Java Workshop: D3 — Tutorial

version #

Yet Another Kind of Assistants 2020 <yaka@tickets.assistants.epita.fr>

Copyright

Contents

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

1 Value objects

1.1 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 equal 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.

1.2 Problematic

In most of our programs, values and fields are often represented with standard types or data structures, which are defined by the language itself and have no particular semantics in the context of our program other than how they contain data (`int`, `BigDecimal`, `String`, `ArrayList`, etc.). Entities of the same type cannot be distinguished from one another by anything other than their **values**.

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 and their semantics (but nothing would fail if I call my brand "AZERTY" or my keymap "ZenBook", they both are just different valid **values** for strings).

1.3 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:

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

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

    /* ... */
}

/* Value objects */

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;
}
```

Therefore, a value object's identity is defined by the values of its fields. Thus, if we want to compare two `Resolution` objects, we will compare all of its fields:

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

Moreover, 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:

```java
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);
    }
}

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 gains in structure and readability of the code, since the garbage collector is especially good at cleaning short life-time items.

1.4 Advantages

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

First, our code gets more expressive, since our types are named according to the concept they represent:

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

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

Plus, similarly to the code above, the type system of value objects allows us to make the code safer. In the example underneath, we mix up the model and the brand of our computer:

```java
// 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:

```java
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.

2 Time

Since Java 8, the language provides the `java.time` library to represent dates. This API allows us to represent the date and time depending on the precision and the needed quantity of information.

2.1 Temporal

The `Temporal` package contains several interfaces that are at the root of the `java.time` API, and are implemented by the classes that will be presented in this section.

The most notable interfaces of this package are:

- `Temporal` represents a - potentially zoned - date, time, point in time: To name only two, `LocalDateTime` and `Instant` implement this interface;
- `TemporalAmount` represents an amount of time: `Period` and `Duration` implement this interface;
- `TemporalUnit` represents a unit of time such as seconds, months, centuries...: the enum `ChronoUnit` implements this interface and gives access to a wide variety of units of time;
- `TemporalField` represents a field of a date, such as the day of the week or the month of the year: the `ChronoField` enum provides a set of those possible fields.

2.2 Dates

First of all, Java provides enums to represent days and months: `DayOfWeek` and `Month`. You can get their textual representation by using the method `getDisplayName`, passing a `TextStyle` and a `Locale`. `TextStyle` is an enum that can take the values `FULL`, to get the complete name, `SHORT` to get an abbreviation (e.g. "Jul" for July with the English locale) or `NARROW`, to get a single letter. You can get the system's default locale by calling `Locale.getDefault()`.

To represent a date without the notion of time nor timezone, we use the `LocalDate` class. It is instantiated with the static method `of(int year, Month month, int dayOfMonth)`, and only contains those three values. You can get the `DayOfWeek` of a `LocalDate` by using the method `getDayOfWeek`. `LocalDate` objects are value objects.

Some classes also represent parts of a date, namely `YearMonth`, `MonthDay` and `Year`: they are useful when you don't need a heavy, complete date. They also provide some useful methods, such as `Year.isLeap()` to check if a year is a leap year.

2.3 Time

To represent the time of a day, without any notion of date nor timezone, we use the class `LocalTime`. Although it is mostly used to represent time as it could be seen on a clock, the time is actually stored with a nanosecond precision. `LocalTime` instances are value objects. You can obtain the current time based on the system clock with `LocalTime.now()`.

