



Ca' Foscari
University
of Venice

Object-oriented programming

Pietro Ferrara

Università Ca' Foscari di Venezia, Italy

pietro.ferrara@unive.it

<https://www.dais.unive.it/~ferrara/>

Premise

This book contains the notes of the course “Object-oriented programming 1” (“Programmazione ad oggetti Modulo 1”)¹ at Ca' Foscari University of Venice taught by prof. Pietro Ferrara². These notes have been written during the academic year 2021/22, and they have not been revised, proofread, or anything else. Therefore, use them at your own risk. Please apologize in advance for any typo, misunderstanding, or embarrassing mistake, and remember that reports to pietro.ferrara@unive.it are highly welcomed.

The main goal of this book is to introduce students of the 2nd year of a bachelor course in Computer Science to object-oriented programming, and to the Java programming language. Those students already attended a 6 ECTS course about C (“Introduction to programming”) and a 12 ECTS course about C++ (“Programming and laboratory”). Therefore, we assume the reader has some basic background about imperative programming, such as variables, basic control flow structures (if-then-else, loops, etc.), memory allocation, etc..

The book is structured into three main parts:

- 1) Encapsulation covers how we can structure our code into many small capsules (aka, classes) containing some data (aka, fields) and offering some functionalities (aka, methods), and how we can group them into larger capsules (aka, packages). We also cover how we can hide some parts of our implementation, and we deeply discuss what is exposed to the external world (where documentation and contracts are a part of what is delivered).
- 2) Polymorphism covers how we can obtain that a same symbol has different behaviors based on the context of execution. We discuss two main forms of polymorphism: by inheritance and subtyping (and this covers most of this part of the book), and by type generics.
- 3) Java in action presents the main components of the Java standard library that are essential for any Java program we might want to develop nowadays. This includes `java.lang.Object`, wrapper classes, exceptions, annotations, and reflections. We also briefly introduce Spring in order to provide some insights about how nowadays (aka, at the beginning of the 2020s) Java is mostly used.

¹ <https://www.unive.it/data/insegnamento/339895>

² <https://www.dais.unive.it/~ferrara/>

The most part of the book refers to what was already available in Java 8.

Unfortunately, this book does not include any exercise, as it is mostly the scripting of the aforementioned course. The plan is to add exercises to this book as well as a couple of new chapters (one about modules, and one about the Java Virtual Machine and bytecode), but this won't happen before fall 2022.

Any feedback, suggestion, request, etc.. is highly appreciated and it should be addressed to pietro.ferrara@unive.it. There exist also slides covering the content of this book, but they are not publicly available, and they should be requested to the aforementioned e-mail address. Feel free to use this material for your courses giving appropriate credit.

Chapter 1: Why yet another programming paradigm?

Nowadays, nobody asks this question when talking about object-oriented programming languages, since the answer looks obvious: because it is object-oriented, everybody uses this, there are tons of job offers for people that develop software using these programming languages. This is an acceptable answer for somebody that is looking for a job, since he/she should target the technologies that are quite popular in order to raise the chance to get hired. Unfortunately, this is not an acceptable answer for somebody that is starting to study object-oriented programming. Before entering into the details of this (beautiful) world, we need to understand why this world even appeared. So let's go back by some decades and (try to) understand what needs gave rise to object-oriented programming languages.

1.1 Engineering

But first of all, we need to agree on a very basic concept: software engineering has nothing different from any other form of engineering. By engineering, we simply mean the process that applies our knowledge and brain in order to devise some solutions. These solutions might be roads, airplanes, or... software! However, engineering has millennia of history, while software appeared less than a century ago. So there is much we can learn from other forms of engineering in order to understand how software engineering is evolving and why.

We might take millions of examples of engineering, ranging from the pyramids in ancient Egypt to modern airplanes and rockets. Picking a random one, I will discuss the engineering that is behind the Eiffel tower in Paris. Why this one? Because I think it contains all the elements that could inspire our road to give a sense to OO programming language, and it is simple enough to be understood even by people with limited knowledge about how a tower can be designed and built.

Let's go straight to the point: when we say Eiffel tower we always think about a unique steel block with the form of this well-known tower. What we (luckily) miss to see are all the pieces it is made of, how the tower was assembled, and all the thousands of small

details that are essential for keeping this tower up. Let's give some numbers just to give a rough intuition of its grandeur: the tower is made of 18,038 distinct metallic parts, 2,500,000 rivets, 7,300 tonnes of iron, and 60 tonnes of paint. The most part of the tower was created somewhere else (e.g., in the Levallois-Perret factory with 150 workers) and only assembled on the construction site (where there were between 150 and 300 workers). But there is much more to say: 5,300 workshop drawings were needed in order to draw all the components of the tower, and 50 engineers and designers were involved in such an effort. Indeed, the original project is attributed to two engineers, and it seems that Gustave Eiffel was not so enthusiastic about the project, even if he approved to proceed with the project. More than 2 years were needed in order to construct the tower.

Well, I hope that by now I have convinced you that the construction of the Eiffel required a huge effort by people with different expertises and skills. This means that the different pieces used to build up the tower had some standard and easy-to-understand structure. In particular, rivets were the key component chosen to assemble the various pieces. Each rivet was inserted by four people that had different tasks to perform, and they didn't have any knowledge about how the different pieces and rivets were built. All together, the construction of the Eiffel tower required to (i) build different pieces in different sites (involving different workers with different expertises) with some workers, and (ii) assemble these pieces with different workers. The overall dimensions of the tower required such an approach, since a single worker (or even a small team) cannot be successful (in a reasonable time) in the construction of the tower.

1.2 Back in (relatively old) time

Now let's go back to software and analyze when OO programming languages were invented, then developed, and finally became popular. These three moments are clearly distinct, and several years passed from one step to another.

In the field of software development, the term object appeared between the end of the 50s and the beginning of the 60s, that is, more than half a century ago. Sketchpad that officially appeared in 1963 in Ivan Sutherland's PhD thesis, is probably the first example of an object-oriented program. While at that time the notion of object was still under construction, the intuition was definitely already there. In the mid-60s, the programming language Simula started to support some features of OO programming. Finally, SmallTalk, probably the first programming language supporting OO programming by its design, was released in 1972.

However, the popularity of this programming pattern remained very low for several decades. The (probably) most popular OO programming language in history is Java, released for the first time in 1995. Since the beginning of 2000s Java has always been one of the most popular programming languages in all the rankings of this area. C++ is another popular OO programming language that since the mid-80s has been steadily one of the most popular programming languages. But why did it take more than 2 decades to achieve such popularity?

To answer this question, we need to consider what types of software were developed over the years. In the 60s, 70s, and part of 80s the most part was focused on safety critical embedded systems, such as flight controls, automotive systems, and medical devices. These systems had minimal computational capabilities, and strict realtime requirements (that is, they needed to provide some strict time guarantees). In addition, they are often closed systems, where by close we mean that every piece of software (and usually of firmware and middleware) was under the control of the manufacturer. Briefly, this is not the Eiffel tower scenario, that indeed is the target of OO programming languages.

What do I mean by this? How can we relate the construction of a concrete solid tower with the implementation of a digital untouchable information system? Let's go ahead with the analogy: if we want to build up a digital Eiffel tower we need to find a way to use the pieces (aka, software) created by others through some standard interfaces. Even more, we need to structure our digital tower into many small pieces that are understandable for people that did not have the knowledge to construct them. In the digital world, we can push the metaphor even to one more level: we would love to take pieces constructed by others for other reasons (aka, not for building up a digital tower) and use them inside our project. While this is out of the scope of an engineering project dealing with concrete constructions (since the construction of each piece requires to employ material and has a cost), with digital software one might have available some software that could be useful for his/her final goal.

The type of software that was developed until several decades ago did not fall inside this scenario: the dimensions of the project were limited, the software was completely developed in house (that is, from a well-defined and closed team of developers), and there were strict requirements in terms of safety and security of the software.

Such a scenario changed completely with the advent of personal computers and the Internet. In particular, application software (such as word processors and browsers) and Web applications often implement very complex logics (that is, they have relevant

dimensions) and perform non-safety critical tasks (that is, even if the software breaks or it can be attacked the damages are negligible). Nowadays, it is pretty common to build up software composed of millions of code lines, where the most part of this code is developed by external people. This scenario is exactly the one targeted by OO programming languages and that gave them all the popularity they have nowadays.

1.3 The essence of object-oriented programming language

Finally, we can approach the big question: what is the essence of OO programming languages? All the essence stays in one single world: object. What is an object? An object can be... anything! So what is the essence of OO programming language? To provide a means to model... anything! If we would have such capabilities, we might structure our code into many coherent capsules representing something, and these capsules might be reused in many different contexts without the need of understanding the inner details. I know that all this sounds (and indeed is) pretty vague, but starting from now things will become more and more concrete.

Let's start to think about how we might want to represent a generic object. Briefly, we can think about two main components: the state of an object, and its functionalities. For instance, if we think about a car, we might represent its weight, speed, and amount of fuel in the tank (that is, a state), and we would like to accelerate consuming some fuel, break, and fill the tank (that is, some functionalities). OO programming languages allow us to define fields (aka, a state) and methods (aka, some functionalities) on our objects.

We are finally ready to enter the world of object oriented programming languages!

Part 1: Encapsulation

Chapter 2: Classes, fields and methods

Let's start by asking ourselves what components we need to represent a generic object? In particular, starting from some concrete object, we want to *abstract* only some parts, and then represent them in our program. Abstraction is an extremely common thinking process: our mind cannot grasp all the details from the real objects, and it proceeds by focusing only on some parts of that, recognizing and talking about objects referring only to some of their features. For instance, what would you answer if I asked you “what car do you have?”? Probably you would list the manufacturer of the car, the specific model, and its color. For sure you won't describe any detail of any single component of the car, and your description could match many different concrete cars. Indeed, to simplify we added various ways to uniquely identify cars, such as plate numbers and vehicle identification numbers.

Therefore, the first question we should ask ourselves is: what do we need to abstract an object?

2.1 Data and functionalities

Generally speaking, our way of thinking abstracts object with some data (e.g., the manufacturer, model and color of a car) and some functionalities (e.g., accelerate and break). Let us introduce a concrete example: we want to model cars. Rather than their identification, we are more interested in representing their motion. In particular, we focus on the acceleration and breaking of the car, as well as its fuel consumption.

Therefore, about the data of interest we are going to focus on the modelling of the speed and the amount of fuel stored in the tank of the car. Instead, as functionalities we plan to support the acceleration and breaking of some amount of speed, and refueling of some amount of fuel. From this very brief and informal example, it should be already evident that in an OO program the data should be tightly coupled with the functionalities that need to access or modify the data.

2.2 Fields and methods

OO programming languages represent data through fields, and functionalities through methods. A field consists in the definition of a name, and its static type. For instance, we can define the speed of a car and the quantity of fuel of a car through two fields of type double:

```
double speed;  
double fuel;
```

Note that the choice of double values to represent this information is arbitrary, and by simply looking at the fields we cannot figure out exactly what they represent: are we referring to litres or gallons of fuel? Petrol or diesel? And about the speed, km or miles per hour? The implementation of the fields specify only the type and name of the field, but their concrete meaning needs to be specified in other ways (that is, through some specific comments we will discuss later).

Moving towards the functionalities, our OO program supports them by some code (aka, implementation) inside a method

```
void refuel(double amount) {  
    fuel = fuel + amount;  
}  
void brake(double amount) {  
    speed = speed - amount;  
}  
void accelerate(double amount) {  
    double fuelConsumed = amount*litersPerKmH;  
    speed = speed + amount;  
    fuel = fuel - fuelConsumed;  
}
```

The implementation of refuel simply adds the given amount of fuel to the car's tank, while similarly brake reduces the speed of the car by the given amount. Instead, accelerating exposes some more complexity. In fact, when accelerating we need not only to increase the speed, but also to consume some fuel. However, in our model we specify the increase of speed, but we need to know something more in order to calculate how much fuel is consumed. For this reason, we need to add another field litersPerKmH that contains how many liters of fuel are needed to increase the speed by one km per hour.

