Day 10: Inheritance

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:

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

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

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

You remember that we said that static *fields* should be used sparingly, and mostly (or only) for 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 (a "box") for the whole class rather than one duplication in every object of the class.

You may also remember that we mentioned that constant fields can be made public, as there is no risk of anyone modifying their values producing some undesirable side effect. If you ever make a field public in a class that has methods, make sure you make it static and (especially) final.

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

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

2 Documenting your code with JavaDoc

We 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 documenation 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, we just need to write the comments of our code in a special way and then use the program `javadoc`.

2.1 How to write comments

As we briefly saw when we introduced interfaces, JavaDoc comments start by `/**`, end by `*/`, and can span several lines. JavaDoc comments can also tag additional information regarding parameters or return values, as shown in the following example:

```
/**
 * An StringStack is a dynamic structure that can contain any number
 * of strings.
 *
 * New elements (i.e. Strings) can be put on top of the stack or
 * removed from the top of the stack. Only the element at the top of
 * the stack can be removed at any given time; to access a specific
 * element, all elements on top of it must be removed (popped) first.
 *
 * This interface allows the user to replace some elements of the list
 * even if they are not accessible.
 */
public interface StringStack {
    /**
     * Put an element at the top of the stack.
     *
     * @param newString the new string to be put
     */
    void push(String newString);
    /**
     * Removes the element at the top of the stack and returns it.
     *
     * @return the element at the top of the stack
     */
    String pop();
    /**
     * Replaces every occurrence of one string in the stack
     * for another string.
     *
     * @param oldString the string to be replaced
     * @param newString the new string to replace it with
```

```
     */
    void push(String oldString, String newString);
}
```

There are three important things to notice in this example:

- All 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.

Comments should explain the functionality of your classes and methods in a short and clear way. 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, comments should be as clear as possible, and as short as possible, in that order of importance[1].

2.2 How to create the online documentation

Creating the webpages with the documentation of your classes and methods is very simple. Assuming that you have all your classes documented (i.e. with proper 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:

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

This will generate 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.

3 Inheritance

Repeating code is a bad thing. If the same code is used in more than one place in a program, that is a disaster waiting to happen. Sooner or later, one of the copies of the code will be changed (probably to fix a bug), but not all copies will be changed... and unpredictable things will happen (never good).

We already know one strategy to prevent code repetition: using methods inside a class. By enclosing code inside a method, the programmer can just call the method from any other section of the code and the same instructions will always be executed (probably with different values for the parameters). In this section, we will move one step further to avoid code repetition by using *inheritance* to prevent code repetition *among* classes.

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

3.1 Extending already-existing classes

From all the forms of inheritance, the easiest to understand is the extension of already existing classes. This is done by means of the keyword `extends`. You can think of this keyword as stating a "is a" relationship: you use it to say that objects of your class are also objects from another class... with something on top.

Let's assume that we had a class `Ebook`, part of a program to run an electronic book reader. The class would have several methods, including some like `nextPage()`, `prevPage()`, and maybe even `readAloud(int page)`. When the next version of our electronic book reader comes out, it includes a new feature to regulate the background light of the screen. At this point we have at least two options. First, we can add a new method `setLight(int)` to the `Ebook` class to regulate the background light; however, this method would also be included as part of the software running in all the "plain with no light" e-books. A second possibility, is to create two different classes, `Ebook` and `LightEbook`, each of them being used for a different hardware; but this would lead to having lot of methods repeated in both classes (e.g. `nextPage()`), which is bad.

There is a third, much better possibility. We can make `LightBook` extend `Ebook`, as in the following example code (JavaDoc comments ommited for the sake of brevity):

```
public class LightEbook extends Ebook {
    private int lightLevel = 100;
    public void setLight(int newLevel) {
        this.lightLevel = newLevel;
    }
}
```

You can see that extending a class is very easy: just say that you are extending it (in this example, `extends Ebook`). From that point on, your class will be able to use all the public methods (and some more, see below on Section 3.4) from the class that you are extending without the need to implement them again. In other words, you can call `nextPage()` on a `LightEbook` object exactly in the same way as you would do it on a `Ebook` object.

The base class that is extended is called the *parent class* or *superclass*[2], while the extending class is called a *subclass*. When there are many levels of hierarchy (a class extends a class that extends another class, etc), those classes up in the hierarchy are called *ancestor classes*, while those down in the hiearchy are called *descendant classes*.

Everything is an object! In Java, every class extends class `Object` or descends from a class that extends `Object`. `Object` is a class from the core Java library that provides a few methods that every other class can use. The most commonly used is `equals()`, which compares two objects and returns `true` if they are the same (according to some class-specific rules) and `false` otherwise.