2.4 DateTime

2.4.1 LocalDateTime

One of the main classes we use to represent date and time is `LocalDateTime`. It is based on the ISO 8601, and doesn't contain time zone information. This is the combination of a `LocalDate` and a `LocalTime`. To instantiate a `LocalDateTime` for a known date, you can use the static method `of`, specifying at least the year, month, day, hour and minute (second and nanosecond are defaulted to 0). The class provides several useful methods:

- `now()`: static method that gives the current `LocalDateTime` based on the system clock;
- Getters on each field (e.g. `getHour()`, `getMonth()`...). It is possible to get different information about the day: `getDayOfWeek()`, `getDayOfMonth()`, `getDayOfYear()`...
- Comparisons to other `LocalDateTime`, through `isAfter()`, `isBefore()` and `isEqual()`;
- `plus`/`minus` methods: return a new `LocalDateTime`, adding or subtracting the specified quantity of a field, depending on the method called, i.e. `plusDays(long days)`, `plusHours(long hours)`...
- `with` methods: return a new `LocalDateTime`, but with specific fields modified with the given arguments, as in `withDayOfMonth(int dayOfMonth)`, `withHour(int hour)`...

As for `LocalTime` and `LocalDate`, `LocalDateTime` instances are value objects.

2.4.2 Time zones

To represent a time zone, the `java.time` library offers two classes: `ZoneId` and `ZoneOffset`. `ZoneID` represents a time zone identifier, which can be implemented either as an `Area/City` such as `Europe/Paris`, or with a fixed offset of UTC/GMT, such as `+02:00`. `ZoneOffset` extends `ZoneId` and defines the fixed offset of the current time-zone with UTC/GMT. They are both value-based classes.

Those two classes can be combined with a `LocalDateTime` to form a `ZonedDateTime` or an `OffsetDateTime`, to express a date and time in relation to a time zone.

2.4.3 Parsing and formatting

`LocalDate`, `LocalTime`, and more generally `LocalDateTime`, provide a `parse()` static method to parse a date and time from a string and get an object. It takes the string to parse as parameter, and can be overloaded with a second argument of type `DateTimeFormatter`. A `DateTimeFormatter` is a class that represents the formatting of a date, that can either be predefined, or user-defined through a format string. Details on how to describe patterns are listed in the class' javadoc[1]. If no `DateTimeFormatter` is provided to `parse()`, it will expect a date in ISO standard format. As an example, March 14, 2042 at 4:00 PM should be formatted `"2042-03-14T16:00:00"` to be parsed by the default `DateTimeFormatter`.

Those same classes provide a `format()` method, that takes a `DateTimeFormatter` and returns a string with date and/or time expressed in the given format.

[1] https://docs.oracle.com/en/java/javase/11/docs/api/java.base/java/time/format/DateTimeFormatter.html

2.5 Instant

The `Instant` class is another class widely used to represent time. Rather than representing a date and time with days, months, seconds or hours, it represents a point in time expressed in nanoseconds relatively to the EPOCH (the first of January, 1970, at midnight). It doesn't mean that the minimal date that can be expressed is the EPOCH, a date anterior to it will be stored with a negative nanoseconds count.

Similarly to `LocalDateTime`, `Instant` provides some useful functions to manipulate it, such as `now()` to get the current `Instant`, the comparisons (`isAfter()`, `isBefore()`) and the `plus` and `minus` methods. It is also possible to create an `OffsetDateTime` and a `ZonedDateTime` from an `Instant` through the methods `atOffset()` and `atZone()`. Also, `Instant` provides a `parse()` method, but does not offer the possibility to specify a format: it must be formatted as an ISO instant. Also, there is no `format` method, but `toString()` can be used to get a textual representation of the `Instant`. `Instant` objects are value objects.

2.6 Period and Duration

To express an amount of time, the `java.time` library provides two classes: `Duration` and `Period`. The former uses nanoseconds to measure the amount of time with precision, and is commonly used with `Instant`, while the latter expresses the amount of time in terms of date, with years, months and days, and is better used with `LocalDate`. Both of those classes are value-based classes. Both classes can be instantiated using the static methods `of`, specifying the amount of time in the desired unit. They also offer the `plus` and `minus` operations, as well as a product with a scalar with the `multipliedBy()` method. You can also obtain a `Duration` by calling the static method `between()` with any two `Temporal`. You can also use between with `Period`, but you can only compute the difference between two `LocalDate`.

