Day 6: Dynamic data structures (and the static keyword)

1 Static data

There are few Java keywords that we do not know yet. One of them is `static`. This keyword means that the field or method that comes after it does not belong to an object of a class but to the class itself: it is a class member rather than an object member[1].

Why would we be interested in using `static`?

Static fields There is little use for static fields. There are few cases in which some information of our objects should not be stored in the objects themselves but in the class, and it is common that this information is constant. For example, we could store the number of chromosomes of an animal: this information does not change from animal to animal:

```
public class Human {
    private static chromosomeCount;
    private String name;
    private int age;
    // ...
}
```

Every human has a different name and a different age, but the same number of chromosomes. Making it `static` makes it clear to other programmers with little knowledge of biology that the chromosome count is the same for all humans[2].

It also saves a little memory because the variable is stored only once (in the class) rather than many times (once per object). For every name, there is one and only one static variable per class per machine; in other words, there is only one `chromosomeCount` in class `Human` in the whole computer[3].

You can use a static variable to keep a counter of how many objects of a class have been created, as in the example:

```
public class Spy {
    private static int spyCount = 0;
```

[1] The name *static* is a bit confusing at first, because it does not mean that static things do not move but rather that they belong to the class and not its instances (i.e. objects). It may have been better to use a keyword like *class*, but it is already used to define classes themselves.

[2] For the sake of simplicity, we ignore in this example chromosomical conditions such the Down, Patau, Turner, and Klinefelter syndromes.

[3] Being precise, there is only one per *Java Virtual Machine* (JVM). There may be more than one JVM in a physical computer, although this is not relevant in most cases and the "there is one static X per machine" is mostly true.

```
    public Spy(...) {
        spyCount++;
        // ...
    }

    public static int getNumberOfSpies() {
        return spyCount;
    }
    // ...
}
```

Every time the constructor of `Spy` is called (using `new`) the counter `spyCount` is increased; we could also think of a `die()` method that reduces the counter. In any case, it is rarely necessary to keep a count of objects of a class, and therefore static variables are seldom used except as constants.

Static methods Static methods are class methods. They do not need to be called from an object, but can be called directly from the class. You have already seen some examples as `Math.random()` and the widely used `Integer.parseInt(String)`.

Static methods can only access `static` fields. As they are run from the *class*, they can only access data that is *in the class* and not *in the objects*. You can think of it in the following way: the class does not know where the objects are stored in the memory, so all instance data (i.e. data in the objects) cannot be accessed. Class data and class methods, however, can be accessed from anywhere, just using the name of the class (as in `Integer.parseInt(String)`).

Rules of thumb It is important to understand what `static` means in Java, because you will find this keyword quite often in your life as a Java programmer. On the other hand, it is a keyword that you will not use that much; and when you do, it will be almost always in one of three situations as explained below.

In other words, if you had to remember just four things about the keyword `static`, remember these four rules:

- Use `static` *sparingly*. If you use it often, you are probably doing it wrong. Thing again about your program, and look for ways to get rid of your static state (i.e. fields/variables).
- If you have static fields, they should be *constants* and they should be `public`. One such example is `Math.PI`: its value is always the same 3,14159...
- If you have static methods, they should be *pure functions*. This means they should not have side effects, and they should return a value that depends only on their arguments. One such example is `Integer.parseInt(String)`.
- If you have a `main` method, it must be static as there is only one per class (at most).

What is the `main` method? I am glad you asked. The `main` method is the entry point for your Java application.

2 The main method

As we leave Groovy and Java Decaf behind and move into full-flavour Java, we need a way to launch our program: a starting point. It is all too well to have a lot of Java classes, each of them in their own `.java` file, but this is of little use if we cannot start our program from the command like (or clicking on an icon, for that matter).