As everything is an `Object` by default, you do not need to write it explicitly (`...extends Object`): Java will do it for you. Programmers must only explicitly say which class they are extending if they are extending a class that is *not* `Object`.

```
public class MyClass {  // no need to say "extends Object" here
```

[2]In this context, "super" is as in Latin, meaning "up"; there are no connotations of big, extra, or more powerful. Remember that the convention in computing is that general classes or interfaces are "up" and more specific or extending classes are "down".

```
    // ...
}
```

Final classes You remember that we could use the keyword `final` to indicate that a variable holds a constant value, i.e. that the variable could not be changed. In a similar way, we can use `final` to tell the compiler that a class cannot be changed either. A class defined as `final` cannot be extended; trying to do so will result in an error at compile time.

3.2 Top-down inheritance

We have already seen that we can use *interfaces* to specify what a class is like or, in other words, what is its behaviour, i.e. the methods that are public and can be used by other classes. For example, we may have an interface `Animal` that was implemented by classes `Dog`, `Human`, and `Horse`. The interface may define methods like `move()`, `makeSound()`, and `breath()`. The three classes would each implement each method.

It is quite possible that the implementations of movement and sounds are quite different, but on the other hand it seems likely that `breath()` is very similar (or exactly the same) in the three classes. As we know, repetition of code is bad: if we have a bug in the `breath()` method, we will be in trouble if we forget to fix it in all classes. It would be much better if we could have the same method in just one place so that each class could use it, in the same way that the code in a method can be used from anywhere in the class.

In cases like this one, it would be great if we could put the real code in the interface, not just the declaration; then, all classes would have it... Well, it turns out we can do something like that in Java. Although we cannot write any code in an interface, we can transform the interface into an *abstract class*, and then make the other classes *extend* it. Let's see an example: assuming that the original interface looked like this (JavaDoc comments ommited for the sake of space):

```
public interface Animal {
     void move(int meters);
     String makeSound();
     void breath(int air);
}
```

...if we implement the method `breath()`, the resulting abstract class could look like this:

```
public abstract class Animal {
     private int oxygen = 0;

     public abstract void move(int meters);
     public abstract String makeSound();
     public void breath(int air) {
         this.oxygen += air / 4;
     }
}
```

As you can see, there are many differences between an interface and an abstract class:

- First and foremost, an interface never contains any actual code, while an abstract class —like any other class— can contain real implementation, i.e. statements that perform some actions.
- Interfaces are *implemented*, while abstract classes are *extended*.
- All methods in an interface are by definition public and abstract (i.e. without actual code), so there is no need to write it down (but it can be done). On the other hand, methods in an abstract class can be private or public, abstract or not, and it must be said explicitly for each of them (see `move(int)`, for example).
- Abstract classes can contain mutable fields. All fields on an interface are by definition static and final (i.e. constant).

There are also some similarities:

- Both abstract classes and interfaces can be used as data types to *declare* a variable of a complex type (e.g. `Animal indiana = new Dog()`).
- Neither can be used to *instantiate* an object using `new` (because neither contains a full "blueprint" of what the new object would look like, e.g. `new Animal()` is not valid).

3.3 Bottom-up inheritance

Sometimes, inheritance is a consequence of the design of your program: you have an interface (e.g. `Animal`) that is implemented by several classes (e.g. `Horse`, `Dog`) and then decide to add a little code into it and turn it into an abstract class.

Sometimes it happens the other way around: you have two or more classes that are apparently unrelated, and then realise that they have the same code. At this point you want to move the code "up" to an abstract class. For example, imagine that we had a class `ManorHouse` and another class `Castle`, initially unrelated in the design of our program:

```
public class ManorHouse {
   private Gate gate;
   // ...
   public void closeForTheNight() {
       gate.moveInwards(90);
       gate.getLock().setLocked(true);
   }
   // ...
}
(...)
public class Castle {
   private Gate gate;
   // ...
   public void closeForTheNight() {
       gate.moveInwards(90);
       gate.getLock().setLocked(true);
   }
   // ...
}
```

Code repetition is bad. As soon as we realise that both classes have the same method, such method should be put in a parent class:

```
public abstract class GateGuardedBuilding {
   private Gate gate;
   // ...
   public void closeForTheNight() {
       gate.moveInwards(90);
       gate.getLock().setLocked(true);
   }
}
(...)
public class ManorHouse extends GateGuardedBuilding {
   // ...
}
(...)
public class Castle extends GateGuardedBuilding {
   // ...
}
```

Now the method `closeForTheNight()` is available to both `ManorHouse` and `Castle` because they are extending the parent (abstract) class `GateGuardedBilding`.

