you need to use an abstract class. But if you want to define a behavior and/or *multiple inheritance*, you need to use interface. Default methods are just a tool to factorize your code.

3 Functional

3.1 Functional interface

In *Java*, a `functional interface` is an interface that **only** contains a **single abstract method** and optionally some default and static methods. When working with `functional interfaces`, we advise you to use the `@FunctionalInterface` annotation to ensure that the functional interface cannot have more than one abstract method. Indeed, the *Java* compiler will not allow an annotated functional interface to have multiple abstract methods.

```
public abstract class Vehicle {
    final protected String model;

    protected Vehicle(final String model) {
        this.model = model;
    }

    abstract void roll();
}

public class Car extends Vehicle {
    public Car(final String model) {
        super(model);
    }

    @Override
    void roll() {
        System.out.println(model + " is rolling: VROOUUUMMMMMM");
    }
}

public class Motorcycle extends Vehicle {
    public Motorcycle(final String model) {
        super(model);
    }

    @Override
    void roll() {
        System.out.println(model + " is rolling: Beh Beeeh Beh Beeh");
    }
}
```

(continues on next page)

(continued from previous page)

```java
@FunctionalInterface
public interface VehicleFactory<TYPE_T extends Vehicle> {
    // The only abstract method of the interface.
    TYPE_T create(String name);

    // Default method calling the interface abstract method.
    default void createAndRoll(String name) {
        create(name).roll();
    }
}

public class Main {
    public static void main(String[] args) {

        /* Anonymous class with an implemented method matching the abstract
           VehicleFactory.create() method. */
        VehicleFactory<Motorcycle> motorcycleFactory = new VehicleFactory<Motorcycle>() {
            @Override
            public Motorcycle create(String name) {
                return new Motorcycle(name);
            }
        };

        /* Lambda expression matching the abstract VehicleFactory.create()
           method. */
        VehicleFactory<Car> carFactory = (name) -> new Car(name);

        motorcycleFactory.create("Peugeot 103 Nitro").roll();
        carFactory.createAndRoll("Audi RS4 Gris Nardo");
    }
}
```

The main advantage of `functional interfaces` is that they can be implemented using lambdas that match the prototype of the interface's abstract method. Therefore, you do not need to use anonymous classes in order to implement and use interfaces.

3.2 Lambda expressions

`Lambda expressions` allow you to write more concise, easier to write, read and maintain code. A `lambda expression` is an anonymous function. Indeed, its definition does not take any return type, access modifiers or name.

As shown in the previous example, it is a syntactic sugar for anonymous classes implementing an interface with a single abstract method: `functional interface`. This syntactic sugar also allows you to define a method directly where it is used.

When a `lambda expression` is evaluated by the compiler, the compiler infers the signature of the `lambda` to the prototype of the single abstract method of the underlying `functional interface`. This allows the compiler to retrieve information about types of the `lambda parameters`, the type of its return value or exceptions the `lambda` can throw.

Here is the general syntax of a lambda expression:

```
// no argument
() -> expression;

// multiple arguments
(arg1, arg2, ..., argN) -> expression;

// one argument and several instructions
arg -> { instr1; instr2; ...; return expression; }
```

You will encounter more `lambda` examples throughout the tutorial. Indeed, `lambdas` are one of the main features of the *Java* language since their introduction in *Java 8*. They are way more used in *Java* than they are in *C++*, do not worry, they are also easier to use.

3.3 Method Reference

Sometimes a `lambda` does nothing but calling an existing method. In this case, writing a `lambda` would be syntactically heavier than referring directly to the called method. Method references enable you to do so: they are compact, easy-to-read lambda expressions for methods that already have a name.

The syntax of method references is composed of 3 elements:

- a **qualifier** that specifies the name of a class or instance on which the method will be invoked
- the **operator** `::`
- an **identifier** that specifies the name of the method or the `new` operator that refers to the constructor

```
public static void main(String []args){
    List<String> list = Arrays.asList("foo", "bar");

    /* With a lambda */
    list.forEach(str -> System.out.println(str));

    /* With a method reference */
    list.forEach(System.out::println);
}
```

They are 4 types of `method references`:

- reference to a static method
- reference to an instance method of a particular object
- reference to an instance method of an arbitrary object of a particular type
- reference to a constructor

Going further...