2.3 Classes and objects

We are finally ready to create our first “meaningful” capsule of code! In particular, we want to assemble together the fields and methods in a unique piece of code. This is supported by OO programming languages through the definition of classes.

```
class Car {  
    double speed = 0;  
    double fuel = 0;  
    double litersPerKmH = 0.01;  
  
    void refuel(double amount) {  
        fuel = fuel + amount;  
    }  
  
    void brake(double amount) {  
        speed = speed - amount;  
    }  
  
    void accelerate(double amount) {  
        double fuelConsumed = amount*litersPerKmH;  
        speed = speed + amount;  
        fuel = fuel - fuelConsumed;  
    }  
}
```

But what is exactly a class? A class is nothing else than a *template* from which objects can be instantiated. Classes define the structure of the data stored in the objects through a set of fields, and their functionalities through a set of methods. Obviously, each method and field needs to have a unique signature (that is, field name, and method name and number and type of parameters), otherwise we would not know what field or method we would access when accessing them from outside.

The implementation of class Car couples together the data (fields) and functionalities (methods) of the abstraction of cars we had in mind. Note that the three methods all read and write the state of the object. But now how can we create cars starting from class Car? Java provides us the keyword `new`, that given a class name, instantiates it (that is, allocates the memory for the fields and initializes their values) and returns a reference to this “fresh” object. We can then store this reference in a local variable, and we can access and modify fields as well as invoke methods through this variable. Consider the following example.

```
Car myCar = new Car();  
myCar.refuel(2);  
myCar.accelerate(100);  
myCar.brake(30);  
myCar.accelerate(50);  
myCar.brake(200);
```

This piece of code (i) creates a new car, (ii) adds to liters of fuel to the car's tank, (iii) accelerate by 100 km/h, (iv) brakes by 30 km/h, (v) accelerates by 50 km/h, and (vi) brakes by 200 km/h. If we execute this piece of code, we will end up with a car with 0.5 liters of fuel, and with a speed of... - 80km/h! Such a result exposes some inconsistencies in our implementation: if the idea was to brake, we should never end up with a negative speed. The same would happen if we do not refuel before accelerating: the car will start to move, and it will get a negative amount of fuel. Therefore, we need to improve our implementation in order to take into account all these cases.

```
class Car {  
    double speed = 0;  
    double fuel = 0;  
    double litersPerKmH = 0.01;  
  
    void refuel(double amount) {  
        fuel = fuel + amount;  
    }  
    void brake(double amount) {  
        if(amount > speed)  
            speed = 0;  
        else speed = speed - amount;  
    }  
    void accelerate(double amount) {  
        double fuelConsumed = amount*litresPerKmH;  
        if(fuelConsumed < fuel) {  
            speed = speed + amount;  
            fuel = fuel - fuelConsumed;  
        }  
        else {  
            double increaseSpeed = fuel / litresPerKmH;  
            speed = speed + increaseSpeed;  
            fuel = 0;  
        }  
    }  
}
```

When we design and implement an OO program, we need to take into consideration all the possible cases that might happen, in order to provide a coherent implementation of the abstraction of the concrete world we have in mind. We need always to remember

that the main goal of OO programming is to provide a piece of software that can be reused in many different contexts without entering into the implementation details. Therefore, the implementation must be always consistent with the real objects it wants to abstract. We will discuss these aspects more in detail later.

2.4 Running a program

Last but not least, we need to understand how an OO program is executed. Since the program is structured into many capsules (classes), there is no unique entry point for the application. Each class can potentially define an entry point through a method `main`, and when we execute the program we need to specify the class from where we want to start the execution. For instance, we can put the code instantiating the `Car` class above into the `main` method of class `Car` as follows.

```
public class Car {  
    .....  
    public static void main(String[] args) {  
        Car myCar = new Car();  
        myCar.refuel(2);  
        myCar.accelerate(100);  
        myCar.brake(30);  
        myCar.accelerate(50);  
        myCar.brake(200);  
    }  
}
```

Then when we start the execution we need to instruct the Java Virtual machine to execute this class. For instance, from the command line we can launch the execution with the command “`java Car`”.

2.5 Interactions between objects

So far we used only primitive types (e.g., `double` and `int`) for our fields and method parameters. However, each class defines a type (like a `struct` in C), and therefore it can be used as (static) type of fields and parameters. In this way, an object can store (a pointer to) another object inside its fields, or it can receive (a pointer to) another object as a parameter of a method.

Let's structure our example a bit more. In particular, so far we talked generically about “fuel”. However, we all know that there are different types of fuel (e.g., `diesel` and `petrol`), each car relies on one specific type, and we should not refuel the car with a

different fuel type. Conceptually, the amount of litres needed to accelerate highly depends on the fuel type, since some fuel might be more performant than another one. In addition, there are other properties that are specific for a fuel type and we might be interested in, such as the price of a liter of the fuel. All this leads to the same conclusion: we need to represent this real world concept with an ad hoc class that abstracts the reality carrying on only the information we are interested in. We then define the class `FuelType` as follows.

```
class FuelType {  
    String type;  
    double costPerLiter;  
    double litresPerKmH;  
}
```

`FuelType` can be then used as a field of class `Car` in order to store the fuel type of a car. Inside the code we can then access the fields of the fuel type in order to compute how much fuel is consumed.

```
class Car {  
    double speed = 0;  
    double fuel = 0;  
    FuelType fuelType;  
    void accelerate(double amount) {  
        double fuelConsumed = amount*fuelType.litresPerKmH;  
        ...  
    }  
}
```

Note that when we have a field, parameter or local variable that points to an object we can (i) access its fields (to read or write it), and (ii) invoke its methods. In both cases, we can dereference the pointer by following the name of the field or local variable with a dot (e.g., `fuelType.litresPerKmh` and `myCar.refuel(2)`). When we do not assign an explicit initial value to a field, this is automatically initialized to its default value (that is, zero or null).

Let's move one step forward: when we refuel a car we need not only the amount of fuel, but also the type. If the type is different, we should not refuel the car. One solution might be to add a `FuelType` parameter to method `refuel`. However, with such a solution we would use two parameters to represent a unique concept, that is, a tank of fuel. Therefore, we decided to add a class `FuelTank` that stores a fuel type and an amount of fuel, and we pass this to method `refuel` of class `Car`.

```
class FuelTank {  
    FuelType type;  
    double amount;  
}
```

```
class Car {  
    ...  
    void refuel(FuelTank tank) {  
        if(tank.type.type.equals(this.fuelType.type))  
            fuel = fuel + tank.amount;  
    }  
}
```

2.6 Static fields and methods

So far, the declarations of fields and methods resembled the declaration of standard variables and functions of other imperative and not OO programming languages. Instead, field and method declarations can be preceded by various types of modifiers. We will study a few of them in the next chapters, since these allow us to define different types of behaviors w.r.t. our OO model. Now we introduce static fields and methods.

Up to now, we have seen how we can define fields to store information about instances of the class. Each time a class is instantiated into an object, some memory to store the values of these fields is allocated, and each object has its own state for the field. However, in some cases we would like to have fields whose state is shared among all the instances of the class. For instance, we want to count how many FuelTank have been created so far. To do so, we need to introduce a field that is (i) shared among all the instances of class FuelTank, and (ii) incremented each time an instance of FuelTank is created. This is obtained by declaring a static field numberOfTanks. We add also a field id that stores the identifier of the current tank by assigning the current value of numberOfTanks, and incrementing it by one.

```
class FuelTank {  
    static int numberOfTanks = 0;  
    FuelType type;  
    double amount;  
    int id = numberOfTanks++;  
}
```

Similarly we can define static methods. These methods can access only static fields, invoke only static methods, and the keyword this does not exist since the method does not refer to a specific instance of the class. So let's implement a static method that reset

the number of tanks created so far.

```
class FuelTank {  
    static int numberOfTanks = 0;  
    static void resetCounter() {  
        numberOfTanks = 0;  
    }  
}
```

Finally, we can also define a static constructor. Such constructor is invoked only once at the beginning of the execution of the program³, and it can invoke only static fields and initialize static fields. The static constructor cannot receive parameters (since it is called by the runtime environment), and it is defined through the static keyword followed by a block containing the code to be executed. For instance, we might define in our class FuelTank a static constructor that reset the counter at the beginning of the execution.

```
class FuelTank {  
    static int numberOfTanks;  
    static {  
        resetCounter();  
    }  
    static void resetCounter() {...}  
}
```

Last but not least, static fields and static methods of a class can be accessed in three distinct ways: (i) using a variable pointing to an object instance of the given class (e.g. tank.resetCounter()), (ii) inside the class by simply using the field or method name (e.g., resetCounter() as in the static constructor above), and (iii) using the name of the class followed by a dot and then the name of the component (e.g., FuelTank.resetCounter()). Even if allowed, the first way is highly discouraged, since it gives the impression we are accessing a component of an instance of a class instead of a static component. The second way has the same drawbacks we already discussed about instance fields and methods. Therefore, using the class name is considered the preferred method to access static methods and fields. Class FuelTank can be then implemented as follows.

³ Indeed, this might be invoked later, but for the sake of simplicity we restrict the scenario.


```
class FuelTank {  
    static int numberOfTanks;  
    static {  
        FuelTank.resetCounter();  
    }  
    static void resetCounter() {  
        FuelTank.numberOfTanks = 0;  
    }  
}
```

2.7 Structure of a class

In this chapter, we presented all the main components of a class in an OO programming language in general, and Java in particular. While we are usually free to define fields and methods in any arbitrary order inside a class, by convention we should first define the fields, and then the methods. In this way, we clearly distinguish between the data stored in the objects, and the functionalities they provide. Below you can find a sort of template that should be followed when implementing a class.

```
<class_modifier>* class <class_name> {  
    <field1_modifier>* <field1_type> field1_name;  
    <field2_modifier>* <field2_type> field2_name;  
    ...  
    <method1_modifier>* <method1_returntype> <method1_name>(<method1_pars>) {  
        <method1_body>  
    }  
    <method2_modifier>* <method2_returntype> <method2_name>(<method2_pars>) {  
        <method2_body>  
    }  
    ...  
}
```

2.8 Exercise

Using the concepts of OO programming (classes, methods, fields), how would you represent the identity of a car?

Chapter 3: From classes to objects

In the previous chapter we defined what we mean by class and what components are part of a class (fields and methods). A class can be seen as the definition of the structure of the data we want to play with. Compared to databases, a class represents the names and types of the various columns (as well the implementation of some functionalities). We now see how we can instantiate a class in order to obtain objects with their own state like rows in a table or database. Generally speaking, a class can be instantiated many times, and each object is an instance of a specific class.

3.1 Constructors and new

In the previous chapter, some snippets of code contained the instruction `new`. Such instruction is used to instance a class, and it returns a pointer to a fresh object. When we instantiate a class, we usually need to initialize its state to some coherent values. For instance, we introduced class `FuelType`, and now once we instantiate it we need to set fields `type`, `costPerLiter` and `LitresPerKmH` to some values. For instance, in the main method of Section 2.4. we might initialize an instance of class `FuelType` as follows:

```
public class Car {
    .....new CarCar
    public static void main(String[] args) {
        Car myCar = new Car();
        myCar.fuelType = new FuelType();
        myCar.fuelType.type = "diesel";
        myCar.fuelType.costPerLiter = 1.4;
        myCar.fuelType.litresPerKmH = 0.01;
        ...
    }
}
```

Such a solution is far from being ideal: our main method needs to manually assign each field, and we do not have a way to initialize all fields together.

