Programming in Java (23/24) – Day 6: Constants, JavaDoc, interfaces

1 Constants

Programs use data, lots of data. Data is stored in variables of simple and complex types (the latter being pointers that point to objects in the memory, in the heap). We have used a lot of variables since we started our journey into programming.

Sometimes, we know that a piece of data is not going to change. Maybe it is a physical constant like the speed of light, maybe it is a mathematical constant like π, or maybe it something else. Any such piece of data in your program that is known to not change, is called a *constant*.

You do not want your program to allow any section of the code to modify your constants, even by mistake. This is achieved by using the keyword **final**. This keyword tells Java that the value of a variable must never change from the moment it is initialised until the program ends. See an example:

```
1    public static final double PI = 3.14159265359;
```

By convention, constants are written all in capital letters. Multi-word constants use underscores, as in:

```
1    public static final int SPEED_OF_LIGHT = 299792458;
```

You remember that we said that static *fields* should be used sparingly. The one use case that is an exception to this rule are *constants*. The keyword **final** is used to declare an identifier as constant, and therefore both keywords (**static** and **final**) are frequently found together when applied to constants: **final** ensures the value cannot be changed while **static** ensures there is only one field (one "box" in the memory) for the whole class rather than one duplication in every object of the class (many "boxes" with the same name and type and, for constants, also the same content).

Constant fields can be made **public**, as there is no risk of anyone modifying their values producing some undesirable side effect.[1] If you ever make a field **public** in a class that has methods, make sure you make it **static** and (especially) **final**.

[1] This is true only for the content of the field itself, but not for anything that it may point to on the heap: for `public static final Point ORIGIN = new Point(0,0);` in a class `CoordinateSystem`, anyone could still call method `CoordinateSystem.ORIGIN.moveTo(new Point(4,2))` and thereby change what can be seen through the constant `ORIGIN`. Thus, in a `public static final` field, you should not store a pointer of a complex type with mutator methods. The same also holds for arrays. An example of a class without mutators, so with values safe to store in public constants, is the class `String`.

If a field is `public`, it SHOULD be `static` and `final`.

The only exception to this rule is for classes without methods. Such a class can be used as a way of interchanging several pieces of data bundled together (as a return value, for example).

You can use `final` also to declare a constant in a method: `final int EGGS_PER_PACK = 8;`

2 Classes and their "public interface"

Classes in Java have member fields and member methods. The fields are variables stored inside objects of that class, and contain information about the object. For example, they can represent the age and name of a person or the coordinates of a point. The set of all member fields of an object is usually called the *state* of the object: they describe how the object *is* at the current moment.

On the other hand, methods (at least public ones) describe what the object can *do*, not what it is. For example, a person can say something or a postal service can deliver a letter to an address. The set of (public) methods of a class is usually called the *behaviour* of the class.

The behaviour of a class is more important for a programmer than its state. The state is an implementation detail that can change without affecting its behaviour, and is usually not important for the task at hand: this is one of the reasons why fields should be `private` in most situations. If I want to use an object of type `PostalService` to transport a letter to its destination, I do not really mind if the postal service internally uses some object of type `Train`, or a `Lorry`, or a `CargoAirCraft`, or something else: I just want the method to do what it is supposed to do. I do not mind either if a future version of `PostalService.send(Letter, Address)` is implemented internally in a different way or with different fields. Actually, it is quite common that the implementation changes over time (to make it faster, or to use less memory, or to fix bugs) so I should not worry about it; I should assume it will happen sooner or later.

This is why the behaviour of a class is more important than its state. There are two consequences of this. First, the state should always be private (we knew that). Second, the behaviour must be easy to know and understand without the need to read the whole code of a class.