Clear? Not so easy... There are situations in which this "repetition of code" is not straighforward. Look at the following simplified example:

```
public class DrinkRefrigerator {
    // ...
    public int buyCan(int money) {
       if (money < price) {
           return money;
       }
       int change = money - price;
       releaseCan();
       return change;
    }
    // ...
}
(...)
public class ChocBarVendingMachine {
    // ...
    public int getBar(int moneyGiven) {
       if (moneyGiven < price) {
           return moneyGiven;
       } else {
           giveChocolateBar();
           int change = moneyGiven - price;
           return change;
```

```
        }
    }
    // ...
}
```

Although it may not seem at first that a `DrinkRefrigerator` and a `ChocBarVendingMachine` have much in common, an analysis of the code of methods `buyCan()` and `getBar()` shows a strong similarity, suggesting that the code should be abstracted away to a parent class. How would it look like? Maybe something similar to this:

```
public class Sale {
    public boolean sold; // public fields because it does not
    public int change;   // have methods
}
(...)
public abstract class VendingMachine {
    // ...
    public Sale buy(int money, int price) {
       Sale result = new Sale();
       if (money < price) {
           result.sold = false;
           result.change = money;
       } else {
           result.sold = true;
           result.change = price - money;
       }
       return result;
    }
    // ...
}
(...)
public class DrinkRefrigerator extends VendingMachine {
    // ...
    public int buyCan(int money) {
       Sale sale = buy(money, price);
       if (sale.sold) {
           releaseCan();
       }
       return sale.change;
    }
    // ...
}
(...)
public class ChocBarVendingMachine extends VendingMachine {
    // ...
    public int getBar(int money) {
       Sale sale = buy(money, price);
```

```
            if (sale.sold) {
                giveChocolateBar();
            }
            return sale.change;
        }
        // ...
    }
```

The code in the subclasses is shorter and clearer because all the hard work is done in the code that has been abstracted and put in the parent class. If this code was more complicated and involved several comparisons or long intrincated calculations, the benefits would be even greater.

In this second example, it is important to note that the "repeated" methods had two effects: a return value (to give change) and a side effect (release can or chocolate bar). Because of this, we needed to create a new intermediate class `Sale` —with no methods— in order to be able to return two values from the new method `buy()`. Remember that methods can only return one value; the only way they can return more is by using a complex type to encapsulate them all.

3.4 Visibility revisited

As it evolves and grows, a program can become really complex, usually beyond of what its programmer(s) can cope with. We know that one of the basic strategies to keep complexity under control is by hiding information.

Information is hidden by making it not visible: visible fields can be read and written, but non-visible fields cannot; visible methods can be called, but non-visible fields cannot. We have used two keywords to control the visibility of our classes' fields and methods: `public` and `private`. Although these two keywords will cover most of your needs as a programmer, it is good to know that there are four levels of visibility in Java: public, protected, default (package), and private.

3.4.1 Public

Public fields are visible for any other class, and can be read and written accordingly. Letting other classes (even classes that you have written yourself) to modify the fields of your classes can have unpredictable effects in a normal-size program, and that is why you should make all your fields private as a rule of thumb. The only exception are the fields of method-less just-used-as-intermediate-data classes (like `Sale` in the former section).

Public methods are visible to other classes and define the public interface of your class. This is usually made explicit by implementing one or more Java interfaces. Note that there is an ambiguity in natural language here: the public methods of a class are its "interface" in the sense that they define how other classes can interact with it, but that does not necessarily mean that the class `implements` (in the Java sense) a explicit Java `interface` defined in a `.java` file. You should keep this distinction clear when you read the documentation of software projects as both meanings of the words are sometimes used without much care.

3.4.2 Protected

Protected fields and methods are visible for classes in your package (i.e. the same folder/directory as the class) and for descendant classes. There are a few cases in which the distinction `protected`

vs. `public` makes sense (i.e. in event-oriented architectures) but in most cases this nuance is not worth the additional complexity, and that is why many of the most recent programming languages do not include a `protected` visibility, only `public`.

As a rule of thumb, use always public if you want other classes to call the methods in your class. On the other hand, it is important to understand what `protected` means because you will find it often when you read Java code, especially if it is old legacy code.

3.4.3 Default (i.e. package)

When the visibility of a field or method is not specified, its default visibility is "package-wide", i.e. it is visible to other classes in the same package. There is not a keyword for declaring "package" visibility: fields and methods of unspecified visibility become visible for other classes in the same package.