This is why OO programming languages provide us the possibility to implement special methods called *constructors*. As the name suggests, a constructor is a method that is

executed only once on each object when it is created. In Java, a constructor is defined as a method without a return type, and whose name is the same as its class. For instance, the following piece of code defines a constructor of `FuelType` that receives as parameters and initializes all the fields.

```
class FuelType {  
    String type;  
    double costPerLiter;  
    double litresPerKmH;  
    FuelType(String t, double c, double l) {  
        type = t;  
        costPerLiter = c;  
        litresPerKmH = l;  
    }  
}
```

When we introduced the main method in Section 2.4, we used the keyword `new` followed by the name of the class we instantiated (`Car`) and an open and close parenthesis. Such a keyword executes four distinct steps: (i) allocates the memory required by the object (that is, space to store all the values of the class fields), (ii) initializes all the fields to default values (zeros or null), (iii) executes a constructor, and (iv) returns a pointer to the freshly allocated object. Note that in Java (and in the most part of existing OO programming languages), objects are allocated in the heap, thus they persist in the memory after the execution of a method. By default, if a class does not define a constructor, Java adds a constructor without parameters that (intuitively) initializes the fields to their default value (or the value explicitly defined in the field declaration). Therefore, in our previous example class `Car` has such an implicit constructor, and when we ran in the main method “`new Car()`” we executed that constructor. Once we define a constructor in a class like `FuelType`, then the constructor with empty parameters is not implicitly added anymore, and we need to invoke the implemented one. For instance, we can now initialize an object of class `FuelType` as follows.

```
Car myCar = new Car();  
myCar.fuelType = new FuelType("diesel", 1.4, 0.01);
```

3.2 this

So far we saw that we can access fields and methods by dereferencing a local variable through a dot, or just by using the field and method names inside the class itself. Unfortunately, in this way field and local variables’ accesses look exactly the same, and if we have a parameter whose name is the same as a field, the parameter will hide the field. For instance, we would like that the parameters of the `FuelType` constructor have

the same name of the fields, but in this way they would hide the fields. All the OO programming languages provide a special identifier that points to the *current* object, aka the object on whom the method was called. For Java, this identifier is `this`. Therefore, by dereferencing `this` we can directly access the fields and methods of the current object. Let's rewrite class `FuelType` as follows.

```
class FuelType {
    String type;
    double costPerLiter;
    double litresPerKmH;
    FuelType(String type, double costPerLiter, double litresPerKmH) {
        this.type = type;
        this.costPerLiter = costPerLiter;
        this.litresPerKmH = litresPerKmH;
    }
}
```

In general, it is a good practice to always access fields using `this`, in order to make explicit that we want to reference the fields of the current objects instead of parameters or local variables. In the same way, `this` can be used to invoke methods on the current object. For instance, let's add a method `fullBrake()` to our class `Car` that sets the speed of the current car. When we had a break that exceeded the current speed, we set the current speed to zero in order to avoid having negative speeds. At this point, we might instead call method `fullBrake()` on the current object in such a case .

```
class Car {
    ....
    void fullBrake() {
        this.speed = 0;
    }
    void brake(double amount) {
        if(amount > speed)
            this.fullBrake();
        else speed = speed - amount;
    }
}
```

In addition, keyword `this` can be used to pass a pointer to a current object to other methods that can then access it as a parameter. We now implement a method in class `FuelType` that receives a `Car` and returns `true` if and only if the current fuel type is compatible with (aka, equal to) the one of the given car.

```
class FuelType {  
    ...  
    boolean isCompatible(Car c) {  
        return this.type.equals(c.fuelType.type);  
    }  
}
```

Now we modify the implementation of method `refuel` in order to use this method. This requires to call method `isCompatible` passing a reference to the current object (aka, `car`).

```
class Car {  
    ...  
    void refuel(FuelTank tank) {  
        if(tank.type.isCompatible(this))  
            fuel = fuel + tank.amount;  
    }  
}
```

Last but not least, keyword *this* can be used to call a constructor from another constructor. So far, we have seen classes where at most one constructor was implemented. However, one might want to implement several constructors with different parameters. For instance, we might implement a constructor of a `FuelTank` that receives both amount and fuel type, and another one that receives only the fuel type and sets the amount to zero.

```
class FuelTank {  
    FuelType type;  
    double amount;  
    FuelTank(FuelType type, double amount) {  
        this.type = type;  
        this.amount = amount;  
    }  
    FuelTank(FuelType type) {  
        this.type = type;  
        this.amount = 0.0;  
    }  
}
```

However, this code looks pretty bad since a part of the code is duplicated (the assignment of field `type`). A cleaner solution would be that the second constructor invokes the first one passing the given type, and an amount of zero litres. We can do this by invoking `this(type, 0.0)` as first statement of the second constructor:

```
class FuelTank {  
    FuelTank(FuelType type, double amount) {...}  
    FuelTank(FuelType type) {  
        this(type, 0.0);  
    }  
}
```

Such invocation of a constructor can be performed only (i) from another constructor, and (ii) as the first statement of the constructor. In all the other scenarios, it is not possible to invoke a constructor, since this is a special method that can be executed (at most) once when an object is created.

3.3 Final fields

Another modifier widely used on fields is `final`. A final field must be assigned only once in the constructors, and it cannot be reassigned later. Note that the field might be assigned when it is declared, or inside the constructor (but not in both). Other methods of the class can only read but not write (aka assign) final fields. For instance, we want to enforce that the name of the fuel and the number of liters per km/h consumed cannot be modified after a `FuelType` has been constructor, while instead the price might change over time. Such a constraint can be achieved by declaring fields `type` and `litresPerKmH` as `final`.

```
class FuelType {  
    final String type;  
    double costPerLiter;  
    final double litresPerKmH;  
    FuelType(String type, double costPerLiter, double litresPerKmH) {  
        this.type = type;  
        this.costPerLiter = costPerLiter;  
        this.litresPerKmH = litresPerKmH;  
    }  
}
```

3.4 Fields, parameters, and local variables

At this point it is important to understand and underline the differences between fields, parameters, and local variables. In particular:

- Fields are declared inside the class (and not inside the code of methods), and each time a class is instantiated the memory allocated in the heap contains the field. Fields can be read and written by dereferencing an object (e.g., pointed by a local

variable), or just by using their name inside a method (but this is a bad practice, and it is preferable to access the field through the keyword `this`);

- Local variables are defined inside the body of a method, allocated inside the local stack of the method, and thus they are removed after the execution of the method. They can be read and written inside the method by using their name.
- Parameters are declared inside the method declaration. Like local variables, they are removed after the execution of the method, and they can be both read and written. However, writing a parameter is considered a bad practice and it should be avoided.

We should always keep in mind that fields are intended to represent the state of an object, parameters to pass information to methods, and local variables to store the results of local computation inside a method. In Java, the fact that a field of the current object can be accessed without using the `this` keyword (e.g., `this.speed`) is quite misleading, since it gives the impression that fields and local variables are similar. Therefore, we should always access fields of the current object by prepending the `this` keyword.

3.5 Value types

Java (like the most part of existing programming languages) provides a set of built in (primitive) types. These types are intended to provide some base representation of data, and they usually represent numerical values. In particular, we have `int`, `byte`, `short`, `long`, `float`, `double`, `boolean`, and `char`. All these types start with a lower-case character, and in this way we distinguish them from the other class types (that, by convention, should start with an upper-case character). When we declare a local variable or parameter with one of these types, the bytes required by the specific type are allocated, and the value is stored in the local stack. Consider for instance the following snippet of code:

```
double costPerLiter = 1.4;  
double litresPerKmH = 0.01;  
FuelType diesel = new FuelType("diesel", costPerLiter, litresPerKmH);
```

Here two 64 bit floating point variables are allocated in the local stack and initialized by the first two statements. When these variables are passed to a method as parameters (that is, to the constructor of `FuelType`), the variable is passed *by value*, that is, the constructor will receive the values 1.4 and 0.01, and not a pointer to some memory containing these values. This means that, even if the constructor assigns some arbitrary values to the second and third parameters, these values won't be seen by the caller (that

is, our code snippet). Therefore, primitive types are passed to methods by value, and this is why we call them value types.

3.6 Reference types

Instead, all other types in Java are passed *by reference*, and therefore we call them reference types. These types are usually arrays or classes. By reference we mean that a pointer to the object or array is passed to the method (and not a copy of the object or array). In this way, if the method writes a value in some cells of the array or in some fields, the caller will see the value written by the method.

```
class Car {  
    ...  
    void refuel(FuelTank tank) {  
        if(tank.type.isCompatible(this)) {  
            fuel = fuel + tank.amount;  
            tank.amount = 0;  
        }  
    }  
    static void main(String[] args) {  
        FuelType diesel = new FuelType("diesel", 1.4, 0.01);  
        FuelTank tank = new FuelTank(diesel, 2.0);  
        Car myCar = new Car(diesel);  
        myCar.refuel(tank);  
    }  
}
```

For instance, in the code snippet above, at the end of the execution of the main method the tank will contain no fuel. Let's imagine that FuelTank object was allocated at reference #123. Then a reference to #123 is passed to refuel; so when refuel assigns zero to field amount it operates on object stored at #123. Therefore, after the execution of the method the assignment to the field will persist in the object stored in the heap at reference #123.

3.7 Aliasing

The discussion above leads to the concept of aliasing. We talk about aliasing when we have two different names for the same thing (aka, object in our context). For instance, if we assign a field using the first name, the results of this assignment will be visible (as a side-effect) through the second name as well. In OO programming languages, aliasing is

- an extremely powerful and expressive feature, since it allows to pass objects

around, and the callees can change the state of the object keeping its logical coherence, and

- an extremely dangerous and bug-prone feature, since it does not allow to reason modularly on a method: once we have a method call, we cannot know what happened to all the objects reachable from the callee without looking at its code.

When we want to prevent undesired side-effects we might (i) pass a copy instead of the object itself to the callee (but it is not always possible to deep copy a whole object), or (ii) define all the fields as final in order to deny the possibility of writing fields (but if a final field points to an object with some non final fields, than the contained object might be still modified).

The existence of aliasing led to the development of several architectural patterns in object-oriented programs in order to limit or exploit such a feature. These topics belong to more advanced courses on OO programming languages and design patterns, and therefore are not discussed in this book.

3.8 Garbage collection in Java

In this chapter, we discussed how objects are instantiated, and they can be accessed. The careful reader would have noticed that we do not have *free* pointers to the memory, where by free we mean that we can access arbitrary cells of the memory through pointer arithmetics (like `*(obj+4)`). The only way we have to access the heap is to dereference an object and access one of its fields. In all the cases where direct access to the memory is required (e.g., for efficiency reasons), Java allows defining native methods, that is, methods written in other programming languages (such as C) and interfaced in some ways (aka, the Java Native Interface) with the Java code. Other programming languages provide different solutions:

- in C# we can tag a portion of code as unsafe, and inside such portion the memory can be freely accessed through pointer arithmetics;
- in C++ we can access memory like in C programs as well as through objects' constructs like in Java.

The main advantage of the Java approach is that the runtime environment can always trace what is reachable by the program and what is not. In this way, Java programs do not need to explicitly deallocate memory (and indeed they cannot free memory at all), and they cannot contain buffer overrun (that is, statements that read portions of the memory that were not previously allocated). The memory is deallocated under the hood

by the garbage collector, which checks what parts of the heap are not anymore reachable from the program and deallocates it. On the one hand, the presence of the garbage collector eases the life of the developer that does not need to take care anymore of the memory management. On the other hand, the garbage collector might take quite some time when analyzing the memory (in particular, for programs that allocate a lot of memory), and therefore the time of the execution of a program might vary unexpectedly based on when and how the garbage collector runs.