In Groovy and Java Decaf, we did not need an entry point. The programs or scripts were their own entry point. For example, we only needed to write `groovy myScript.groovy` and Groovy would start the execution. The situation is slightly more complex in Java because all files are classes, not scripts. We need to have an entry point, a place in the code where Java knows that the execution of the program can start. This entry point is the `main` method. The main method is a method like any other, and every class can have one. It looks like this:

```
public static void main(String[] args) {
  // Here be the code to launch the program
}
```

If a class has a main method, it can be run from the operating system by typing:

```
> java myClass
```

assuming that there is a `myClass.class` file. The signature of the `main` method is a bit long, but we have already learnt what each of its elements means:

- It must be `public`, because otherwise it cannot be accessed from outside the class (and how would the program launch then?).
- It must be `static`, because it pertains to the class, not to each of its instances (i.e. objects).
- It returns `void`. This is a method that is never called by other classes, so a return type is unnecessary.
- The only parameter is a 1-D array of Strings, called `args` (short for arguments). Actually the name is not important. The elements in the array are the parameters passed in the command line, if any. For example, if you run a program with a line like `java myClass www.google.com 80 true`, the first element of `args` (i.e. args[0]) will be "www.google.com", the second element (i.e. args[1]) will be "80", and the third one (i.e. args[2]) will be "true". All of them are Strings, if you want to use these parameters with a different type you need to parse them.

Liberating your main method from static restrictions

As static code does not understand anything about instance (i.e. non-static) methods or data, you will not be able to use your instance data in your main method. This seems restricting at first, as in the following example: imagine that we want to start a program from a BitTorrentDownloader class, and we want to give the name of the host and the port from the command line

```
public class BitTorrentDownloader {
    private String host;
    private int port;
```

```
    public BitTorrentDownloader(String host, int port) {
        this.host = host;
        this.port = port;
    }

    public static void main(String[] args) {
        host = args[0];
        port = Integer.parseInt(args[1]);
        // with the fields initialised,
        // launching code comes here...
        // ...
    }
}
```

When you try to compile this program, java will complain with the following error:

```
non-static variables cannot be referenced from a static context
host = args[0];
^
port = Integer.parseInt(args[1]);
^
```

One (bad) solution is to make `host` and `port` static, but that means that there can only be one of each. This is usually a bad idea in the long run. Maybe in a future version you will like several BitTorrentDownloaders working in parallel, or maybe you would to launch many at the same time and put them in a queue. All of this would be impossible if the fields were static. Remember: `static` means "only one per class", and programmers usually like to have as many as possible of everything, just in case. As a rule of thumb, **never make anything "static" unless it is absolutely necessary**.

There is another (much better) solution: to instantiate the class inside its own main method, and then run all the code from a non-static launching method. This method is commonly called `run()` or `launch()`. See the example:

```
public class BitTorrentDownloader {
    private String host;
    private int port;

    public BitTorrentDownloader(String host, int port) {
        this.host = host;
        this.port = port;
    }

    public static void main(String[] args) {
        // 1. Parameter analysis / processing
        String host = args[0];
        int port = Integer.parseInt(args[1]);
```

```
        // 2. Object construction / building of main data structures
        BitTorrentDownloader downloader = new BitTorrentDownloader(host, port);
        // 3. Execution of the program
        downloader.launch();
    }

    private void launch() {
        // with the fields initialised,
        // launching code comes here...
        // ...
    }
}
```

As you can see the main method is very simple and has only three parts: parameter analysis, creation of the main structures, and starting the execution of the program. It is a good idea to follow this structure in every main method. This will keep your main methods simple and to the point. If you notice that your main methods are getting longer than a few lines, make a pause, take a deep breath, and think things through again.

3 More differences between Groovy and Java

3.1 String.equals() != ==

In Groovy we have become used to comparing String using "==". In Java we have to be careful and use the method `.equals()`. Otherwise we may get funny results like in the following simple code:

```
System.out.print("Enter a string: ");
String str1 = System.console().readLine();
System.out.print("Enter another string: ");
String str2 = System.console().readLine();
System.out.println("Are '" + str1 + "' and '" + str2 + "' the same?");
System.out.println("Using ==: " + (str1 == str2));
System.out.println("Using .equals(): " + (str1.equals(str2)));
```