There is a third way to express amounts of time, using the `ChronoUnit` `between()` method. It allows to express an interval of time in a single given unit. For instance, the difference between 12 January 2017 and 12 January 2018 would give a `Period` with the field year set to 1, 0 month, and 0 day. The same difference using `ChronoUnit.DAYS.between()` would give 365 days.

3 Observer

3.1 Presentation

Sometimes, a class may need to be informed of any attribute changes from another class. This is common when developing an application composed of a business or data class, and view classes displaying the data contained in the business layer. A good use case is the Model-View part of the MVC (Model-View-Controller) architectural pattern.

The `Observer` design pattern can handle the situation. It is about defining a one-to-many relationship between objects. If one object (the **subject**) changes, all depending objects (the **observers**) are informed of this change and updated automatically. In an online video game for example, the player would be a subject, and the GUI and network objects would be observers: when the player's stats change, the GUI and the server are notified and updated accordingly.

3.2 Java implementation

In the Java standard library, a class and an interface are provided to implement the `Observer` design pattern:

- `Observable`[1]: when extending this class, a class is designed to be the **subject** of the design pattern. We can use methods such as `add` to add observers on this subject, and `notifyObservers` that notifies (if this subject has changed) all of its observers. It has changed when `hasChanged` is `true`. `setChanged` marks this subject as changed. Other methods are provided, feel free to read the documentation to know more;
- `Observer`[2]: this interface forces any class that implements it to have an `update` method, which will be called every time a subject observed by this observer calls its `notifyObservers` method.

3.3 Note about JDK 9

Since the release of Java 9 in October 2017, `Observer` and `Observable` are labelled as deprecated, but are one of the simplest implementations of this design pattern.

The reason for this deprecation is described in this issue.

The alternative to this is the `Listener` system, that you can find in the `java.awt.event` part of the library. However, we introduce you to observers for educational purposes, as is a good introduction to this design pattern.

3.4 Example

Picture a simple example, with one subject: the assistants' laboratory, and only one observer: a class logging the number of people in the Lab every time someone enters.

3.4.1 The assistants' laboratory

```java
import java.util.HashSet;
import java.util.Observable;

public class AssistantsLab extends Observable {
    public void enter(Student student) {
        studentsInLab.add(student);
        setChanged();
        notifyObservers();

        /* Greet and listen to student's request */
    }

    public HashSet<Student> getStudentsInLab() {
```

(continues on next page)

[1] https://docs.oracle.com/en/java/javase/11/docs/api/java.base/java/util/Observable.html
[2] https://docs.oracle.com/en/java/javase/11/docs/api/java.base/java/util/Observer.html

(continued from previous page)

```
        return studentsInLab;
    }

    private HashSet<Student> studentsInLab = new HashSet<Student>();
}
```

In this class, all the students currently present in the laboratory are represented in a single set. Each time a student enters in the laboratory, the `enter` method is called. Once the student has been added in the set, observers must be notified. To do so, the `setChanged` method is called, followed by `notifyObservers`. It will automatically call the `update` method of each observers which have been added to this subject.

3.4.2 The entries logger

```
import java.util.Observer;
import java.util.Observable;

public class EntriesLogger implements Observer {
    @Override
    public void update(Observable o, Object arg) {
        System.out.println(((AssistantsLab)o).getStudentsInLab().size()
            + " student(s) are present in the Lab.");
    }
}
```

This is the only observer. For it to be a valid observer, it must implement the method `update`. This is the method called when the subject notifies its observer.

3.4.3 Usage

```
public static void main(String[] args) {
    AssistantsLab lab = new AssistantsLab();

    lab.addObserver(new EntriesLogger());

    lab.enter(new Student("login_x", 2021));
    lab.enter(new Student("tigrou", 2020));
}
```

```
1 student(s) are present in the Lab.
2 student(s) are present in the Lab.
```

Here, we only have to add the `EntriesLogger` object to the list of observers of our subject. After that, we can use our `AssistantsLab` as we want, without worrying about communications between the subject and its observer.

