Day 8: Interfaces (and more data structures)

1 Interfaces

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 of a person or the number of patients in a hospital. 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 hospital can take a new patient. 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 why fields should be `private` in most situations. If I want an object of type `Person` to say something, I do not really mind if the person internally uses some object of type `VocalCord`, or some `Vocoder`, 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 `Person.say(String)` 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. This where *interfaces* come in.

An interface in Java is just a way of showing and explaining the behaviour of your class. An interface does not contain any information at all about the implementation or the state of your class. It only describes some (sometimes all) of the public methods of your class. Let's see an example:

Example of interface

```
/**
 * A person is defined by movement (as opposed to plants)
 * and by speech (as opposed to animals).
 */
public interface Person {
    /**
     * Move a distance in a straight line, given in meters.
     */
    void move(int distance);
```

```
    /**
     * Say something, printing it on screen.
     * It may or may not be a perfect transcription.
     */
    void say(String message);
}
```

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. 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. Every other detail about the class is left for the class to be defined.

There are two more things you may have noticed:

- There is no need to say that methods are `public` because methods defined on an interface are public by definition.
- We have introduce a new kind of comments, that start with `/**`, end with `*/` and can span several lines. These are called *long comments* and are useful to document your code, especially methods, interfaces, and classes. The comments we already knew about, that start with `//` and go to the end of the line —but not to the next line— are called *short comments*.

A class that implements all the methods defined on an interface can use the reserved keyword `implements` to mark it. This will tell the Java compiler that the class **must** have the methods on 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 three examples of classes that implement `Person`:

Implementation 1

```
public class AdultPerson implements Person {
    private int situation;
    private int energy;
    private Leg leftLeg;
    private Leg rightLeg;

    /**
     * Move a distance in a straight line, given in meters
     */
    public void move(int distance) {
          if (rightLeg.isHealthy() && leftLeg.isHealthy()) {
                run(distance);
          } else {
                walk(distance);
          }
    }

    /**
     * Say something
```

```
     */
    public void say(String message) {
          System.out.println(message);
    }

    private void run(int distance) {
          situation = situation + distance;
          energy--;
    }

    private void walk(int distance) {
        for (int i = 0; i < distance; i++) {
                situation++;
          }
    }
    // ...other methods, including constructors, come here...
}
```

Implementation 2

```
public class KidPerson implements Person {
    private int position;
    private Brain brain;

    /**
     * Move a distance in a straight line, given in meters
     */
    public void move(int distance) {
        crawl(distance);
    }

    /**
     * Say something
     */
    public void say(String message) {
          String finalMsg = getUnderstoodWords(message);
          System.out.println(finalMsg);
    }

    private void crawl(int distance) {
        for (int i = 0; i < distance; i++) {
                position++;
                waitALittle();
          }
    }

    private String getUnderstoodWords(String text) {
          String result = "";
```

```
        String[] words = brain.divideIntoWords(text);
        for (int i = 0; i < words.length; i++) {
              if (brain.isKnown(words[i])) {
                  result = result + words[i]; // if not, ignore it
              }
        }
        return result;
    }
    // ... other methods, including constructors, come here...
}
```