3.9 Exercise

Given the CarID class, that represents the car identity:

- if necessary, apply the final modifier to the fields.
- implement the constructor of CarID
- implement isSameLicensePlate, that is a method that return true, if a given car has the same license plate of the current object

```
class CarID {  
  
    String make;  
    String model;  
    String licensePlate;  
    String VIN;  
  
    boolean isSameLicensePlate (Car car) {  
        ...  
    }  
    ...  
}
```

Chapter 4: Encapsulation and Information Hiding

The main goal of OO programming languages is to develop software that can be reused in many different contexts. In order to achieve such a goal, we need to hide all the internal details of the implementation (that is, how a task is performed) and expose only the information that is needed by a user of the system (that is, what task is performed). In this chapter we will introduce the mechanisms that allow us to hide a part of the implementation.

4.1 Packages

So far we put all the classes we implemented in the same software “unit”. OO programming languages offer means to group classes that conceptually belong to the same unit. In particular, Java supports packages. Each class can belong to a specific package, and if it wants to use classes belonging to other packages it needs to explicitly import them. The name of the package should contain the name of the organization, as well as the specific project of the code. Package names are structured as a sequence of names, and they are structured like reversed Internet URLs.

Let’s imagine that in our example we group the classes about fuel types and tanks into a package `fuel`, while the car will stay in a package named `vehicles`. We are at the Italian (suffix `it`) university of Venice (`unive`), in the DAIS department, the course is called “Programmazione ad Oggetti mod. 1”. Therefore the two package names will be `it.unive.dais.fuel` and `it.unive.dais.vehicles`. The Java files must be placed in a directory structure that reflects the package name (that is, `Car.java` must stay in `it/unive/dais/vehicle` while `FuelType.java` and `FuelTank.java` in `it/unive/dais/fuel`) to allow the Java compiler to retrieve them. Then the first line of the Java file must contain the keyword `package` followed by the name of the package. In addition, our `Car` class must explicitly import the fuel classes in order to use them as static types of fields (`FuelType`) and parameters (`FuelTank`). This can be done by using the `import` keyword after the declaration of the package, and before the declaration of the class. The code of class `Car` will therefore look as follows:

```
package it.unive.dais.pol.vehicle;
import it.unive.dais.pol.fuel.FuelType;
import it.unive.dais.pol.fuel.FuelTank;
class Car {
    double speed = 0;
    double fuel = 0;
    FuelType fuelType;
    void refuel(FuelTank tank) {... }
}
```

The Java compiler requires that all the imported classes are compiled together, or they are present in the given classpath (that is, a special path that might be passed to the compiler in order to let it know where it can retrieve classes that were previously compiled).

Note that the import statement supports names ending with an asterisk at the end. In this way, we can import together all the classes inside a package. For instance, we might replace the two import statements in class Car with `import it.unive.dais.pol.Fuel.*`

4.2 Encapsulation

Briefly, a program is well encapsulated if the data is bundled with the methods that operate on it. Classes allow us to define units of code that are composed of data (fields) and functionalities (methods), and therefore they are needed to encapsulate code. In addition, we can create capsules containing many classes with packages. Even more, a library can be seen as a capsule containing many packages. Therefore, the concept of encapsulation can be applied at different levels (at least, classes, packages, and libraries). Encapsulation is a key concept in OO programs; indeed, the main goal of OO programming languages is to provide the developers with means to encapsulate their code. However, a badly designed OO program might provide almost no encapsulation.

A developer needs to carefully think about the structure of the program in order to achieve a good level of encapsulation. Let's go back to the example of cars and fuel: encapsulating the speed and the fuel amount in the Car class looked like a natural choice, and indeed it is the right way to proceed. In fact, speed and fuel amount are tightly connected with functionalities like acceleration, brake, and refueling. Let's imagine that instead we chose to store the speed in FuelType or FuelTank⁴: how could we manage to accelerate or break a car? We would need to have access to such information in the other class, and therefore the functionality would not be bundled with the data.

⁴ Let's ignore the problems that might arise about the duplication of such data or its sharing with other instances of Car

Now one might correctly ask: why should I care about all this? Why should we not have everything available everywhere (like in C)? The main problem is that with such an approach the whole code is a unique “block” and cannot be separated into smaller units. In particular, we would have a lot of (potentially useless) dependencies, and this might become problematic during the maintenance of the code, since it might require to modify the whole code base even for small changes. For instance, at a certain point we might consider to represent the speed of a car not only with an amount (double value), but also with the measure unit (e.g., km/h or mph). If we encapsulate such information inside the car, we would need to change the code of only this class. Instead, if we put this data in FuelType or FuelTank, we would need to change both this class and Car, that is, two different packages.

4.3 Information hiding

However, despite the fact that we might well encapsulate our code, so far we did not have a means to restrict the visibility of some portions of our data and functionalities. This is exactly the goal of information hiding. The idea is to provide some mechanism in order to hide some class components. In this way, we can hide what is just an implementation detail, and expose only the interface that provides all the components that are expected by the users of our software and nothing else. This approach brings to various benefits:

- The dependencies between different components are minimized, since an external user of our program can see only the parts of our implementation that we decided to expose;
- A user can see only the components that were intended to be the interface of the piece of the software, and not its implementation details. In this way the user needs to explore only the interface components in order to understand how he/she could use the software, and not all the other components; and
- We can modularly reason in the local components of the software without the need to consider how this is used. For instance, if a field is always assigned to values greater than or equal to zero and it cannot be accessed from outside our class, then we know it will always be greater than or equal to zero regardless of how our class will be used.

But how encapsulation and information hiding are related? Generally speaking, an ideal encapsulation allows to maximize the information hiding of the software. Note that the

two concepts are clearly distinct: a program might be well encapsulated but without any information hiding, and another one might be terribly encapsulated but pushing as much as possible information hiding.

4.4 Access modifiers in Java

	Same class	Same package	Subclasses	Everywhere
public	👍	👍	👍	👍
protected	👍	👍	👍	👎
<default>	👍	👍	👎	👎
private	👍	👎	👎	👎

Java provides information hiding through access modifiers. Like other modifiers (e.g., static and final) these keywords can be applied to the different components of an OO program. In particular, we focus our attention on the class components, that is, fields and methods. The table above synthesizes all the access modifiers provided by Java and how they restrict the components' visibility.

In particular, Java identified 4 distinct levels of accessibility: inside the same class, the same package, the subclasses, or everywhere. For now, we will skip subclasses since these will be later introduced and discussed when we'll deal with inheritance. There are then the four keywords: private (components visible only from the current class), <default> (no keyword, components visible from the same package), protected (visible also from subclasses), and public (visible from everywhere). Such keywords can be applied to fields and methods.

In this way, we can have different views of the same class based on where we are. The public components represent the interface of our class: all the components that are intended to be what our software offers to its clients should be public. On the other hand, private components represent information and functionalities that are internal to a specific class, and they are not interesting (and indeed they should be hidden) to the clients of our software. Finally, components with default visibility need to be shared among the various classes of the same package (aka software unit), and therefore they are relevant for who is developing the specific software, but not for external clients.

At this point a small digression about private components is needed. In Java, private works at class level. This means the class can access the private fields of not only the

current object (pointed by *this*), but also of other objects instances of the current class. Consider for instance the following snippet of code.

```
class A {  
    private int private_field;  
    int foo(A a) {  
        return this.private_field + a.private_field;  
    }  
}
```

Such code compiles and correctly executes in Java because the `private` modifier applies at class level, and therefore the access `a.private_field` is allowed.

Let's go back now to our `Car` class, and let's push as much as possible information hiding in it. First of all we should ask ourselves: what functionalities do we want to provide to our clients? Accelerating, braking, refueling and constructing cars are definitely needed. Instead, the fields should be all private, since for instance we do not want to allow somebody to arbitrarily change the speed of our car (without consuming any fuel).

```
class Car {  
    private double speed;  
    private double fuel;  
    private FuelType fuelType;  
    public Car(FuelType f) {...}  
    public void refuel(FuelTank tank) {...}  
    public void fullBrake() {...}  
    public void brake(double amount) {...}  
    public void accelerate(double amount) {...}  
}
```

So far so good. Now we apply the same approach to `FuelType` and `FuelTanks`. However, apart from constructors, these classes contain only fields.

```
class FuelTank {  
    private FuelType type;  
    private double amount;  
    private int id;  
    public FuelTank(FuelType type, double amount) {...}  
    public FuelTank(FuelType type) {...}  
}  
  
class FuelType {  
    private final String type;  
    private double costPerLiter;  
    private final double litresPerKmH;  
}
```

The code does not compile anymore, since (i) class `Car` needs to access the name of the fuel type, the ratio of liters per km/h, etc., (ii) class `FuelType` needs to access to the fuel

type in a car in order to check if it is compatible with the fuel type of the current object. So, what's the solution? Put all the fields back to public? Obviously not!

4.5 Getters and setters

The main problem of having public fields is that (i) arbitrary values might be assigned to them, (ii) if the field stores an object reference, this might be leaked outside and we might incur in unexpected side-effects, and (iii) we won't be able in the future to change how we change our internal representation of data since we would expose those fields as interface of our classe. The common solution is to have ad hoc methods that allow to read and write fields' values. Such a solution consists of getters and setters in Java.

Briefly, a getter is a method `get<field_name>()` that simply returns the value of `<field_name>`, while a setter is a method `set<field_name>(<value>)` that assigns `<value>` to `<field_name>`. Indeed, a getter might do something different, like returning a copy of the object pointed by a field, while a setter might modify the field only if the given value respects some constraints. In this way we can overcome the problems pointed above: a file might be only read and not written, the value returned from the getter might be equivalent from the one stored in the field but through a distinct object, and we can change our fields as we wish updating the behaviors of the getters and setters.

Let's go back again to our example. First of all, we need to add getters to `FuelType` to access the string containing the ratio liters per km/h, and `FuelTank` to get the type and amount of fuel stored in the tank. However, when refueling class `Car` sets the amount of fuel in a tank to zero. Should we provide a setter as well for this field? Indeed, there is no need to allow somebody to set an arbitrary value there, since the only assignment we need to perform is to empty the tank. Therefore, we implement a method that empties the tank instead of a setter.


```
class FuelTank {
    public FuelType getFuelType() {
        return this.type;
    }
    public double getAmount() {
        return this.amount;
    }
    public void emptyTank() {
        this.amount = 0;
    }
}

class FuelType {
    public double getLitresPerKmH() {
        return this.litresPerKmH;
    }
    boolean isCompatible(Car c) {
        return this.type.equals(c.fuelType.type);
    }
}
```

We are almost done! Unfortunately, we still have a problem: `FuelType.isCompatible(Car)` requires access to the field `fuelType` of the car to be able to answer.

4.6 Choosing how to encapsulate

Often when we maximize information hiding we discover that something was wrong with our initial idea and implementation. In fact, we might discover that we cannot restrict the visibility of something because the initial approach did not correctly encapsulate our code. A common workflow for an OO program might be:

- 1) Decide (before starting to write code) what components are part of the interface and set them to public
- 2) All the rest should be private
- 3) Step by step analyze why something that we need to access in our code is instead private, and then decide if we should relax its visibility, or instead refactor the code in order to improve its encapsulation.

Going back to our example, why should an instance of `FuelType` need to know about the existence of cars, and access one of its fields? While cars need to know something about the fuel (otherwise... they would not be cars!), fuel exists despite the fact that it is used by cars. While, as usual, our examples are tiny and ad-hoc, this discussion raises a crucial point that is often observed in real-world software. Two components (packages

autovehicles and fuel) mutually depend on each other, even if this should not be the case. Such circular dependencies obligate us to always compile and deliver the two components together, and this might be an important limitation in the long run. For instance, we might want to re-use our fuel implementation for airplanes. But if our implementation depends on autovehicles, we will need to ship this part as well! Therefore, we should have such circular dependencies only when strictly needed, that is, when the two classes are inherently connected and need to be part of the same capsule of code.