Let us revisit class Point from Day 5:

```java
public class Point {
    private int x;
    private int y;
    public Point(int x, int y) {
        this.x = x;
        this.y = y;
    }
    public int getX() {
        return x;
    }
    public int getY() {
        return y;
    }
    public void moveTo(Point remote) {
        this.x = remote.x;
        this.y = remote.y;
    }
}
```

All that other programmers will really care about is what they can do with the *public interface* of class `Point` – the members of the class that are labelled as **`public`** and will (should) stay available also in updated versions of the same class.[2] The public interface of class `Point` looks like this:

```java
public class Point {


    public Point(int  , int  )



    public int getX()


    public int getY()


    public void moveTo(Point       )



}
```

[2] In fact, Java has a very related notion of "interface types" that builds on the concept of the public interface of a class. We will learn more about interface types today as well.

Everything else may be changed by the author of class `Point`, without harming the functionality of other classes that use class `Point`. That is, as long as the behaviour of class `Point` (i.e., what its methods do) stays the same!

This is why other programmers will be very keen to know what the behaviour of class `Point` is, that is, what they can expect from the methods of the class. The way to tell them is with the help of dedicated *documentation comments* that can be used to generate documentation for the public interface of a class. In the next section you will learn how you can write documentation comments for your own classes.

3 Documenting your code with JavaDoc

You have already seen how the Java library is completely documented online. All classes are described on a web page that explains what they are and what they do (i.e., their methods, explaining the parameters), and this information is online on the Java API documentation or JavaDoc. You can find the JavaDoc easily by typing "Java API" on your favourite search engine.

The good news is that it is easy to document your own classes in the same professional way as the core Java library. In order to do so, you just need to write the comments of your code in a special way and then use the program `javadoc`.

3.1 How to write comments

A bit of history about comments: in the C programming language, one of the "ancestors" of Java, programmers could write comments over many lines by starting them with `/*`, then writing one or more lines of text that should be ignored by the compiler (but not by other programmers), and then finishing the comment with `*/`. We can do the same in Java, as an alternative to the *short comments* that start with `//` to label the rest of the line as a comment that we saw at the very beginning of our journey. Comments using `/*` and `*/` are also known as *long comments*.

For writing comments aimed at readers who just want to *use* our classes, but do not really care how they work internally, Java provides a variant of long comments. They are called *JavaDoc comments* or *documentation comments*. JavaDoc comments start with `/**`, end with `*/`, and can span several lines. JavaDoc comments can also *tag* additional information regarding parameters or return values. Look at this example:

```
1  /**
2   * Implementation of the geometrical concept of a point in two dimensions.
3   * Provides methods to access the coordinates as well as to move a point.
4   */
5  public class Point {
6      private int x;
7      private int y;
8
9      /**
10      * Constructs a new Point with the given coordinates.
11      *
12      * @param x the x coordinate of the new Point
```

```
13      * @param y the y coordinate of the new Point
14      */
15     public Point(int x, int y) {
16         this.x = x;
17         this.y = y;
18     }
19
20     /**
21      * Getter for the x coordinate of this Point.
22      *
23      * @return the x coordinate of this Point
24      */
25     public int getX() {
26         return x;
27     }
28
29     /**
30      * Getter for the y coordinate of this Point.
31      *
32      * @return the y coordinate of this Point
33      */
34     public int getY() {
35         return y;
36     }
37
38     /**
39      * Changes the coordinates of this Point to be the same as those of remote.
40      *
41      * @param remote the Point to which we want to move this Point
42      */
43     public void moveTo(Point remote) {
44         this.x = remote.x;
45         this.y = remote.y;
46     }
47 }
```

There are three important things to notice in this example:

- All documentation comments start with `/**` and end with `*/`. It is customary to make each line in the comment start with a star too.
- Parameters are documented with the tag `@param`, followed by the name of the parameter, followed by the description.
- Return values are documented with the tag `@return`, followed by the description of what is returned.