As you can see, both `AdultPerson` and `KidPerson` implement the interface `Person`. You can read the keyword `implements` as "is like a"; therefore, `AdultPerson` *is like a* `Person`, and `KidPerson` *is like a* `Person`. However, both class 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 both classes have been coded by different programmers.
- `KidPerson` only prints part of the message when the method `say(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 that want to use an object of type `Person`, all these differences may be irrelevant and confusing. The programmer just wants an object that moves and says things. 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 son = new AdultPerson();
// move in front of mother
son.move(10);
// give the message
son.say("I love you, Mum.");
```

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`.

A note about names In Java, the convention is to use short names for interfaces and 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 `PointerManagerEmployee`.

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.

Exercise

Implement a simple class that executes the former code in its main method. Change the class `AdultPerson` for `KidPerson` and verify that it still compiles and runs.

2 Interfaces and data structures

Interfaces are very important in Java and in any modern object-oriented programming language. Interfaces allow programmers to *hide the implementation* of their classes and expose only their behaviours. This is good because other programmers can use their classes with minimal effort: they only need to understand the behaviours, not the full code. Using other programmers' classes (or your own between different projects) is called *code reuse*, and saves a lot of effort to programmers (and money to their companies). For example, you do not need to create a new `String` class every time you want to write text in your programs: you just use the `String` that comes with Java. And you do not need to understand the code of the class[1], you only need to understand its behaviour: what `charAt(int)` does, what `substring(int,int)` does, etc. This makes your life much easier and is good.

We have seen how dynamic data structures —like dynamic lists— make it possible to store unlimited amounts of data in a program (unlike arrays or discrete variables). Now we are going to see three new special examples of dynamic lists, and we will define them by using interfaces.

2.1 Stacks

We are all familiar with the concept of stacks. When we are putting books on a box, we usually pile them in stacks; each book is placed on top of the next one, and we can only see the last book we have put, i.e. the one on top. When we pile papers on a table we also create stacks.

Stacks are very important in computing, and are used in many different contexts. The method-call *stack* (as opposed to the *heap*) that stores the variables for each method in a program is just one well-known example. Every time you call a method, you create a new level on the stack to store your variables; while you are in the method, you can only "see" the variables at the top level; when you end the method, those variables are forgotten and you see the variables on the former level (see Figure 1).

In a stack, only the last level is accessible: all former levels are hidden until the levels on top are removed. This is why stacks are sometimes called LIFO structures: Last In, First Out. A Stack is defined by the following methods:

push(<type>) puts an element at the top of the stack.

pop() removes the element at the top of the stack and returns it.

peek() returns the element at the top of the stack, but does not remove it from the stack.

isEmpty() returns true if there are no elements on the stack, false otherwise.

[1] The code of all classes in the basic Java library is public. You can find it online. Java is free software under the GPL license.

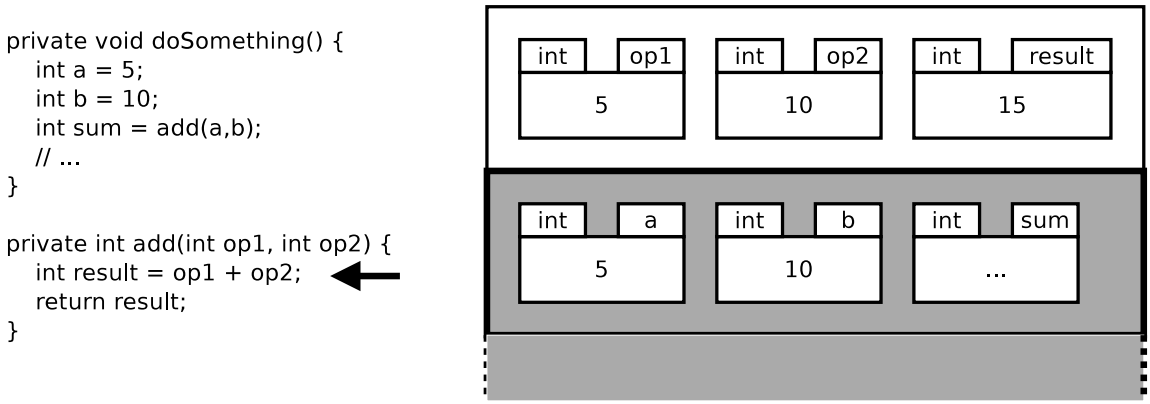

Figure 1: The parameter stack is an example of a stack structure. When a method (add) is called, a new level is put on the stack to store the parameters and the local variables. While in the method, the variables of the other method (a, b, sum) cannot be accessed: they are on a different level of the stack, and only the most recent one is accessible. Note that if those variables were of complex types, their "boxes" would be pointers pointing to objects in the heap. The arrow marks the current statement.

Now have a look at the accompanying Java files. The first file (`StringStack.java`) is the definition of an interface for a stack of strings. The next one (`ArrayStringStack.java`) is one possible implementation of this interface based on arrays. The other two files (`PointerStringStack.java`, `StringStackNode.java`) are another possible implementation using linked lists. Which implementation would you prefer to use? The answer is usually that you do not mind: *as long as the implementation complies with the interface, all implementations are equivalent from your point of view.* You can check that both behave exactly the same, regardless of the number of strings that you *push* on them or the order in which you *push* them. You can use the sample `StringStackScript.java` file provided or make your own test.