This is not the case for cars and fuel. Therefore we need to re-thing the interface of our implementation, and in particular of method `isCompatible`. Do you see any good reason why `isCompatible` should receive a `Car` instead of a `FuelType`⁵? I don't. So let's modify `isCompatible` by receiving a `FuelType` instance, and class `Car` accordingly.

```
class FuelType {
    public boolean isCompatible(FuelType other) {
        return other.type.equals(this.type);
    }
}

class Car {
    public void refuel(FuelTank tank) {
        if(this.fuelType.isCompatible(tank.getType())) {
            fuel = fuel + tank.getAmount();
            tank.emptyTank();
        }
    }
}
```

I hope you'll agree that this solution is definitely better (in terms of encapsulation) of the initial one, and that by now we got the best encapsulation for our (tiny ad-hoc) example.

4.7 Artifacts

After the compilation and build of a program, an artifact is produced. This artifact always contains the results of the compilation of the program (e.g., Java bytecode), and it might contain additional information, such as support files (e.g., images to be rendered by the program), structured data about the program (e.g., Java manifests), etc.. An artifact can be seen as an additional level of encapsulation, and it is the one exposed to users of our software⁶. An artifact usually encapsulates several packages together, it can

⁵ Except the fact that it was a good way to show how we can use the keyword `this` to pass a pointer to the current object, but that was a different part of this book :)

⁶ By user we mean both final users (that run the program and use its functionalities to perform

be downloaded and installed, and it is meant to be shared with a community.

When talking about Java programs, the most common artifacts are jar files⁷. The main idea of such an artifact is that a developer might load it inside his/her Java application, and use our classes as a library. Nowadays, there are dozens of millions of jar files published in the main repositories (such as Maven - <https://mvnrepository.com/>). The Java Development Kit provides a utility (jar.exe) that allows to pack together several .class files into a unique jar archive. A jar file is nothing else than a zipped file containing (i) all the .class files obtained by the compilation of our source code, (ii) all the resource files (such as property files), and (iii) a manifest file. For instance, the following snippet of code contains the manifest file of the runtime environment of Oracle Java 11.0.6

```
Manifest-Version: 1.0
Specification-Title: Java Platform API Specification
Specification-Version: 11
Specification-Vendor: Oracle Corporation
Implementation-Title: Java Runtime Environment
Implementation-Version: 11.0.6
Implementation-Vendor: Oracle Corporation
Created-By: 10 (Oracle Corporation)
```

Essentially, the manifest file contains information about the name of the artifact, who implemented it, and its version number. Note that the name and version number should always uniquely identify an artifact, that is, it is not possible to have two different artifacts (that is, two artifacts containing different files) with the same name and version number. In addition, a jar file might contain the full name (aka, package+class name) of the main class. In such a case, the artifact can be also ran through the command “java -jar <file>”.

Overall, the encapsulation of an artifact should be such that everything that is public can be accessed and used, while the packages are intended to be sealed. This means that the user of an artifact should not write classes belonging to the packages of the artifact, and therefore he/she should not access everything that has default (aka, package) visibility.

4.8 Exercise

Given the *CarID* class, that represents the car identity:

some tasks) and software developers (that use some functionalities inside their software using the API of our program).

⁷ In this book we do not discuss Java modules (introduced with Java 9).

- Add a package to CarID conforms to that of the Car class of the previous sections.
- Add a CarID field into Car class and, if necessary, add the correct import.
- If necessary, apply the access modifiers *to the CarID class and its components*
- Implement the getter and setter for the fields of *CarID* class

```
class CarID {  
  
    final String make;  
    final String model;  
    String licensePlate;  
    final String VIN;  
  
    CarID(String make, String model, String licensePlate, String VIN){  
        this.make = make;  
        this.model = model;  
        this.licensePlate = licensePlate;  
        this.VIN = VIN;  
    }  
  
    boolean isSameLicensePlate(String licensePlate) {  
        return this.licensePlate.equals(licensePlate);  
    }  
  
    ...  
}
```

Chapter 5: Documentation

When using an artifact (or anyway a piece of software written by external developers), a developer should never look at the code, but only at signatures and documentation. For instance, if we would package together the classes discussed so far, we would expect that a user would see something like the following snippet of code for class Car:

```
class Car {  
  
    //Add the given amount to the fuel tank  
    void refuel(FuelTank tank) {...}  
  
    //Increment the speed  
    void accelerate(double a) {...}  
  
    //Stop the car  
    void fullBrake() {...}  
  
    //Slow down the car  
    void brake(double amount) {...}  
}
```

The method signature tells only very partial and potentially ambiguous information about what a method is intended to do, since its structure is fixed and very limited. On the other hand, text comments might contain any kind of information, but since they do not have a well defined structure, they need to be manually processed and they might not be exhaustive. Anyway, documentation is an integral part of an artifact that is intended to be used by software developers. Therefore, object-oriented programming languages often provide means to properly structure and format comments that are part of the documentation of the artifact.

5.1 Comments

All programming languages provide some means to add comments to code. For instance, in Java we write single line comments by starting the comment with `//`, or multiline comments starting with `/*` and ending with `*/`

It is important to underline that those comments explain what the single lines of code do. For instance, let's imagine that we want to implement a method in class `FuelType` that updates the cost per liter. However, such setter should perform some checks, such as the

sign of the new price per liter, or that it does not change too much w.r.t. the previous cost. A possible implementation might be the following one.

```
public boolean setCostPerLiter(double costPerLiter) {
    if(costPerLiter < 0) {
        //Setting the field to zero because we should not earn money when refueling...
        this.costPerLiter = 0;
        return false;
    }
    else {
        //Compute the difference in percentage
        double difference = Math.abs(this.costPerLiter-costPerLiter);
        double differencePercentage = difference/this.costPerLiter;
        //Updating the cost only if the change is below 20%
        if(differencePercentage < 0.20) {
            this.costPerLiter = costPerLiter;
            return true;
        }
        else return false;
    }
}
```

As you can see, we added some comments inside the code. Those comments refer to implementation details that are not exposed to the developers that will be using our code.

5.2 Comments as documentation

Indeed, we would like to add some comments to our code that will be shipped as documentation of our artifact. In particular, we need to document all the components of our software, and in particular our classes, fields and methods. Those comments should have a different format and they should be structured in some ways.

In Java, we can document our code using the Javadoc standard. Javadoc comments are multi-line comments that starts with `/**` (instead of `/*`) and they are closed with `*/`. They should precede the component they document. We might document our Car class as follows.

```
/**
 * This class represents a car
 */
class Car {
    /**
     * Refuel the car with the fuel in the given tank.
     */
    void refuel(FuelTank tank) {...}
    /**
     * Accelerate the car of the given amount of km/h
     */
    void accelerate(double a) {...}
    /**
     * Stop the car
     */
    void fullBrake() {...}
    /**
     * Lower the speed of the car of the given amount
     */
    void brake(double amount) {...}
}
```

5.3 From comments to documentation

We stressed several times that a user of our code should not look at our code, but instead rely on its abstraction, and such abstraction should be composed by signatures and documentation. Therefore, writing documentation as comments allows us to have everything together (and to bind some comments to a precise version of our software), but it still obliges a user to go through our code. Therefore, OO programming languages provide various means to deliver and read such documentation without going through the code.

A first way is to deliver the documentation as some HTML pages or a manual. For instance, the JDK provides a command line tool (javadoc) that, given some classes commented following the Javadoc standard, produces several HTML pages containing all the Javadoc documentation. For instance, in the example above Javadoc produces the following HTML page for class Car.

```
public class Car
extends java.lang.Object
```

This class represents a car

Constructor Summary

Constructors
Constructor and Description
<code>Car()</code>

Method Summary

All Methods	Instance Methods	Concrete Methods
Modifier and Type	Method and Description	
void	accelerate (double amount) Accelerate the car of the given amount of km/h	
void	brake (double amount) Lower the speed of the car of the given amount	
void	fullBrake () Stop the car	
void	refuel (FuelTank tank) Refuel the car with the fuel in the given tank.	

While such documentation might be interesting when navigating the whole software, a user is also interested to directly know the documentation of a single component when he/she is using it. Since Javadoc comments refer to a specific component, tools that facilitate the development of OO programs (such as Integrated Development Environments) might show just this information on the fly to the developer. For instance, IntelliJ IDEA shows the pop-up on the right when some code invokes method `brake` over an object of type `Car`.

```
it.unive.dais.po1.autovehicles.Car
public void brake(double amount)

Lower the speed of the car of the given
amount
```

5.4 Structured documentation

The components we are documenting have a structure, and we would like to attach some

comments to specific components. Javadoc adopted the tag @<name> in order to denote the structure of some comments.

On classes, we can enrich our documentation using the following tags:

- @author to specify who wrote the class
- @since to specify since which version of the software the class has been present
- @version to specify the current version of the class. We strongly discourage the use of this tag, since it might conflict with the version reported on the manifest attached to the artifact, and it would need to be updated each time there is a new version of the software.

Class Car might be documented as follows.

<pre>/** * This class represents a car * * @author Pietro Ferrara * @since 1.0 */ class Car {...}</pre>	<pre>public class Car extends java.lang.Object This class represents a car Since: 1.0</pre>
--	---

Note that by default the information of the tag @author (as well as others) is not reported in the documentation produced by Javadoc. The command line tool has an option that enables this.

Similarly, we can document methods with:

- @param <name> to document a specific method parameter
- @return to specify what a method returns

For instance, our complex setCostPerLiter method might be documented as follows.

<pre>/** * This method sets the cost per liter to the given price if the change of the price * is reasonable w.r.t. to the changes in the fuel market. * If the cost per liter is negative, it sets the price to zero. * * @param costPerLiter the new cost per liter of the current fuel type * @return true if and only if the cost per liter of the fuel is set to the * given price after the execution of this method */ public boolean setCostPerLiter(double costPerLiter) {...}</pre>

setCostPerLiter
<pre>public boolean setCostPerLiter(double costPerLiter)</pre>
<p>This method sets the cost per liter to the given price if the change of the price is reasonable w.r.t. to the changes in the fuel market. If the cost per liter is negative, it sets the price to zero.</p>
<p>Parameters:</p> <p>costPerLiter - the new cost per liter of the current fuel type</p>
<p>Returns:</p> <p>true if and only if the cost per liter of the fuel is set to the given price after the execution of this method</p>

It is important to note that in general the documentation should not report implementation details, and it should focus only on what a developer using our code might be interested in. For instance, we decided to hide the fact that a change above 20% was not reasonable, as we considered it as an implementation detail (but one might consider it as part of the specification as well and put it in the documentation).

5.5 What should we document?

Finally, a big question arises: we know how to document, but what should we document? Briefly, all the components (aka, classes, fields, and methods so far) that are part of the interface of our artifact. So what is the interface of our artifact? Javadoc allows one to specify if it wants to include the documentation of components that are public, protected (that will be introduced later in this book), default/package, or private. Therefore, it is up to us to decide what to include. However, there is no doubt that public components are part of the interface, while private components are not. But what about components that have default visibility and so they are visible only inside the package? In our opinion, when we deliver an artifact the idea is that packages are *sealed* (that is, they cannot be augmented with more components), since the users of the artifact will be external to our organization and they should not write classes using our package names (even if they can do so). Therefore, we believe that such components might be not documented, and it is a good practice not to report them in the official documentation. But this is just an opinion, and the technology we have (Java in particular) allows us to produce documentation on those components.