Up to now, you have not noticed any difference between `public` visibility and default visibility because all your interacting classes have been part of the same package or were part of the standard Java library (e.g. String) and you used only their public methods.

When Java was originally created, making things visible to the same package seemed like a good idea for several reasons. However, things have changed over the years and it is very rare today to find a situation where "package" visibility makes sense but `public` visibility does not, with the additional complexity of keeping track and what is visible at which level. Therefore, you will very rarely find default visibility in Java code for a good reason; on the contrary, it is almost always a symptom of sloppy programming, a sign of a programmers not thinking properly about the design of their program or –even worse— not understanding the difference between `private` and default visibility.

3.4.4 Private

Private fields and methods are only visible inside their own class. This is the most complete form of "hiding", and should be the default option for all the fields and most of the methods (unless they are defined in an interface, i.e. they are useful for other classes).

Note that `private` only determines visibility between *classes*, not between *objects*. Objects of the same class are able to see each other fields and methods. This is very evident in most implementations of the `.equals()` method.

```
public class Integer {
    private int value;
    // ...
    public boolean equals(Integer other) {
        if (this.value == other.value) {
            return true;
        } else {
            return false;
        }
    }
}
```

Although the field `value` is private and cannot be seen from any other class, any object of class `Integer` can see the `value` method of other objects of the same class.

3.5 Changing behaviour, overriding methods

Extending a class is a way of reusing part of its code in the form of its public methods. However, there are cases in which by extending a class you are inheriting a method you do not want. For example, your class `Employee` may inherit from class `Person` a method `equals()` that says that two objects are the same person if they have the same passport number; however, you want your employee-objects to represent the same employee if they have the same national insurance number. When this happens, you have to *override* the method.

Overriding a method is trivially easy: just write the method again. Java will automatically use the most specific version of the method when the method is called. Methods `equals()` and `hashcode()`, from class `Object`, are typical examples of methods that are overriden by many classes.

```
@Override
public boolean equals(Object other) {
     // ... different code here to establish equality
}
```

The `@Override` annotation[3] is optional, but it is a good idea to always use it. This will make it evident to other programmers that the method is overriding a method from an ancestor class.

Final methods In the same way that we can declare a class to be `final` so that it cannot be extended, we can declare a method as `final`; this will prevent the method from being overriden.

If a class is final, then all its methods are final too. The opposite is not true: a class with all its method marked as final can still be extended (even if the existing ones cannot be overriden, it is always possible to add new methods as long as the class is non-final).

3.6 The keyword super

There are few keywords in Java that we have not seen yet, but one that is extremely relevant when we talk about inheritance is `super`. This keyword designates the *superclass* or *parent class*, and can be used to use methods defined in it, as in this trivial example:

```
/**
 * This method does exactly the same as the method of the same
 * name of the parent class, plus printing a message on screen.
 * /
@Override
public void someMethod() {
    super.someMethod();
    System.out.println("Calling someMethod() at subclass.");
}
```

In practice, calling methods from the superclass is not something that is used very often (there are some exceptions, but few and far between). The only situation in which it is common to use `super` to call methods from the parent class —and it is important to understand it— is when the

[3] We will see more about annotations in a week, when we start doing automated testing. For now, you just need to know that they are added just before the method declaration and start with a @ sign.

keyword is used to call the constructor of the parent class inside the constructor of the subclass. This is so important that it must be done at the first line of the constructor:

```
public MyClass() {
    super();
    // rest of method here...
}
```

```
public MyClass(arguments) {
    super(arguments);
    // rest of method here...
}
```

If you try to do anything —like initialising your fields— before the `super(...)` constructor is called, you will get an error from the Java compiler.

Java magic constructors and the hidden super()

As this point, you may be wondering how was it possible that you have created all your classes until now without calling `super()`. The answer is that Java makes a little bit of magic behind the scenes: Java can add a default constructor to your classes and/or it can call the `super()` constructor even if you forget to use it.

Default constructor Every Java class must have *at least one* constructor method. If your class does not have an explicit constructor, Java will add one like this:

```
public NameOfMyClassHere() {
    super();
}
```

This constructor does not do much (just calls the superclass' constructor) and has no parameters. If you do not add your own constructor to your classes, they will at least have this one. On the other hand, as soon as you add a constructor to a class Java will see it, and will not add the default constructor in that case.

Default call to the parent class' constructor The first thing that every constructor must do is calling the parent class' constructor. The parent class constructor must always be executed before the current class constructor starts to do anything. In other words, a `Dog` is an `Animal`, but it becomes an `Animal` first (e.g. initialises legs and eyes) and a `Dog` later (e.g. sets sound as barking and smell sensititivy to a high value).