Sometimes you do mind: maybe one implementation is faster, or uses less memory, or provides additional methods that are useful for your program. For instance, the implementation of `PointerStringStack` has a public method `size()` that the other implementation does not. Is this a good addition or an unnecessary burden? The array-based implementation is usually faster because it does not need to reserve memory for each element, but sometimes it needs to duplicate the array, which is slow and uses a lot of memory. Is it more important to be faster or to use less memory? There is no general rule that applies to all cases, so a judgement call as a programmer is always needed. However, judgement is acquired though practice and experience, and it cannot be expected from a novice programmer. Therefore, for now, it will suffice to know that the rule of thumb is: *work with interfaces, not implementations; if the class complies with the interface, it is good enough.* You have an example in `StringStackScript.java`: the name of the class is only used to instantiate (i.e. create the object); after that, we only use the name of the interface. We do not need anything else.

2.2 Queues

Queues are, in a way, the opposite of stacks. Stacks are LIFO structures, queues are FIFO structures: First In, First Out (Figure 2). It is the same idea as the queues at the airport or at the supermarket.

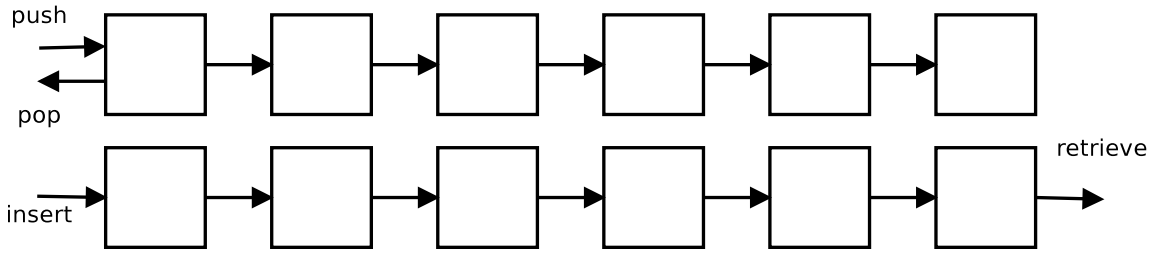

Figure 2: Stacks are LIFO. Queues are FIFO.

The same as stacks, queues are very popular data structures that you can find everywhere in computing: your wi-fi connection has a queue of bytes to send on the air, the web server at `www.bbk.ac.uk` has a queue of requests from web browsers, and your hard disk has a queue of petitions to read or write on it.

Queues are defined by the following interface:

insert(<type>) adds an element to the queue.

retrieve() removes an element from the queue.

It is common that queues provide a method `getSize()` that returns the current size of the queue. Some queues may have a maximum size[2] and provide a method `getMaxSize()` that returns the maximum size possible (so that other classes know in advance, in order to prevent data loss). For example, when you hear in the news that a website has suffered a Distributed Denial of Service attack (DDoS) that means that a lot of computers are issuing simultanenous requests to the same web server until its queue is full and all new requests are ignored.

2.3 Maps

When a list has many elements, it can take a long time to find the element you are looking for. In order to reduce the time needed to delete or find an element in long lists we have *maps*.

A map is a special kind of data structure that links some piece of data, called the *key*, with one or more pieces of data, called the *value* or *values*. The contact agenda in your mobile phone is an example of a map: it maps names to phone numbers. Other examples of maps are address books that link addresses with people, indexes in corporate buildings that link floors with companies, and keyboards that link physical keys with letters and symbols.

A map can link each key with one value or more. Depending on the application, one behaviour may be more appropriate than the other. Maps that link exactly one value to every key are sometimes called *dictionaries*.

[2]Strictly speaking, all queues are limited: even "unbounded" queues cannot be larger than the memory available on the machine.

Maps are useful in computing because we want programs to run fast but lists are usually very long; think of the number of files on your computer; the number of past and future patients in a big hospital; or the number of citizens in the Goverment's tax computers. If a map can link each key (like an ID) to one value, looking for it is very fast. Even if a map links a key to many different values (like the initial of a family name) it reduces the search time considerably. Maps that link one key to many values are sometimes called *hash tables*. This is because they use a *hash* function that reduces the whole search space (e.g. every possible family name in the world) to a reduced one (e.g. the letters of the alphabet). One simple hash function is the modulo operator: it reduces every possible integer in the world to a reduced set of integers. Hash functions are important in many fields of computing —like computer security— and the search of good hash functions is an active field of research.

A map is defined by the following interface:

put(<keytype>, <valuetype>) adds a new element to the map, associated with a key; depending on whether the key has already been used, and whether the map allows more than one value per key, this method can return different types.

get(<keytype>) gets the value associated with that key.

isEmpty() returns true if there are no elements on the stack, false otherwise.