You can learn more about lambdas and method references by reading the Oracle documentation and the *Java* specification.

4 Streams

4.1 Presentation

Streams are a very powerful tool for general programming. Given a source of elements, for example a `Collection`, streams allow you to express a flow of operations in a concise and efficient way on this data. `Stream<Element_type>` is also a data structure more general than `List<Element_type>`, as it does *lazy evaluation*.

Lazy evaluation means, in the general cases, that expressions are not evaluated when they are bound to variables, but their evaluation is deferred until their results are needed by other computations. In our case, `Streams` will load data and apply transformation on it only if you use it in a later transformation. This means that you can work on a large data set efficiently without exploding your RAM or treat an infinite number of data.

Going further...

The famous MIT book "Structure and Interpretation of Computer Programs" (SICP) includes an awesome chapter on streams. In short, streams are a delayed data structure which allows to build values on the fly and modelize infinite sequences. Read that book!

To keep a correct behaviour, most stream operations must be **non-interfering** (they do not modify the underlying data-source) and **stateless** (their result should not depend on any state that might change during the execution of the stream pipeline).

4.2 Common operators

`Streams` support numerous useful operations:

- The `filter` operator takes a predicate and returns a stream containing the elements that match this predicate:

```
Stream.of(1, 2, 3, 4, 5)
    .filter(i -> i < 3);
// The stream now contains (1, 2)

// Another way to obtain a stream, by using collections
ArrayList<Integer> list = new ArrayList<>(Arrays.asList(1, 2, 3, 4, 5));
var res = list.stream().filter(i -> i < 3);
// The res stream contains (1, 2)
```

- The `map` operator takes a function and returns a stream containing the results of applying the given function to the elements of this stream:

```
Stream.of(1, 2, 3, 4, 5)
    .map(i -> i + 1);
// The stream now contains (2, 3, 4, 5, 6)
```

- The `sorted` operator. If no parameter is given, it will sort the given elements by natural order, otherwise, it takes a comparator and sorts the stream consequently:

```java
Stream.of(3, 5, 2, 1, 4)
    .sorted();
// The stream now contains (1, 2, 3, 4, 5)

Stream.of("ah", "one", "two", "three")
    .sorted((i, j) -> i.compareTo(j));
// The stream now contains ("ah", "one", "three", "two")
```

- Using `.collect(Collectors.toList())` converts your stream back to a list:

```java
List<Integer> list = Stream.of(1, 2, 3, 4, 5)
    .filter(i -> i < 3)
    .collect(Collectors.toList());
```

Auxiliary functions can also be useful to convert your data:

- You can use `.stream()` on a collection to convert it to a stream;
- You can also use `.parallelStream()` to parallelize operations on that stream;
- You can produce a stream out of an array using `Arrays.stream(your_array)`.

Finally, because most of the operators return a stream, you can directly chain the operations.

```java
people.stream()
    .filter(p -> p.getAge() >= 18)
    .map(Person::getFirstName)
    .collect(Collectors.toList());
```

Going further...

There are many more operations that can be done with streams. If you want more information about streams, please refer to the java specification[1].

[1] https://docs.oracle.com/javase/8/docs/api/java/util/stream/Stream.html

Now that you know more about streams, what does this example do?

```java
public static void main(String[] args) {
    Files.lines(Paths.get("big-file.txt"))
            .limit(10)
            .map(elt -> new Tuple<>(elt.split(" ").length, elt))
            .filter(elt -> elt.getKey() % 2 == 0)
            .sorted(Comparator.comparing(Tuple::getKey))
            .map(elt -> new StringBuilder(elt.getValue()))
            .filter(elt -> elt.indexOf("BAM") != -1)
            .map(elt -> String.valueOf(elt.reverse()))
            .reduce("", (accumulator, elt) -> accumulator + elt);
}
```

4.3 Exercise

4.3.1 Objectives

You will implement some functions which will perform operations on `List` using `Stream`.

Be careful!

For the whole exercise, the use of loops and forEach methods is forbidden.