Documentation comments should explain the functionality of your classes and methods in a short and clear way. Documentation comments should not explain the internal implementation details unless it really makes a difference to other programmers using those classes or methods. In other words, documentation comments should be as clear as possible, and as short as possible, in that order of importance.[3]

3.2 How to create the online documentation

Creating the web pages with the documentation of your classes and methods is very simple. Assuming that you have all your classes documented (i.e., with proper JavaDoc comments) in a folder called `src` and want to put all the web pages in a folder called `www`, you just need to run the following command on Mac or Linux

```
> javadoc path/to/src/*java -d path/to/www/
```

or, on Windows,

```
> javadoc path\to\src\*java -d path\to\www\
```

The javadoc tool reads the JavaDoc comments in your classes and generates all the web pages, HTML files, CSS style sheets, links, etc, for you on the `www` folder. The folder will contain a file called `index.html` as a starting point.

Usually programmers who use your classes will look *only* at these web pages, but not at the source code of your classes. In fact, your source code may not even be available to other programmers!

4 String.equals() vs ==

This section builds on Day 3, when we discussed the difference between simple and complex data types. It is meant to make you aware of a possible "pitfall" of Java that can confuse programmers who already have experience with other programming languages (e.g., Python or C++).

We may be tempted to check whether two Strings are equal using "==". In Java we have to be careful to use the method `.equals()` instead. Otherwise we may get funny results like in the following simple code:

```
1        java.util.Scanner scan = new java.util.Scanner(System.in);
2        System.out.print("Enter a string: ");
3        String str1 = scan.next();
4        System.out.print("Enter another string: ");
5        String str2 = scan.next();
6        System.out.println("Are '" + str1 + "' and '" + str2 + "' the same?");
7        System.out.println("Using ==: " + (str1 == str2));
8        System.out.println("Using .equals(): " + (str1.equals(str2)));
```

which produces the following output:

[3] The Java documentation itself is a good inspiration of what good documentation comments should look like. Use it as an inspiration as well as a source of information.

```
Enter a string: This is a String
Enter another string: This is a String
Are 'This is a String' and 'This is a String' the same?
Using ==: false
Using .equals(): true
```

This is because the equality operator "==" works to compare *values* only for simple types in Java, looking only into their little "boxes". When "==" is used to compare objects like Strings, the operator again compares the contents of the little boxes, i.e., the pointers rather than the contents of the objects (see Figure 1). The method `.equals()`, on the other hand, can be used to compare the content of the objects.

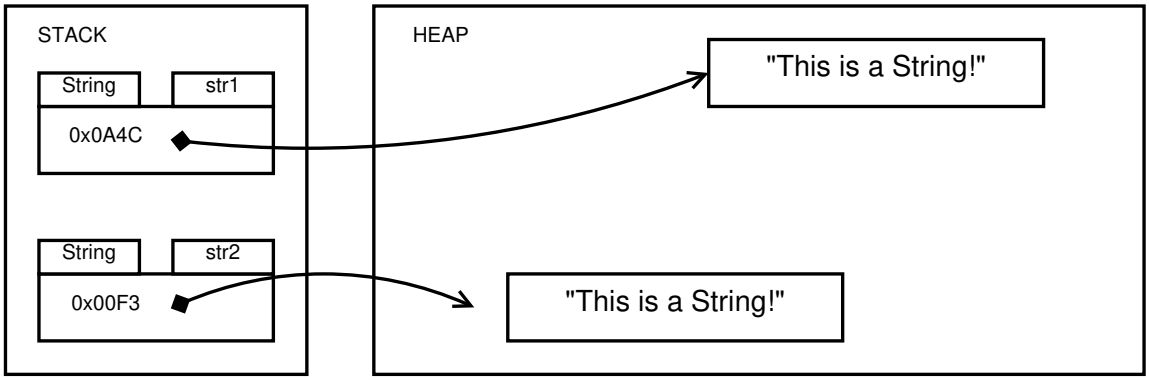

Figure 1: The operator "==" is meant to compare values of simple types. When used to compare complex types, it compares just the pointers. Strings `str1` and `str2` have the same content but their pointers are different, i.e., they point to different memory addresses.