4 Bowling Game Kata

4.1 Objectives

The goal of this exercise is to practice the Test Driven Development (TDD) process, by following the Bowling Game Kata (from http://butunclebob.com/ArticleS.UncleBob.TheBowlingGameKata). It is recommended that you do this exercise at the same pace as your assistants. However, all the steps are at your disposal if you are ahead of the group and want to do the exercise by yourself.

4.2 Specifications

A kata is an exercise used to learn a skill through practice and repetition. The Bowling Game Kata is a popular kata to learn TDD.

4.2.1 Test Driven Development

The objective of Test Driven Development is to create the tests before coding to use the test as a validation of our feature. Then we can refactor our code to obtain a clean code while using the tests created previously as a security harness which prevents our refactoring from altering any existing functionality.

The TDD process is a loop composed of three steps:

- Create a new test that the current code fails;
- Update the code to pass this test without failing previous tests;
- Refactor the code, by checking tests to avoid breaking existing features.

All three steps must be respected. If even the code refactoring step is not followed, you are not applying the TDD process.

4.2.2 Bowling Game Kata

With the Bowling Game Kata exercise we must create a program that, given a valid sequence of rolls, will produce the total score of the game. We will not check for the validity of a roll or the given number of rolls, and we will not compute the intermediate score. For this part, “rolling `n`” means knocking down `n` pins in a roll.

The rules of the bowling game are the following:

- A game lasts ten turns;
- A turn consists in knocking down ten pins with a ball from a distance, in two rolls or less;
- If after two rolls some pins are still standing, the score of the turn is the number of pins that have been knocked down during this turn;
- If every pin gets knocked down in two rolls, this is a **spare**, and the score of the turn is ten plus the number of pins knocked down during the next roll (during next turn);
- If every pin gets knocked down with the first roll, this is a **strike**, the turn is over and the score is ten plus the number of pins knocked down during the next two rolls;

- If a spare or strike occurs during the last turn, one or two bonus rolls are respectively accorded, with ten new pins. These bonus rolls are part of the same turn, and are not cumulative should a spare or strike occur a second time. Bonus rolls count in the score of the final turn;
- The game score is the sum of every turn's score.

4.3 Steps

If you want to do this kata alone, here are the steps to follow:

4.3.1 Initialization

Steps:

- Create the *Game* Class in `src/main/java/Game.java` and the *GameTest* class in `src/test/java/GameTest.java`.

4.3.2 Test 1: Gutter Game

Steps:

- Add a test which checks the score of a game when we only roll 0. You must code your test in the simplest way possible. Any refactoring will happen later;
- Try to call your test. It should fail, as you did not create the methods used to roll and to get the score;
- Declare those needed methods in the *Game* class (they just need to compile).
- Try to call your test;
- As long as the test fails, add or modify behaviors of the method that gets the score;
- Now, refactor code that can be refactored (most likely in `GameTest.java`).

4.3.3 Test 2: Only Ones

Steps:

- Add another test. This time you want a test that checks the score of a game where every roll knocks one pin down (the game score should be 20);
- Try to call your tests. The new test should fail because you didn't implement correctly the method to throw a ball;
- Modify this method to pass the new test in the simplest way you can imagine;
- Try to pass your tests and correct your method until you pass all the tests;
- Refactor your code.

4.3.4 Test 3: One Spare

Steps:

- Add a test checking the score of a game which contains one spare;
- Update your class and your different methods to pass all the tests;
- Refactor your code.

4.3.5 Test 4: One Strike

Steps:

- Add a test checking the score of a game which contains one strike;
- Update your class and your different methods to pass all the tests;
- Refactor your code.

4.3.6 Test 5: Only Spares

Steps:

- Add a test checking the score of a game only containing spares (if every throw strikes 5 pins the score should be 150);
- Update your class and your different methods to pass all the tests;
- Refactor your code.

4.3.7 Test 6: Perfect Game

Steps:

- Finally, add a test checking the score of a perfect game (the score should be 300);
- Update your class and your different methods to pass all the tests;
- Refactor your code.

You should now have a nice and clean code with multiple tests.

Please note that we have added a consequence for failure.