that produces the following output:

```
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 "==" only works with simple types in Java, with little "boxes". When it is used to compare objects like Strings, it compares the pointers not 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.

As a rule of thumb, **always use the method .equals() when comparing Strings** or any other complex types in Java.

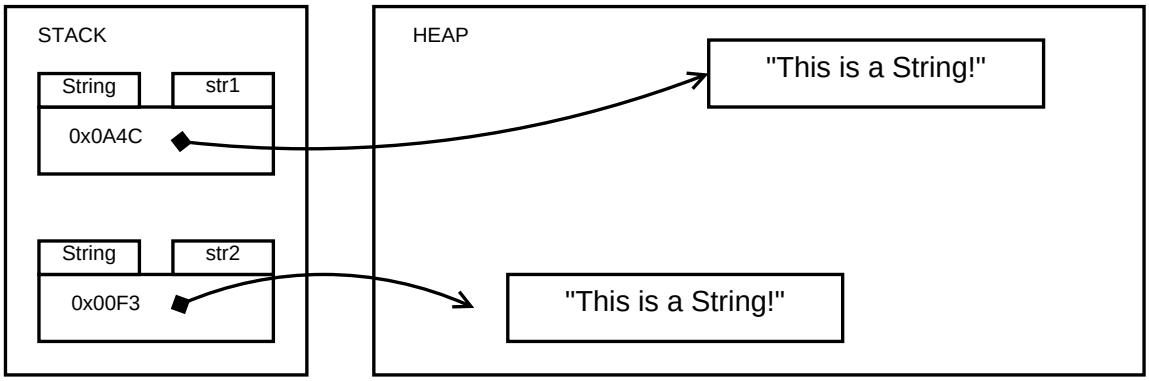

Figure 1: The operator "==" compares simple types. When used to compare complex types, it compares just the pointers. Strings str1 and str2 have the same content but their pointers are different, i.e. they point to different memory addresses.

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. 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 `Person` 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).

3.2 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:

```
System.out.println("Enter the names of your friends. Finish by typing END.");
String name = System.console().readLine();
System.out.println(name + ": friend");
while (!name.equals("END")) {
    System.out.println(name + ": friend";
    name = System.console().readLine();
}
```

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:

```
System.out.println("Enter the names of your friends. Finish by typing END.");
String name;
do {
```

```
        name = System.console().readLine();
        System.out.println(name + ": friend");
    } while (!name.equals("END"));
```

Note that you must declare the string `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.

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). Use `readLine()` exactly once.

4 Beyond arrays: lists, stacks, queues

We have seen what to do when we want to store a lot of data of the same type: we create an array, and then store all elements side-by-side in memory. Arrays are convenient because they store a lot of information of the same type without the need to use a lot of variables. Their main disadvantage is that we need to now the lenght of the array in advance and it cannot be changed afterwards.

Most applications in the real world, however, do not know in advance how much data they will need: a hospital does not know how many patient records it will need every day, a car manufacturer does not know in advance how many employees it will have or how many cars it will be producing at any given week. Even if they knew, these desired characteristics of the program —called *requirements*— may change over the life of the program, e.g. : maybe the hospital knows it only has ninety beds, but five years later they receive funding from the Government to extend to another building and duplicate their capacity. A real program needs to be able to handle arbitrary amounts of data.

This is done by means of *dynamic* data structures. The most basic examples are lists. Stacks and queues are special types of lists.

4.1 Lists

Dynamic lists are composed of a sequence of complex types in memory, linked by their pointers. In other words, every element in a dynamic list contains one element of information of a certain type (which may be complex) and a pointer to the next element. Let's use the hospital above to see an example. For the sake of simplicity, we can assume that the hospital only stores three pieces of data about each patient: name, age, and illness. The file `Patient.java` may be similar to:

```
public class Patient {
    private String name;
    private int age;
    private String illness;
    // methods like constructors, getters
    //   and setters come here...
}
```

And the hospital program (a different file to `Patient.java`) could have an array of patients:

```
public class HospitalManager {
    private Patient[] patientArray = new Patient[90];

    // methods here to add or remove patients, etc
}
```

But, as we already know, this is limited because the hospital may have at a later point more than 90 patients. We can overcome this limitation by using a dynamic data structure, a structure that can change size after creation, e.g. a linked list. Our list will be composed of objects of a slightly modified class `Patient`:

```
public class Patient {
    private String name;
    private int age;
    private String illness;
    private Patient nextPatient;

    public Patient(String name, int age, String illness) {
        this.name = name;
        this.age = age;
        this.illness = illness;
        this.nextPatient = null; // Pointer to another patient!
    }
    //  other methods come here, including getters / setters
}
```

The only difference (but it is important!) is the pointer to another `Patient`. A linked list of patients is nothing more than a sequence of patients in which each patient links to the next one, and the last one points to `null`. The starting point of the list will appear in the container class as before, but instead of being a pointer to an array it will be a pointer to the first element in the linked list:

```
public class HospitalManager {
    private Patient firstPatient = null;