4.3.2 Specifications

Create a public class named `StreamsSongs` in the package `dataAccess` with the following static methods.

getOlderArtists

Get the 10 first **distinct** artist surnames of artists that are 30 years old or older. You have to keep the order from the given list.

- Prototype:

```
public static List<String> getOlderArtists(List<Song> songs) { /* ... */ }
```

getSumAges

Get the sum of ages of all artists that are 20 years old or more.

- Prototype:

```
public static Integer getSumAges(List<Song> songs) { /* ... */ }
```

getMusics

Get the 10 first music titles whose artist name contains the string *an*. The case does not matter. Hence, you must also match *aN*, *An*, and *AN*. You have to keep the order from the given list.

- Prototype:

```
public static List<String> getMusics(List<Song> songs) { /* ... */ }
```

5 Value objects

5.1 Presentation

From Wikipedia[1]:

> In computer science, a value object is a small object that represents a simple entity whose equality is not based on identity: i.e. two value objects are equals when they have the same value, not necessarily being the same object.

In other words, the motivation for value objects is to use the nature of OOP to give more importance to the value represented by an entity than the entity itself. Therefore **entities** of the same type can only be distinguished from one another by **values** they are representing.

5.2 Problematic

In most of our programs, values and fields are often represented with standard types or data structures, defined by the language itself (`int`, `String`, etc.). Those types have no particular semantics in the context of our program whereas **entities** can fulfill this lack of semantics.

For example, say we have a class `Computer`, it would probably have `String` fields like `brand`, `keymap`, `model`. The computer is an entity and by nature, has an identity, defined by its type (it's a `Computer` instance), and the value of its fields. However, the identity of the different fields are only distinguished by their values (nothing would fail if I call my **brand** "AZERTY", it is a valid **value** for a string even if it does not make sense).

5.3 Transform values to objects

We would need many tests to check for the integrity of a class' attributes, involving heavy logic or conceptual algorithms. A better approach would be to use the power of OOP to combine concepts with the values themselves. That's what value objects do.

To do that, we will create new classes, wrapping each entity type we want to distinguish and/or associate logic to. In our example, we would create four value objects, named by the concept they represent:

```
/* Entity */
public class Computer {
    private Brand brand;
    private Model model;
    private Keymap keymap;
    private Resolution resolution;

    /* ... */
}

/* Value objects */
```

(continues on next page)

[1] https://en.wikipedia.org/wiki/Value_object#Value_objects_in_Java

(continued from previous page)

```java
public class Brand {
    public final String value;
}

public class Model {
    public final String value;
}

public class Keymap {
    public final String value;
}

public class Resolution {
    public final int width;
    public final int height;
}
```

In the previous example, you can notice that we made the value object classes immutable (with the `final` keyword). Instead of changing the value contained in a value object, we force users of our class to create new instances to wrap new values. To make a class immutable, it must only contain attributes declared as `final` and immutable themselves:

- `String` or numeric types
- immutable objects (which contain only immutable fields)
- collections made immutable with `Collections.unmodifiableList/Map/Set`

From the conceptual perspective, it makes sense to create a new instance of a value object when the value changes, as we are literally assigning a new value to it, defining a new identity to our entity.

For example, if we want to create a new `Resolution` value for our computer, we would **not** write:

```java
public void setResolution(int newWidth, int newHeight) {
    resolution.setWidth(newWidth); // Impossible, width is final
    resolution.setHeight(newHeight); // Impossible, height is final
}
```

Indeed, the `final` property of `Resolution`'s fields would prevent setters for `width` and `height` to exist. Instead, we would do (provided that a convenient constructor exists):

```java
public void setResolution(int newWidth, int newHeight) {
    resolution = new Resolution(newWidth, newHeight);
}
```

But what if we want to partially change the value of `resolution`? If we only want to change the width, but not the height, we could keep the same structure and do:

```java
public void setResolutionWidth(int newWidth) {
    resolution = new Resolution(newWidth, resolution.getHeight());
}
```