Chapter 6: Design by contract

Generally speaking, a contract can be seen as an agreement between two parties that establishes the rights and duties of both parties. For instance, when somebody buys something paying some money, he/she has the right to receive the goods, and the obligation of paying the given amount. Viceversa, the seller has the right to receive the money, and the obligation of giving the goods.

Moving towards OO programs, a capsule can be seen as a contract. The developer of the capsule has the right of receiving some values, and the obligation of providing back the results of the computation as specified by the capsule APIs (e.g., method signature) and the documentation (e.g., Javadoc comments). Viceversa, a client of an OO program has the obligation of passing some values, and the right to get back the results of the computations. In the rest of this chapter, we will focus on capsules at class level, but the same concepts can apply for a whole package or artifact (e.g., a jar file).

6.1 Syntax elements as a contract

The syntax of a program is the text of the source code we write, but not its meaning (aka, semantics). When talking about the interface of a class, the syntax is composed by the class name, the names and static types of the fields, and the declaration of the methods (that is, name of the method, names and static types of the parameters, and return type). Consider again the interface of class Car.

```
class Car {  
    void refuel(FuelTank tank) {...}  
    void accelerate(double a) {...}  
    void fullBrake() {...}  
    void brake(double amount) {...}  
}
```

The interface of this class establishes a contract where the client has a class Car with four methods available (refuel, accelerate, fullBrake, and brake). Each method defines what it requires (e.g., method refuel requires a FuelTank object) and what it ensures (... nothing, since all the return types are void). We might see the name of the method as part of the contract, so for instance we might infer that after we invoke method fullBrake on a Car, the Car is not moving ahead anymore.

Overall, we can see that these contracts are extremely precise: the types of the values that are required and ensured are exactly defined, and there is no ambiguity about that (e.g., if method `brake` asks for a `double`, a `double` value must be passed!). This is very important because we know that if both parties respect the syntax of the class, then the program can compile and run. Otherwise, the compiler will fail and the code cannot be executed.

However, these syntactic contracts are not so expressive: what happens after we `accelerate`? And what is the meaning of the `double` value received by `accelerate` (since its name is `a`, it is ambiguous what it represents)? How can we specify that after accelerating the car increases its speed of the given amount? We cannot through syntactic elements, and this is why we previously introduced Javadoc documentation.

6.2 Documentation as a contract

Documentation is definitely part of the contract, since it is exposed to the client, and it states what the program is assumed to expect and perform. We have seen that Java provides the Javadoc standard in order to document the code; such special comments can be therefore seen as part of the contracts. Let's consider again class `Car` with the Javadoc comments written in the previous chapter.

```
/**
 * This class represents a car
 */
class Car {
    /**
     * Refuel the car with the fuel in the given tank.
     */
    void refuel(FuelTank tank) {...}
    /**
     * Accelerate the car of the given amount of km/h
     */
    void accelerate(double a) {...}
    /**
     * Stop the car
     */
    void fullBrake() {...}
    /**
     * Lower the speed of the car of the given amount
     */
    void brake(double amount) {...}
}
```

The documentation adds semantic information about what a method requires and what it ensures. Since it is written in plain language, the documentation can be extremely expressive: we can write whatever we want there, and we expect the counterpart (aka, the client of our code) to be able to understand our description. However, the documentation might be partial, ambiguous, and/or misleading like any text written in plain language, since it does not have a predefined structure and a formal semantics. For instance, what do we mean by stopping the car in the example above? We have a general idea about this, but after a full brake is the car standing, or moving a bit backward? And when I accelerate, do I always increase the speed of the given amount, or it depends on external factors (e.g., having enough fuel)?

Since documentation is written in plain text, such contracts might contain more or less details. The depth of the documentation is left to the developer, and we think there is not a right level of specification in such a case. If the documentation is extremely precise, it will end up to be equivalent to the implementation, but we do not want to expose the implementation to a client since it would be too complex to reason on our program. On the opposite side, a vague documentation like the above one of class `Car` might not allow the client to understand what the class provides. Let's instead consider the setter of the cost per liter of the fuel, and the documentation we wrote.

```
/**
 * This method sets the cost per liter to the given price if the change of the price
 * is reasonable w.r.t. to the changes in the fuel market.
 * If the cost per liter is negative, it sets the price to zero.
 *
 * @param costPerLiter the new cost per liter of the current fuel type
 * @return true if and only if the cost per liter of the fuel is set to the
 *         given price after the execution of this method
 */
public boolean setCostPerLiter(double costPerLiter) {...}
```

In this case, the contract is pretty clear: it sets the cost per liter if the change is reasonable, and if the new price is negative it sets it to zero. However, it is more abstract than the implementation: the documentation does not talk about a difference of 20% as the threshold for a reasonable change of price.

It is time to go one step further: the design by contract methodology specifies contracts as mathematical formulae, and it comprises three main types of contracts: preconditions, postconditions, and object invariants. While these components are not part of the Java programming language, there are some of its extensions that support it. We refer the

interested reader to the Java Modelling Language⁸ for more details. In the rest of the chapter, we will adopt a notation that is similar but less strict and formal than the one adopted by the Java Modelling Language, since the goal of this chapter is not to learn how to specify a class to verify it, but to understand how we can formally reason about a class.

6.3 Preconditions

A precondition is a condition that should be satisfied each time we call a method. Intuitively, this is what is required by our method, and it is what the clients should provide when he/she uses our implementation. A precondition is something which our code che rely on, and that should be satisfied by our clients. A precondition is specified by using the keyword `@requires` in the documentation. For instance, when somebody invokes method `accelerate`, we expect that the amount of km/h is greater than or equal to zero.

```
/**  
 * Accelerate the car of the given amount of km/h  
 * @requires a >= 0  
 */  
void accelerate(double a) {...}
```

This precondition is definitely correct. Now a question arises: why not $a \geq 10$, or $a \geq 9999$? From our perspective (aka, who implements the class) all these preconditions are good, since the code will always execute correctly. However, from the perspective of a client they are not: why should you work only if I accelerate by at least 10km/h? This condition is definitely too restrictive to establish a fair contract between the two parts.

In general, we say that the preconditions we write are the weakest ones that ensures the correct execution of our method. Therefore, since $a \geq 9999 \Rightarrow a \geq 10 \Rightarrow a \geq 0$, we choose $a \geq 0$ as precondition of our method.

6.4 Postconditions

A postcondition should be satisfied all the times we end the execution of our method. Intuitively, it specifies what our method guarantees to our clients after its execution. A postcondition is something that we need to guarantee, and that the clients of our code can rely on. A postcondition is specified in the documentation through the keyword

⁸ <https://www.cs.ucf.edu/~leavens/JML/index.shtml>

@ensures. For instance, our accelerate method ensures that after its execution the speed is increased by the given amount.

```
/**
 * Accelerate the car of the given amount of km/h
 * @requires a >= 0
 * @ensures speed == speed + a
 */
void accelerate(double a) {...}
```

Unfortunately, this postcondition is broken for at least three different reasons. First of all, let's consider the condition as standalone: how is it possible that the speed is equal to the speed plus the given amount? Indeed, what we wanted to specify is that the speed *after* the execution of the method is equal to the speed *before* the execution plus the given amount. Therefore, we need to distinguish the previous value (specified by pre(speed)) and the new one in the condition.

The second problem is that this contract is part of the documentation, but the field speed in Car is private and it is not exposed to clients. Since the client cannot see this field, it does not make sense to talk about this field in the postcondition. However, so far we did not implement any method that allows the client to see information about the car speed, and this is definitely something missing in our implementation. What would be the interest of a car that can accelerate and brake, if we cannot see its speed? As you can see, writing contracts might also help to find bugs and missing methods in our implementation. Therefore, let's implement and write the contracts of the getter of field speed.

```
/**
 * Return the speed of the car
 *
 * @ensures return >= 0
 */
int getSpeed() {
    return this.speed;
}
```

Note that we used the keyword return in the contract to denote the value returned by the method. In addition, since we cannot talk about field speed also in the contracts of the getter, we simply stated that the returned value is greater than or equal to zero. Now we can use this getter in the contracts of accelerate.

```
/**
 * Accelerate the car of the given amount of km/h
 * @requires a >= 0
 * @ensures getSpeed() == pre(getSpeed()) + a
 */
void accelerate(double a) {
    double fuelConsumed = a*fuelType.getLitresPerKmH();
    if(fuelConsumed < fuel) {
        speed = speed + amount;
        fuel = fuel - fuelConsumed;
    }
    else {
        double increaseSpeed = fuel / fuelType.getLitresPerKmH();
        speed = speed + increaseSpeed;
        fuel = 0;
    }
}
```

And now by looking at the implementation of `accelerate` we can identify the third problem: the implementation does not respect the postcondition! In fact, if there is not enough fuel, the speed after the execution of the method won't be augmented by the given amount, but only by the amount allowed by the remaining fuel. Note that the fact that there is enough fuel to accelerate is not a precondition of the method, since it works also if there is not enough fuel! In addition, like for the car speed, we didn't define yet a getter for the amount of fuel in the car. For the sake of simplicity, let's imagine that we added a method `getFuelAmount()` returning this information and annotated like `getSpeed()`. Now we can write a complete and correct postcondition!

```
/**
 * Accelerate the car of the given amount of km/h
 * @requires a >= 0
 * @ensures a*fuelType.getLitresPerKmH() < pre(getFuel()) =>
 *           getSpeed() == pre(getSpeed()) + a
 * @ensures a*fuelType.getLitresPerKmH() >= pre(getFuelAmount()) =>
 *           getSpeed() == pre(getSpeed()) +
 *           pre(getFuelAmount()) / fuelType.getLitresPerKmH()
 */
void accelerate(double a) {...}
```

At this point, the careful reader might argue that the postcondition is mostly a copy-and-paste of the implementation, and therefore it does not provide any abstraction w.r.t. to the implementation. While such observation is only partially true (e.g., the postcondition does not expose information about the fields, that indeed are an implementation detail), we agree that the level of detail of such a contract might be overwhelming and unnecessarily detailed for a client of our code.

Note that other postconditions are possible (like for preconditions). For instance, we might simply ensure that $\text{getSpeed()} \geq \text{pre}(\text{getSpeed()})$ AND $\text{getSpeed()} \leq \text{pre}(\text{getSpeed()}) + a$. This is definitely more abstract than the initial one, but it still provides some guarantees. In general, we say that the postcondition should be the strongest one that is enough for what our clients need. However, since we do not know who will use our class a priori, many solutions are possible.

6.5 Object invariants

Object invariants are the third and last type of contract we might write. Object invariants specify what is always true about our objects, that is, before and after calling all the methods of our class. Let's go back to the `getSpeed()` method.

```
/**
 * @ensures return >= 0
 */
int getSpeed() {
    return this.speed;
}
```

How do we know this method returns something that is always greater than or equal to zero? Because we have an (implicit for now) invariant that states that field `speed` is always greater than or equal to 0. Invariants can be specified in the documentation by the keyword `@invariant`.

```
/**
 * @invariant speed >= 0
 */
int speed;
```

Now how are we sure that `speed` always respects the invariant? For instance, using the last postcondition we discussed about method `accelerate` ($\text{getSpeed()} \geq \text{pre}(\text{getSpeed()})$ AND $\text{getSpeed()} \leq \text{pre}(\text{getSpeed()}) + a$) and the precondition ($a \geq 0$) as well as the invariant itself in the pre state, we can conclude that the invariant of `speed` will be always respected since $\text{getSpeed()} \geq \text{pre}(\text{getSpeed()})$ and $\text{pre}(\text{getSpeed()}) \geq 0$. Thanks to this invariant we know that the postcondition of method `getSpeed()` holds⁹.

Note that the invariant we discussed concerns a private field, and therefore this should not be part of the documentation. Since all the fields we defined so far are private (and we preferred to rely on getter and setter methods to manage them), in our examples

⁹ For the sake of simplicity, in this example we did not make a neat and formal distinction about `getSpeed()` and the field `speed` itself.

invariants are not intended to be part of the documentation, but instead internal implementation details that ease the reasoning about our code.