    //  other methods come here...
}
```

Operations on linked lists New patients can be added to the list by following the linked list of patients until the end, and then adding the new patient there, as illustrated by the following method:

```
01    // this is a member method of class HospitalManager
02    public void addPatient(Patient newPatient) {
03        Patient current = this.firstPatient;
04        while (current.getNextPatient() != null) {
05            // this means we are not yet at the end of the list
06            current = current.getNextPatient();
07        }
08        // at this point, current points to the last patient
09        current.setNextPatient(newPatient);
10    }
```

The method `addPatient(Patient)` is very simple. It goes through the whole list of patients checking whether the current patient is the last one in the list (`current.getNextPatient() != null`). If it is, it adds the last patient at the end; if it is not, it moves to the next patient and tries again (`current = current.getNextPatient()`).

But beware! What happens if the list is empty? If the list is empty, we will get a `NullPointerException` on line 04 when we try to call method `getNextPatient()` on a null pointer (`current`).

In order to avoid this case, we must check first whether the list is empty before we add the new patient. If it is, we just make the new patient the first patient.

The list of patients is initialised in the main program with a first patient, as shown in the code below (see also Figure 2):

```
01    // this is a member method of class HospitalManager
02    public void addPatient(Patient newPatient) {
03          if (firstPatient = null) {
04              firstPatient = newPatient;
05              return;
06          }
07        Patient current = firstPatient;
08        while (current.getNextPatient() != null) {
09            // the rest of the method does not change...
```

From then on, new patients can be added by calling the method `addPatient(Patient)` at any point.

```
Patient yetAnotherPatient = new Patient("Mary", 66, "Meningitis");
hospital.addPatient(yetAnotherPatient);
```

Patients can also be removed easily from the list. The program must be careful not to leave any pointers loose to prevent information loss. For example, a method to remove patients by name would look like this (see also Figures 3, 4, and 5):

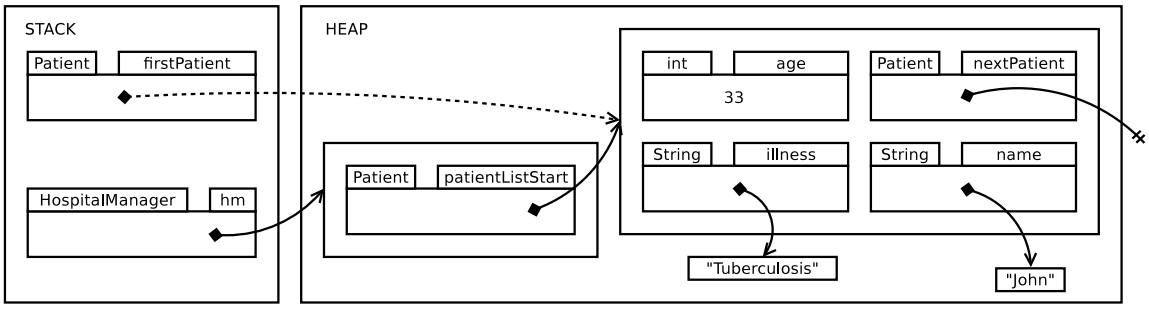

Figure 2: In order to initialise the list, we create a Patient and point the initial pointer (patientList-Start) to it. At the end of the method, the pointer firstPatient is forgotten but the list remains linked from patientListStart.