However, since changing only one field is semantically more a modification of the value rather than a creation of a new value, we want to avoid calling a constructor at this level. The common method to

get a new value which is a transformation of an existing value object, is to call a method `withField`, which modifies only one field and returns a new value which is a partial copy of the previous value:

```
public class Resolution {
    private final int width;
    private final int height;

    public Resolution(final int width, final int height) {
        this.width = width;
        this.height = height;
    }

    public Resolution withWidth(final int width) {
        return new Resolution(width, height);
    }

    @Override
    public boolean equals(Object o) {
        return width == ((Resolution)o).getWidth()
            && height == ((Resolution)o).getHeight();
    }
}

public class Computer {
    /* ... */

    public setResolutionWidth(int newWidth) {
        resolution = resolution.withWidth(newWidth);
    }

    /* ... */
}
```

This technique can also be useful to chain transformations of value objects if more than one `withField` method is called to modify different fields.

Having to instantiate more objects is a small drawback compared to the benefits in structure and readability of the code, since the garbage collector is especially good at cleaning short life-time items.

As the default `equals` method is an identity comparison, we overrided the `equals` method of the `Resolution` class in the previous example, because to compare two value objects we want to examine all of their fields.

5.4 Advantages

Although the number of classes in our project would increase significantly, there are several benefits to use value objects to wrap simple values.

With value objects, our code gets safer and more expressive, since our types are named according to the concept they represent:

```
// Without value objects
Map<String, String> modelToBrand;
```

(continues on next page)

(continued from previous page)

```
// With value objects
Map<Model, Brand> modelToBrand;

// Without value objects
Computer(String model, String brand) {
    /* ... */
}
new Computer("Microsoft", "Surface Pro"); // Compiles

// With value objects
Computer(Model model, Brand brand) {
    /* ... */
}
new Computer(new Brand("Microsoft"), new Model("Surface Pro")); // Doesn't compile!
```

Moreover, as said before, they allow us to encapsulate logic or algorithms within our types:

```
public class Keymap {
    private final String keymap;

    public Keymap(String keymap) {
        if (!keymap.equals("AZERTY") && !keymap.equals("QWERTY")) { // As an example
            throw new IllegalArgumentException("Invalid keymap.");
        }
        this.keymap = keymap;
    }
}
```

In addition, using value objects makes our program safer to multi-threading. Since we use immutable objects, they can easily be shared between several threads without any risks.

5.5 Exercise

5.5.1 Computer Shop

The goal of this exercise is to practice the concept of **value object**. Do not forget that the different value objects you will implement throughout this exercise should comply with the value objects' requirements.

Your value objects should also have getters named according to the convention `get<Field>`.

The constructors of the different value objects and classes should throw an `IllegalArgumentException` in case the values given to instantiate the object are not valid.

5.5.2 Sellable Interface

It will be possible to add or withdraw products from a shop and to execute various operations on those products. A product can be of any class as long as it implements the `Sellable` interface, defined as bellow:

```
public interface Sellable {
    Price getPrice();
    void applyDiscount(int discount);
    void print();
}
```

The `applyDiscount` method takes an `int` representing the discount to apply on the price of a sellable object, in percentage.

5.5.3 Computer Class

You must implement an abstract class named `Computer` that implements the `Sellable` interface. The class holds four attributes:

- a `String brand`;
- a `Processor`;
- a `SerialNumber`;
- a `Price`.

```
public Computer(String brand, Processor processor,
        SerialNumber serialNumber, Price price) {
    /* ... */
}
```

If the parameters passed to the `Computer` constructor are null, you should throw an `IllegalArgumentException`.

Processor Class

The `Processor` value object should contain the following elements:

- `String reference`: the processor's reference, which should be alphanumerics that can also contain dashes;
- `int numberOfCores`: the number of cores of the processor;
- `float frequency`: the frequency of the processor, in GHz.

```
public Processor(String reference, int numberOfCores, float frequency) {
    /*...*/
}
```

SerialNumber Class

The `SerialNumber` value object should contain a unique alphanumerical string.

```
public SerialNumber(String serialNumber) {
    /* ... */
}
```

Price Class

The `Price` value object should contain the price of the computer as well as its currency, of type `enum Currency`, which looks like:

```
public enum Currency {
    US_DOLLAR,
    EURO,
    POUND
}
```

```
public Price(float price, Currency currency) {
    /* ... */
}
```

This class should implement a `withDiscount(int discount)` method taking a discount in percentage, and returning a `Price` with the discount applied to it.