If the first line in one of your constructor methods is not a call to one of the superclass' constructors, Java will automatically add a call to the default parameter-less constructor: `super()`. If your parent class does not have a parameter-less constructor (either explicit of magicaly added), the Java compiler will complain.

3.7 Only one mother(-class)

A class can only extend another one class. In other words, every Java class has one and only one parent class (usually `Object`).

If a class could extend two different classes, it may potentially inherit two methods with the same signature. For example, both parents may implement a `getName()` or `equals()` method in different ways. Which implementation should be inherited by the subclass[4]?

[4]This is known in computing as the *diamond problem*.

Some programming languages allow *multiple inheritance*, (extending several classes), and have rules that determine what to do in the case of method clashes. These rules are not always clear or intuitive, and can be source of confusion and bugs. The Java creators considered (probably correctly) that the benefits of multiple inheritance did not overcome its shortcomings, and Java is thus a *single inheritance* language, where every class extends one and only one class.

On the other hand, a class can implement an arbitrary number of interfaces. As interfaces do not contain any implementation, it is not a problem to implement two interfaces that declare the same method: any implementation of that method will satisfy both interfaces[5].

4 Final note: composition vs. inheritance

Inheritance is a form of code reuse, but it is not the only one, nor is it the most adequate in any situation; there are many more. In Java, the most common forms of code reuse are *inheritance* and *composition*. Although we will learn more about these topics in the next module (Object-Oriented Design and Programming), it is good to have a fair understanding of the basics.

Inheritance establishes an "is-a" relationship (e.g. a dog is-an animal), and is implemented by using the keyword `extends`. When a class extends another class, it gets access to all its public and protected fields and methods.

Composition establishes a "has-a" relationship (e.g. a dog has a leg and has cells), and is implemented by including the "contained" classes as types of fields of the "container" class. When a class contains some other class/es it gets access to all its/their public methods and fields.

Both compositition and inheritance give access to the functionality of another class. Which is the right approach in each situation?

A first question that a programmer should ask is whether there is an "is-a" or a "has-a" relationship between the classes. For example, both `Animal` and `Cell` potentially have a method `breath()` that a class `Dog` may reuse, but it is clear that a dog is an animal but it is not a cell. However, the answer to this question is not always clear-cut. If you have a class `Pound`, a class `Currency`, and a class `Coin`, which one extends which one? Is there any `extends` relationship at all? The final answer depends on many factors that depend on the specifics of the source code involved, but it is important to understand that inheritance and composition are very different approaches.

Only one extension The first difference is very obvious. In Java you can only extend one class, but can include as many classes as you want as fields, even having many fields of the same class.

Flexibility Another difference that is not obvious at first sight is that inheritance relationships are difficult to change, which can become problematic when the requirements of the project change in the future[6]; on the other hand, composition is easier to modify. Type hierarchies defined by `extends` become very rigid over time: changing one of the classes in the hierarchy tree usually involves changes in many other classes in different parts of the source code. In other words, type hierarchies provide structure to the code, but increse its *coupling* (dependance between classes); a

[5] Some people use the argument of implementing several interfaces in Java to say that Java allows for a "special" or "limited" form of multiple inheritance. I think it is clearer to use the term inheritance exclusively for those cases in which actual code is actually *inherited* from another class.

[6] And you can bet the farm that they will. I still have to find a project in which the requirements were not only clear at the beginning (and there are *very few* of those) but that they remained constant until the end.

high coupling is bad because its makes the code more difficult to change and modify[7]. Composition, on the other hand, provides more flexibility. Changes to the classes inside a class are typically invisible to all other classes (they are `private` fields) so the disruption of the code is contained.

Additional baggage An additional difference is that inheritance brings more baggage than composition. Take the example of a class `DancePairMaker`, that takes names of people in pairs and then prints them. Pairs of names is something similar to a `Map`, and adding pairs sounds like `Map.put(String,String)`...so we would like to reuse that code in `Map` without the need to write it all again. If `DancePairMaker` *includes* a `Map`, it will use the method `put()` and nothing else (there is no need to remove dance pairs: just adding and printing). However, if `DancePairMaker` extends a `Map`, other classes may be able to `remove()` pairs because this method is inherited from the `Map`. Maybe this is not what the programmer wanted at the time of extending `Map`, but that is a consequence of extension.

For these two reasons, and some other minor ones, the rule of thumb is "composition works better than inheritance". If you find yourself thinking about extending a class, think twice in case it is better to include the class instead. (Of course, you should still abstract repeated code to parent classes, as explained in Section 3.3).

[7]Some people refer to this as a code of *high viscosity*.