6.6 Popularity of the design-by-contract approach

Here, I have to be honest with you: the design-by-contract methodology never reached a relevant impact on industrial software, while it has been extensively applied on some specific programs. Why? Essentially, contracts require to spend quite some time to write things that are additional to the code (even if they largely overlap with it), are hardly part of the documentation (since clients usually prefers plain text than mathematical formulae), and they are quite hard to write (since they require a very formal mathematical approach).

However, understanding the basics of contracts is essentially to understand the deepest details of OO programming languages, and in particular to approach the more conceptual parts of this world such as the substitution principle.

6.7 Exercise

Given the `checkLicensePlateValidity` method and its description in natural language, provide the documentation of this method in accordance with the notions learned in Section 5 and Section 6.

Description:

The license plate in Italy must have a length of 7 alphanumeric characters of the CCNNCC type, where N is a digit and C is a different uppercase alphabetic letter from “I” and “O”, and the sequences CC must not be “EE” because the double “E” is not allowed.

```
static boolean checkLicensePlateValidity(Car car ) {  
  
    String licensePlate = car.getCarID().getLicensePlate();  
  
    if(!Character.isDigit(licensePlate.charAt(0))  
        || !Character.isDigit(licensePlate.charAt(1))  
        || !Character.isDigit(licensePlate.charAt(5))  
        || !Character.isDigit(licensePlate.charAt(6)))  
        return false;  
  
    if(!Character.isUpperCase(licensePlate.charAt(2))  
        || !Character.isUpperCase(licensePlate.charAt(3))  
        || !Character.isUpperCase(licensePlate.charAt(4)))  
        return false;  
  
    return !licensePlate.contains("I") && !licensePlate.contains("O")  
        && !licensePlate.contains("EE");  
}
```

Part 2: Polymorphism

Chapter 7: Inheritance

By now, we got familiar with what classes are and how we can structure our code properly following the object-oriented paradigm. Encapsulation allows us to define capsules (e.g., classes, packages, or artifacts) of code that define a clear interface (aka, contract) with the clients of our code abstracting the inner details of the implementation. This definitely eases the reuse of code written by others, but it does not give the possibility to extend or modify the functionalities defined in a capsule of code.

Let's consider the example of Car again. We want to implement a class to represent bicycles that can accelerate and brake, and trucks that can accelerate and break, refuel, and load and unload some charge. These implementations might look as follows.

```
public class Bicycle {
    private double speed = 0;
    public void accelerate(double a) {
        if(a>0)
            this.speed += a;
    }
    public void fullBreak() {
        this.speed = 0;
    }
}

class Truck {
    private double loadedCharge = 0.0;
    private double speed = 0;
    private double fuel = 0;
    private final FuelType fuelType;
    public Truck(FuelType f) {
        fuelType = f;
    }
    public void refuel(FuelTank tank) {
        fuel = fuel + tank.getAmount();
        tank.emptyTank();
    }
    public void fullBrake() {
        this.speed=0;
    }
    private double computeConsumedFuel(double speedInc, double litresPerKmH) {
        return speedInc*litresPerKmH;
    }
    public void accelerate(double a) {
        double fuelConsumed = computeConsumedFuel(a, fuelType.getLitresPerKmH());
```

```
        if(fuelConsumed < fuel) {
            speed = speed + amount;
            fuel = fuel - fuelConsumed;
        }
        else {
            double increaseSpeed = fuel / fuelType.getLitresPerKmH();
            speed = speed + increaseSpeed;
            fuel = 0;
        }
    }
    void chargeLoad(double l) {
        if(l>0.0)
            this.loadedCharge += l;
    }
    double unload() {
        double value = this.loadedCharge;
        this.loadedCharge = 0.0;
        return value;
    }
}
```

From the code above it should be pretty clear that we are duplicating quite a lot of code. Conceptually, a car is a bicycle with fuel, and a truck is a car that can load and unload some goods. We might add a further abstraction level, and talk generically about vehicles to identify objects that can move with some speed.

7.1 Aggregation and inheritance

Two main concepts when designing an object-oriented program are aggregation and inheritance. We talk about aggregation when we identify an has-a relationship. For instance, a car has a fuel type, and a fuel tank has a fuel type as well. However, fuel type is a distinct concept from cars, meaning that it exists independently of cars, a-priori a single car might have several fuel types (e.g., hybrid cars) or even none, and different cars might share the same fuel type. The same applies to fuel tanks and types. When aggregating objects, the aggregator might expose some of the properties of the aggregated objects as its own property. For instance, a car might expose how much a liter of the fuel used by the car costs.

```
class Car {  
    public double getFuelCost() {  
        return this.fuelType.getFuelCost();  
    }  
}
```

This is a functionality offered by class `Car`, even if it is obtained by aggregating an instance of `FuelType` and calling one of its methods. Class `Car` exposes only one (out of the many) functionality of the aggregated class.

A rather different concept is instead inheritance. This is applied when we identify an is-a relationship. For instance, a car is a vehicle. In fact, it should expose all the components of a vehicle (i.e., brake and accelerate) plus something else (i.e., refuel). Therefore, the main idea of inheritance is to specify an is-a relationship by inheriting all the functionalities, and extending them with something more. For instance, class `Car` should inherit all the data (speed) and functionalities (brake and accelerate) of `Vehicle`, and add something more (refuel).

7.2 extends

Java supports inheritance through the `extends` keyword. When defining a class, we can specify that it extends (at most) another class. In this way, our class will inherit all the fields and methods of the extended class (called superclass from now on), and we can define more fields and methods in our own class. So let's start by implementing a class `Vehicle`, and then extending it with `Car`, `Bicycle`, and `Truck`.

```
public class Vehicle {  
    private double speed;  
    public void accelerate(double a) {  
        if(a>0)  
            this.speed += a;  
    }  
    public void fullBrake() {  
        this.speed = 0;  
    }  
    public void brake(double amount) {  
        if(amount > speed)  
            this.fullBrake();  
        else speed = speed - amount;  
    }  
}
```

Class `Vehicle` defines a field `speed` to store the current speed of the vehicle, and it

provides methods to accelerate and break.

```
public class Bicycle extends Vehicle {  
    private double frontTire, rearTire;  
    public double frontTirePressure() {  
        return frontTire;  
    }  
    public double rearTirePressure() {  
        return rearTire;  
    }  
}
```

Since class `Bicycle` inherits the data and functionalities of class `Vehicle`, it does not need to define a field to track the speed and the methods to brake and accelerate. Instead, we added two fields and methods to represent the pressure of the two tires. When inspecting the interface of `Bicycle` class through an IDE, we obtain the following list of methods:

```
myBicycle.|  
m frontTirePressure() double  
m accelerate(double a) void  
m brake(double amount) void  
m rearTirePressure() double  
m fullBrake() void
```

```
public class Car extends Vehicle {  
    private double fuel;  
    private FuelType fuelType;  
    public void refuel(FuelTank tank) {  
        fuel = fuel + tank.getAmount();  
        tank.emptyTank();  
    }  
}
```

Similarly, class `Car` adds fields to store the fuel type of the car and the amount of available fuel, and a method to refuel the car.


```
class Truck extends Car {  
    private double loadedCharge;  
    public void chargeLoad(double l) {  
        if(l>0.0)  
            this.loadedCharge += l;  
    }  
    public double unload() {  
        double value = this.loadedCharge;  
        this.loadedCharge = 0.0;  
        return value;  
    }  
}
```

Finally, class `Truck` extends class `Car`. In this way, it inherits the components of both `Car` (fuel) and `Vehicle` (speed). Its interface is composed by the union of `Car`'s and `Vehicle`'s methods.

`myTruck.`

m	<code>chargeLoad(double l)</code>	<code>void</code>
m	<code>accelerate(double a)</code>	<code>void</code>
m	<code>refuel(FuelTank tank)</code>	<code>void</code>
m	<code>unload()</code>	<code>double</code>
m	<code>brake(double amount)</code>	<code>void</code>
m	<code>fullBrake()</code>	<code>void</code>

Generally speaking, when class `A` extends class `B` it inherits all components (where by components we mean both fields and methods) of `B`. Therefore, its interface is the union of fields and methods of class `A` with the ones defined in class `B`.

7.3 Constructors

When we instantiate a class (aka, we create an object) we must call a constructor in order to initialize it. When we extend a class, we build up a hierarchy of classes. So when we initialize a class that extends another one, we need first to initialize the superclass, and then our extension. For instance, let's add a constructor that initializes class `Vehicle` assigning a given value as initial speed.

```
public class Vehicle {  
    private double speed;  
    public Vehicle(double initialSpeed) {  
        this.speed = initialSpeed;  
    }  
}
```

Now Car and Bicycle classes do not compile anymore, since they do not explicitly invoke a constructor of the superclass. Why was this working before? The Java approach in this case is quite similar to the one adopted for “simple” constructors. In particular, if a class does not define a constructor, by default the Java compiler adds a constructor without parameters that essentially does nothing. Indeed, if we have a superclass¹⁰, the default constructor calls the super constructor without parameters if this exists, or it fails to compile otherwise. If we add a constructor with parameters to class Vehicle, the default constructors of Car and Bicycle won’t be able to invoke a super constructor, and therefore the compiler raises an error. Therefore, let’s add constructors to all the classes defined so far in order to compile our code.

```
public class Bicycle extends Vehicle {  
    public Bicycle(int initialSpeed, int frontTire, int rearTire) {  
        super(initialSpeed);  
        this.frontTire = frontTire;  
        this.rearTire = rearTire;  
    }  
}
```

```
public class Car extends Vehicle {  
    public Car(double initialSpeed, FuelType f) {  
        super(initialSpeed);  
        fuelType = f;  
    }  
}
```

```
class Truck extends Car {  
    public Truck(double initialSpeed, FuelType f) {  
        super(initialSpeed, f);  
    }  
}
```

You probably already noticed that all these constructors start with an invocation to `super(...)`. What does this mean? Intuitively, these instructions invoke a constructor of the

¹⁰ This is always the case in Java, but we’ll see this details later when we’ll introduce the Object class

superclass. The keyword `super` allows the class to access and invoke the various components of the superclass. Its use is pretty similar to the keyword `this`, that is, we can use it to invoke methods and access fields, or to invoke a constructor of the superclass.

Generally speaking, the first instruction of all constructors must either invoke a constructor (i) of the same class, or (ii) of the superclass. If we have constructors without parameters, such calls are hidden and added to the bytecode by the Java compiler. For instance, let's imagine that we want to add a constructor to class `Truck` that allows us to set an initial load to our truck. This constructor should first call the constructor of the class `Truck` that receives only an initial speed and the fuel type (thus initializing this data), and then to initialize the field that stores the load of the truck.

```
class Truck extends Car {
    public Truck(double initialSpeed, FuelType f) {
        super(initialSpeed, f);
    }
    public Truck(double initialSpeed, FuelType f, double loadedCharge) {
        this(initialSpeed, f);
        this.loadedCharge = loadedCharge;
    }
}
```

Note that in this way all the constructors call directly or indirectly a super constructor. The Java compiler checks if there is a loop in the chain of constructor calls (e.g., constructor 1 invokes constructor 2, and constructor 2 invokes constructor 1), and it raises an error in this case. In addition, all constructors must invoke a constructor of either the current class or the super class, but they cannot invoke both, as otherwise the principle of invoking always exactly one constructor when we instantiate a class would be broken otherwise.