```
01    // this is a member method of class HospitalManager
02    // returns true if the patient was found and removed, false otherwise
03    public boolean deletePatient(String name) {
04        if (firstPatient == null) {
05            // list is empty, nothing to remove
06            return false;
07        }
08        if (firstPatient.getName().equals(name)) {
09            // first patient in the list must be removed
10            firstPatient = firstPatient.getNextPatient();
11            return true;
12        }
13        Patient current = firstPatient;
14        while (current.getNextPatient() != null) {
15            if (current.getNextPatient().getName().equals(name)) {
16                // We found it! It is the next one!
17                // Now link this patient to the one after the next
18                current.setNextPatient(current.getNextPatient().getNextPatient());
19                return true;
20            }
21            current = current.getNextPatient();
22        }
23        return false;
24    }
```

Read this method carefully. There are three parts in there:

1. Checking whether the list is empty. If it is, we cannot delete any element.
2. Checking whether the first element is the element we want to delete. If it is, we just move the pointer to the first element in the list to point to the second element in the list (that becomes the new "first").

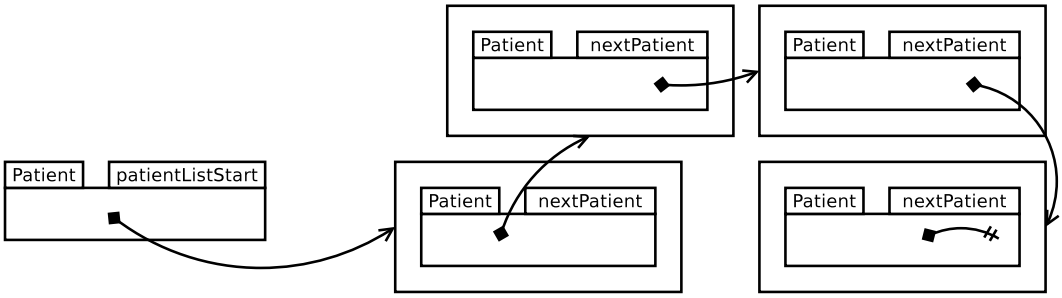

Figure 3: Deletion of elements in a linked list, step one: finding the element we want to delete. Suppose that we want to delete the second elements in the list. In order to delete it from the list of patients, we must link the former patient to the following patient. (Note: simplified diagram.)

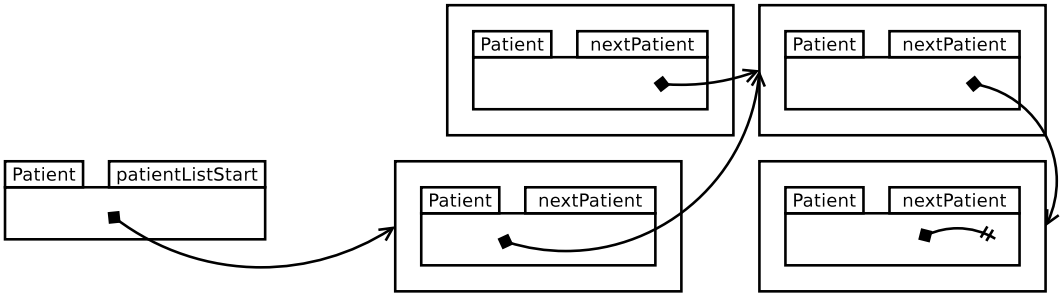

Figure 4: Deletion of elements in a linked list, step two: linking the former patient to the following patient. (Note: simplified diagram.)

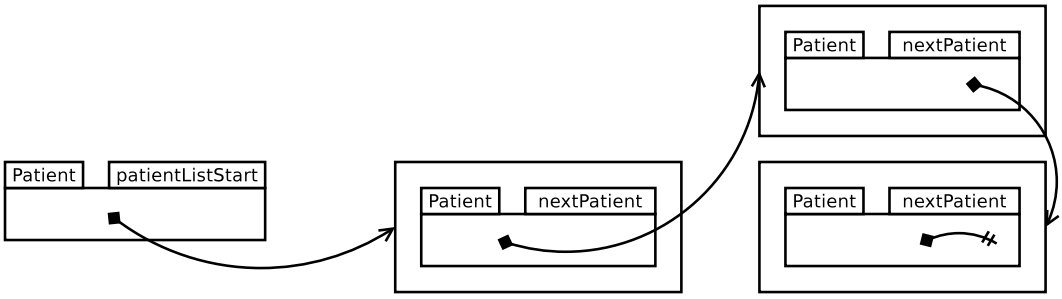

Figure 5: Deletion of elements in a linked list, step three: The deleted patient's memory will be collected automatically by Java's garbage collector. The memory is then free to be used by other objects. (Note: simplified diagram.)

3. The third scenario is where we have to move through the list to delete the right element. We delete an element by making the "next" pointer from the element before point to the element after, making the element unreachable. We have to be careful because we cannot move "backwards" in the list —we can only move forward[4]— so we have to delete an element while `current` points to the element *before* the one we want to delete (line 18).

When we delete an element from a dynamic linked list, it is left "hanging around" without any pointer that links to it. This will be detected automatically by Java and the object will be destroyed in a process called *garbage collection* (Figure 5). The memory freed in this process can then be used to create other objects in the heap. We will learn more about how garbage collection works in the future.

[4] A doubly-linked list, with pointer both to the next and to the previous element, solves this problem. You will create one in the exercises for today.