As a rule of thumb, **always use the method .equals() for comparing Strings** or any other values of complex types in Java, unless you really want to check that both pointers point to the same memory address (that is not usually the goal).[4]

Note. For some classes, the effect of `.equals()` is the same as "==", i.e., it compares the pointers and not the content of the objects. This depends on the class and its semantics (i.e., meaning). For instance, we usually think that two `Strings` or two `Integers` (class, not simple type) are the same if they have the same value, but we do not think of two `Persons` to be same even if they have the same name. We will learn more about this when we learn about class inheritance but, for now, it suffices to say that objects should always be compared with `equals()` and not with "==". The operator "==" should only be used for simple types (also known as *primitive* types). Using the comparison with `equals()` for objects has the benefit that we compare in the way that the author of the class intended, which may or may not be by the address that the pointer is referencing. For class `String`, the authors decidedly did not want to compare just pointers.

[4]For completeness: for arrays, Java provides an `.equals()` method as well, but in the case of arrays this method is implemented via "==", so does not look into the contents of the array either.

5 A new type of loop: `do ... while`

With a loop you can perform the same operations over and over again. However, if the condition for the loop is never true, a **while** loop will never run; and sometimes you need it to run at least once, like in this case:

```
1        java.util.Scanner scan = new java.util.Scanner(System.in);
2        System.out.println("Enter the names of your friends. Finish by typing END.");
3        String name = scan.next();
4        System.out.println(name + ": friend");
5        while (!name.equals("END")) {
6            name = scan.next();
7            System.out.println(name + ": friend");
8        }
```

In this brief example you need to repeat the same code to read input from the user in two different places. If the only way to do loops was using **while**, this repetition could not be avoided (and in a less trivial program, with intelligent error-checking and the like, this could be a lot of repetition!). Fortunately, in Java we can use the **do** ... **while** loop:

```
1        java.util.Scanner scan = new java.util.Scanner(System.in);
2        System.out.println("Enter the names of your friends. Finish by typing END.");
3        String name;
4        do {
5            name = scan.next();
6            System.out.println(name + ": friend");
7        } while (!name.equals("END"));
```

Note that you must declare the variable name before the loop starts. Otherwise, you cannot make checks on it at the end of the loop because it would have been out of scope and forgotten by the computer.

Remember, repeated code is usually a bad sign: a symptom of poor design and/or poor programming. When you have the same code in two different places, and make a change in one of them, it is very easy to forget to change the other too... a common source of bugs for careless programmers. If you notice you have repeated code in your program, think twice about a better way of doing things, without repetition. Remember the DRY principle: **Don't Repeat Yourself**. Keep it in mind at all times.

Now you know three kinds of loops: **for** loops, **while** loops, and **do** ... **while** loops. This raises the question: when should you use which kind of loops? The following can provide a good rule of thumb:

- Use a **for** loop if you can express in advance how many iterations the loop will make. Usually this will involve an explicit variable as a loop counter that is incremented or decremented against a bound.
- Otherwise: use a **do** ... **while** loop if you know that you need to execute the loop body at least once. This is often the case for reading a sequence of input values (or repeated attempts at entering a password).

- In all the other cases: use a standard **while** loop. A typical example would be a loop that repeats until two values in an approximation are close enough.

Exercise

Make a class that implements a method that reads a list of marks between 0 and 100 from the user, one per line, and stops when the user introduces a -1. The program should output at the end (and only at the end) how many marks there were in total, how many were distinctions (70–100), how many were passes (50–69), how many failed (0–49), and how many were invalid (e.g., 150 or -3). Write exactly one call to method `nextInt()` from class `java.util.Scanner` in your code.