5.5.4 Laptop Class

You must implement a `Laptop` class that extends `Computer`. It must hold a `Dimension` attribute, representing the dimensions of the laptop.

```
public Laptop(String brand, Processor processor, SerialNumber serialNumber,
        Price price, Dimension dimension) {
    /* ... */
}
```

The `Laptop` class should have a `print` method to display all the characteristics of the instance, like so:

```
<brand>: <serialNumber>: <currencySymbol><price>
--- Processor: <reference> - <nbOfCores> cores - <frequency> GHz
--- Dimensions: <height>x<width>x<thickness> mm
```

Dimension Class

The `Dimension` value object must have three attributes, a `height`, a `width` and a `thickness`, in millimeters.

```
public Dimension(int height, int width, int thickness) {
    /* ... */
}
```

This class should have 3 methods `with<Field>`, taking an `int`, that transform the current `Dimension` value object by modifying one of its fields.

5.5.5 Desktop Class

You must implement a `Desktop` class that extends `Computer`. It must hold a `Monitor` attribute, representing the presence, or not, of a monitor component and its features.

```
public Desktop(String brand, Processor processor, SerialNumber serialNumber,
        Price price, Monitor monitor) {
    /* ... */
}
```

A `Desktop` can be a simple computer tower, without a `Monitor`. Its `print` method should display the following output:

```
<brand>: <serialNumber>: <currencySymbol><price>
--- Processor: <reference> - <nbOfCores> cores - <frequency> GHz
[--- Monitor: <size> inches - <widthResolution>x<heightResolution> pixels]
```

Monitor Class

This value object should contain a `size` attribute for the size of the screen, in inches, and a `widthResolution` and a `heightResolution` attributes.

```
public Monitor(int size, int widthResolution, int heightResolution) {
    /* ... */
}
```

As for the `Dimension` value object, you should have 2 methods `with<Field>` that transform your `Monitor`: one for the size and one for the resolution (`withResolution(int widthResolution, int heightResolution)`).

5.5.6 Shop Class

You have to implement a `Shop` class containing a list of `Sellable` products. This class should implement some methods:

- constructor(s);
- a `print` method that should match the example bellow;
- an `addProduct` method that adds a given item to the list of available products. This method should throw an `IllegalArgumentException` if the new product is null;
- a `removeProduct` method, that removes an item from the list of products;
- an `applyDiscount` method, taking an int representing the discount in percentage, and applying it to all the shop's products;
- a `sort` method that sorts the shop products by price.

You should be able to convert euros and pounds to dollars with the following conversion values:

- from euros to dollars: price x 1.2
- from pounds to dollars: price x 1.4

This is the expected output for the provided main:

```
**********
Samsung: SAM123: €529.0
--- Processor: i3-L13G4 - 5 cores - 2.8GHz
--- Dimensions: 312x468x12 mm
Asus: AS456: $698.9
--- Processor: i7-1180G7 - 4 cores - 2.2GHz
--- Dimensions: 468x512x9 mm
Lenovo: SUS798: €998.0
--- Processor: i5-1145G7 - 4 cores - 2.6GHz
--- Monitor: 15 inches - 1920x1080 pixels
**********
**********
Samsung: SAM123: €529.0
--- Processor: i3-L13G4 - 5 cores - 2.8GHz
--- Dimensions: 312x468x12 mm
Asus: AS456: $698.9
--- Processor: i7-1180G7 - 4 cores - 2.2GHz
--- Dimensions: 468x512x9 mm
Lenovo: SUS798: €998.0
--- Processor: i5-1145G7 - 4 cores - 2.6GHz
--- Monitor: 15 inches - 1920x1080 pixels
**********
Remove failed.
**********
Samsung: SAM123: €529.0
--- Processor: i3-L13G4 - 5 cores - 2.8GHz
--- Dimensions: 312x468x12 mm
Lenovo: SUS798: €998.0
--- Processor: i5-1145G7 - 4 cores - 2.6GHz
--- Monitor: 15 inches - 1920x1080 pixels
**********
```

Don't be afraid, I just wanna help you.