6 Interface types

Recall the notion of the *public interface* of a class, describing the publicly available members of a class (in particular **public** methods and constructors). The idea was that other programmers using our class need to know how they can work with our class as a helper for their code. However, private implementation details of our class (such as **private** member variables of our class) are of no importance to them.

For greater flexibility, we want to be able to use different classes interchangeably, as long as they offer a certain public interface to other programmers. This is where in Java *interface types* or just *interfaces* come in. An interface in Java is just a way of showing and explaining the behaviour that your class needs to have. An interface does not contain any information at all about the implementation or the state of the objects of your class.[5] It only describes some (sometimes all) of the **public** methods of your class. Let's see an example:

[5] Strictly speaking, since Java 8 it is also possible to provide " **default** " implementations for methods inside of interfaces. However, these default implementations cannot use any member variables with the state of the objects (there are none in an interface), so they are independent from specific implementation decisions for the state of the objects. They tend to be used sparingly – often enough, we *need* the access to the state of the objects to implement the methods in the interface – and can be replaced ("overridden") in classes that implement the interface. Since Java 8, interfaces can also have **static** methods.

```java
/**
 * A person is defined by movement (as opposed to plants)
 * and by making verbal statements (as opposed to animals).
 */
public interface Person {
    /**
     * Move a distance in a straight line, given in metres.
     * @param distance distance in metres to move
     */
    void move(int distance);

    /**
     * Make a verbal statement, printing it on screen.
     * It may or may not be a perfect transcription.
     * @param message the message to use for the verbal statement
     */
    void makeVerbalStatement(String message);
}
```

Listing 1: Person.java

As you can see, the definition of an interface is very similar to the definition of a class, it is even defined in a `.java` file (here: `Person.java`). The difference (and it is a big one!) is that there are no implementation details at all in the definition of the interface. It only declares the methods: their names, and their parameters with types. Every other detail about the class is left for the class to be defined.

There is something you may have noticed: there is no need to say that methods are **`public`** because methods defined in an interface are **`public`** by definition. Note the documentation comments for the interface itself and for its methods. They have two audiences:

- Programmers who use the interface, e.g., as a parameter type for their methods, need to know what behaviours they can expect from the methods listed in the interface. But they do not need to worry about how the methods in the interface could be implemented.
- Programmers who write a class that implements the interface need to know what their public methods listed in the interface must do (and need to think about how to implement a class with these methods). But they do not need to worry about how other programmers might use the methods.

A class that implements all the methods defined in an interface can use the reserved keyword **`implements`** to mark it. This will tell the Java compiler that the class **must** have the methods listed in the interface; if it does not, the compiler will complain with an error:

`class is not abstract and does not override abstract method...`

Let's see two examples of classes that implement `Person`:

```java
/**
 * An AdultPerson is a Person that can move at different speeds
 * and that has full vocabulary.
 */
public class AdultPerson implements Person {

    /** indicates where the person is situated */
    private int situation;

    /** the energy level of the person, must not go below 0 */
    private int energy;

    /** the person's left leg, must not be null */
    private Leg leftLeg;

    /** the person's right leg, must not be null */
    private Leg rightLeg;

    // ...constructors and other methods come here...

    /**
     * Move a distance in a straight line, given in metres
     *
     * @param the distance in metres to move
     */
    public void move(int distance) {
        if (energy > 0 && leftLeg.isHealthy() && rightLeg.isHealthy()) {
            run(distance);
        } else {
            walk(distance);
        }
    }

    /**
     * Make a verbal statement of message "as is".
     *
     * @param message the message to be used for the verbal statement
     */
    public void makeVerbalStatement(String message) {
        System.out.println(message);
    }

    /**
     * Move quickly by a given distance in metres, expending one energy unit.
     * Method may only be used if at least one energy unit is available.
```

```
     *
     * @param distance the distance in metres
     */
    private void run(int distance) {
        situation = situation + distance;
        energy--;
    }

    /**
     * Move at regular pace by a given distance in metres.
     * No energy is expended.
     *
     * @param distance the distance in metres
     */
    private void walk(int distance) {
        for (int i = 0; i < distance; i++) {
            situation++;
        }
    }
}
```

Listing 2: Implementation 1 in AdultPerson.java

```
/**
 * A KidPerson is a Person that has a limited vocabulary and
 * that moves slowly.
 */
public class KidPerson implements Person {
    /** the kid's position */
    private int position;

    /** the kid's brain, shall never be null */
    private Brain brain;

    // ...constructors and other methods come here...

    /**
     * Move a distance in a straight line, given in metres
     * @param distance the distance in metres to move
     */
    public void move(int distance) {
        crawl(distance);
    }

    /**
     * Makes a verbal statement based on the understood words in message.
     * @param message the message to use for the verbal statement
```

```
25      */
26     public void makeVerbalStatement(String message) {
27         String finalMsg = getUnderstoodWords(message);
28         System.out.println(finalMsg);
29     }
30 
31     private void crawl(int distance) {
32         for (int i = 0; i < distance; i++) {
33             position++;
34             waitALittle();
35         }
36     }
37 
38     private String getUnderstoodWords(String text) {
39         String result = "";
40         String[] words = brain.divideIntoWords(text);
41         for (int i = 0; i < words.length; i++) {
42             if (brain.isKnown(words[i])) { // if not, ignore it
43                 result = result + words[i];
44             }
45         }
46         return result;
47     }
48 }
```

Listing 3: Implementation 2 in KidPerson.java

As you can see, both `AdultPerson` and `KidPerson` implement the interface `Person`. You can read the keyword **`implements`** as "is a"; therefore, `AdultPerson` *is a* `Person`, and `KidPerson` *is a* `Person`. However, both classes implement the interface in different ways:

- `AdultPerson` uses a member field called `situation` while `KidPerson` uses a member field called `position`. This is usually a sign that the two classes have been written by different programmers.
- `KidPerson` only prints part of the message when the method `makeVerbalStatement(String)` is called.
- `KidPerson` moves slower than `AdultPerson`. The latter can move at two different speeds (`run` and `walk`), one of them using more energy than the other. `KidPerson` has unlimited energy.

From the point of view of a programmer who wants to use an object of type `Person`, all these differences may be irrelevant and confusing. The programmer just wants an object that moves and makes verbal statements. Thanks to the definition (and implementation) of Java interfaces, it is perfectly possible to use objects without knowing their internal details. Look at the following code:

```
        Person child = new AdultPerson();
        // move in front of parent
        child.move(10);
        // give the message
        child.makeVerbalStatement("I love you.");
```

This code will work exactly the same with `AdultPerson` and `KidPerson`. Actually, it will work regardless of the specific class that is used as long as it implements the interface `Person`. So we might also write a method

```
    public static void moveAndMakeSpecialStatement(Person child) {
        // move in front of parent
        child.move(10);
        // give the message
        child.makeVerbalStatement("I love you.");
    }
```

It would be up to the user of the method whether they pass a `AdultPerson` or a `KidPerson` (or an object of some other class that implements `Person`) as an argument to the method. To compile this method, the compiler would need to know only about interface `Person`, but not about the classes that implement interface `Person`.

A note about names In Java, the convention is to use short names for interfaces and possibly-longer, more-specific names for the classes that implement them. Interfaces can have names like `Person` or `Employee`, while classes have names like `AdultPerson` and `HumanResourcesEmployee`.

Conclusion Interfaces are the way to say what a class *is* and *does*, without saying anything about *how* the class does it. Hiding the details (i.e., the actual code) keeps the complexity of the program lower, resulting in simpler programs that work better and have less bugs.

In your programs, make an effort to define interfaces for your classes (especially the important ones) and use always the interface instead of the actual class to declare new variables of the type defined by that class.