7.4 protected

	Same class	Same package	Subclasses	Everywhere
<code>public</code>	👍	👍	👍	👍
<code>protected</code>	👍	👍	👍	👎
<code><default></code>	👍	👍	👎	👎

private				
---------	---	---	---	---

Section 4.4 introduced the access modifiers supported by Java. However, we skipped the discussion of one of them, that is, `protected`. This modifier is less restrictive than the default level (that is, it allows the component to be visible from the same class and package) by adding the possibility to see the specific component from the subclasses, but not from everywhere (like the `public` modifier). For instance, we might want to allow subclasses to see and modify the value stored in field `speed` of class `Vehicle`. In this way, class `Car` can decrease the speed (e.g., by 10%) if the car tank is empty by directly reading and writing the value of this field. The implementation of such a method (called `isFuelEmpty()`) might look as follows.

```
public class Vehicle {
    protected double speed;
    public Vehicle(double initialSpeed) {
        this.speed = initialSpeed;
    }
}

public class Car extends Vehicle {
    public boolean isFuelEmpty() {
        if(fuel <= 0) {
            super.speed = super.speed * 0.90;
            return true;
        }
        else return false;
    }
}
```

7.5 Method signature and method definition

Before discussing how we might override an inherited method to redefine its behavior, we need to understand what is the difference between signature and definition of a method. The method signature is composed of all the information we have when we *call* a method. For instance, when we execute the statement `car.accelerate(10.0)` we know that we are invoking a method (i) on class `Car`¹¹, (ii) whose name is `accelerate`, and (iii) with one parameter of type `double`. Therefore, the method signature is composed by (i) the static type of the receiver, (ii) the name of the method, and (iii) the number and static type of parameters.

Instead, when we define a method we specify more information. In particular, method

¹¹ Assuming that the static type of variable `car` is `Car`

accelerate is defined and implemented as follows.

```
public class Vehicle {  
    public void accelerate(double a) {  
        if(a>0)  
            this.speed += a;  
    }  
}
```

In addition to the method signature (that is, (i) the class containing the definition of the method, (ii) its name, and (iii) number and static type of parameters), the definition contains also the return type (void in this case since the method does not return any value), and the visibility (public).

It is important to underline that when we invoke a method we can specify only its signature. Therefore, inside a class we cannot have the definition of two methods with the same signature, since otherwise the runtime environment won't be able to decide what method is invoked. Consider for instance, the following code.

```
public class Vehicle {  
    public void accelerate(double a) {  
        if(a>0)  
            this.speed += a;  
    }  
    public int accelerate(double a) {  
        if(a>0)  
            this.speed += a;  
        return this.speed;  
    }  
}
```

Which method would `car.accelerate(10)` invoke? We cannot decide a priori what the developer wanted to execute! In such cases, the compiler raises an error and it does not compile this code.

7.6 Overriding and overloading

Overloading means that the same class defines several methods with the same name but different signatures. This is possible since when calling the method the runtime environment will be able to decide exactly what method to execute. For instance, consider the two following implementations of method `accelerate`.

```
public class Vehicle {  
    public void accelerate(double a) {  
        if(a>0)  
            this.speed += a;  
    }  
    public void accelerate(String a) {  
        if(a>0)  
            this.speed += Double.valueOf(a);  
    }  
}
```

Since double and String are two distinct types, the signature is different. For instance, if we execute `car.accelerate(10.0)` we know we have to execute the first implementation, while `car.accelerate("1.2")` will execute the second implementation.

Instead we talk about method overriding when a class implements a method with the same signature of a method already implemented in a superclass. First of all, note that, as discussed in the previous section, it is not possible to implement two methods with the same signature in the same class. Instead, it is possible to implement a method with the same signature of a method in the superclasses. In these cases, the implementation in the subclass will override (aka, hide) the implementation in the superclass, redefining its implementation (aka, behavior). For instance, in the first part of this book we spent quite some time discussing how the car consumes fuel when accelerating. However, now class Car inherits the implementation of accelerate from class Vehicle (that does not have any knowledge about fuel), and therefore it does not consume fuel when accelerating! Therefore, we need to redefine the method accelerate in class Car in order to properly implement it. This can be achieved by overriding the method.

```
public class Vehicle {  
    private double speed;  
    public void accelerate(double a) {  
        if(a>0)  
            this.speed += a;  
    }  
}  
public class Car extends Vehicle {  
    public void accelerate(double amount) {  
        double fuelConsumed = amount*fuelType.getLitresPerKmH());  
        if(fuelConsumed < fuel) {  
            super.accelerate(amount);  
            fuel = fuel - fuelConsumed;  
        }  
        else {  
            super.accelerate(fuel / fuelType.getLitresPerKmH());  
            fuel = 0;  
        }  
    }  
}
```

```
}  
}  
}
```

When we invoke method `accelerate` on an instance of `Car`, the implementation in this class will hide the implementation of `Vehicle`, and therefore the execution will consume the fuel as expected. Note that indeed the overriding method calls the overridden method by using the keyword `super`: since this keyword `super` allows to access the components of the superclass, we can also access the components that are hidden in the subclass (aka, the overridden methods).

As we have seen, the definition of a method contains something more than its signature. In particular, for now we will focus on its visibility. What happens if an overriding method declares a different visibility w.r.t. the overridden method? We must always reason in terms of contracts when we think about classes and the interface they expose. When a method overrides another one, it needs to define an interface that covers *at least* the interface of the overridden method. This means it should be applicable at least in all the contexts in which the overridden method was available. When thinking about visibility levels, the overriding method should be at least as much visible as the overridden method. That is, its visibility can be more relaxed. For instance, if the overridden method has protected visibility, then the overriding method can be either protected or public.

7.7 abstract

In Java, the keyword `abstract` allows one to define a method without providing its implementation. However, if a class contains abstract methods cannot be instantiated, as otherwise the runtime environment would not know what code should be executed when the abstract method is invoked in the object instance of the class. Therefore, if a class contains at least one abstract method, then the class itself must be abstract. However, one can also define a class as abstract even if it does not have any abstract method. In any case, abstract classes cannot be instantiated, but they define constructors in order to properly initialize their own state. Classes extending abstract classes can either (i) override all the abstract methods (then they can be instantiated and they do not need to be abstract), or (ii) be abstract.

For instance, let's imagine that we want to define `accelerate` in `Vehicle` as abstract, since we do not know a priori how a vehicle accelerates (e.g., a car would consume fuel). Therefore, the class `Vehicle` must be declared as abstract, while classes `Car` and `Bicycle`

will need to implement and override method `accelerate`. In such a scenario, their implementation might look as follows.

```
public abstract class Vehicle {
    protected double speed;
    public abstract void accelerate(double a);
}

public class Car extends Vehicle {
    public void accelerate(double amount) {
        double fuelConsumed = amount*fuelType.getLitresPerKmH();
        if(fuelConsumed < fuel) {
            this.speed += amount;
            fuel = fuel - fuelConsumed;
        }
        else {
            this.speed += fuel / fuelType.getLitresPerKmH();
            fuel = 0;
        }
    }
}

public class Bicycle extends Vehicle {
    public void accelerate(double a) {
        if(a>0)
            this.speed += a;
    }
}
```

In addition, note that we do not need to modify class `Truck`, since this class already inherits from class `Car` the implementation of `final`.

7.8 final

With overriding methods, a subclass can arbitrarily redefine the behavior of the method. There are however cases in which we want to prevent this: a car should always consume fuel as we defined when accelerating, and therefore we want to forbid the overriding of `Car.accelerate`. Java supports this by defining methods as `final`.

```
public class Car extends Vehicle {
    public final void accelerate(double amount) {...}
}
```

In this way, class `Truck` cannot override method `accelerate`.

The keyword `final` can be also applied to classes. In this case, the whole class cannot be extended. This is usually adopted when our class contains the ultimate implementation of the objects we want to represent, and nobody should be allowed to customize (that is,

add or modify its functionalities) it. For instance, we might think that class `Bicycle` contains already the whole implementation for bicycles, and therefore we set it as `final`.

```
public final class Bicycle extends Vehicle {...}
```

Note that the semantics of `final` when applied to methods and classes is rather different from its semantics on fields. In particular, in the first case `final` reduces the inheritance and overriding of the various components. Instead, in the second case `final` simply states that a field can be assigned only once during the construction of an object, and in no other place. We think that the keyword `final` in Java is rather misleading and confusing since it mixes together concepts that are very different (constant values, and methods and classes that cannot be overridden or extended).

7.9 Applicability and combination of modifiers

We already covered the main modifiers that we have in Java both about visibility and other semantic features. However, not all modifiers can be applied to any component in any context, and only some of their combinations are allowed.










	Class ¹²	Method	Field
<code>public</code>	👍	👍	👍
<code>protected</code>	👎	👍	👍
<code><default></code>	👍	👍	👍
<code>private</code>	👎	👍	👍

First of all, we identified three main components in an OO program: classes, methods, and fields. Accessibility modifiers (`public`, `protected`, `default/package`, and `private`) can be freely applied to any method and field. However, some restrictions apply to classes. In particular, a class cannot be `private`, since otherwise it would be deadcode: if a class were `private`, it would be visible only from the class itself, and it would not be accessible from any other class (aka, piece of code of our program)!

Another restriction imposes that a class cannot be `protected`. To understand why, we have to focus on the gap between the `default/package` visibility (that is, a component is visible only from the classes in the same package) and the `protected` one (package plus subclasses). A class can declare to be a subclass of another one only if the latter is visible from the former. A `protected` class would mean that a class `C1` is visible from a subclass

¹² In this book we do not introduce and discuss nested classes

C2, but C1 should be visible to C2 in order to allow C2 extending C1! Here we are facing a logical loop about the visibility level on subclasses, and therefore the protected modifier is forbidden on classes by the Java compiler.

	Class ¹³	Method	Field
static			
final			
abstract			

Let's move on with the other modifiers we introduced: static, final, and abstract. We have already discussed the fact that final can be applied to classes, methods, and fields, but with a rather different semantic: on classes and methods it restricts inheritance mechanisms, while on fields it obliges to initialize the value of the field exactly once in the constructors.

Instead, abstract is not allowed on fields. The idea of abstract is to define a signature, but not fully the component. For instance, an abstract method provides a full definition of a method but not its implementation. However, the signature of a field is composed by its name and static type, but we do not need to define any additional component (such as an implementation). Therefore, it is not possible to decouple the definition and implementation of a field, and the abstract modifier is not allowed on those components. Finally, we cannot declare a class as static. The static modifier specifies that something is not attached to an instance, but to the class itself. A static class would mean that the whole class (aka, fields and methods) belongs to the class itself and there is no state of instances of the object. This is nonsense for an OO program, and therefore it is not allowed.

Finally, we might think about combining static, final and abstract together. However, abstract and static are not allowed together since with abstract we would mean that the component is implemented in the subclasses, and with static that the component belongs to the class itself (and thus cannot be overridden by the subclasses). The same happens with final and abstract: abstract means that we do not fully define the component and we leave the duty of defining them to the subclasses, while final specifies that the component cannot be redefined by the subclasses. Clearly the semantics of the two modifiers are in contrast. Finally, the Java compiler¹⁴ allows to specify that an abstract

¹³ As before, please remind that we do not introduce and discuss nested classes

¹⁴ In this book, we always refer to the Java compiler applying the standard options in terms of warnings, errors, etc..

method is static. However, a warning is raised: since a static component belongs to the class itself and cannot be overridden, the final modifier is superfluous.

7.10 Exercise

Given the following classes `CarID` and `MotorcycleID`, refactor the code according to the principles of aggregation and inheritance of Section 7.

```
package it.unive.dais.pol.id;

public class CarID {

    private final String make;
    private final String model;
    private String licensePlate;
    private final String VIN;

    public CarID(String make, String model, String licensePlate, String VIN){
        this.make = make;
        this.model = model;
        this.licensePlate = licensePlate;
        this.VIN = VIN;
    }

    public String getMake() { return make; }

    public String getModel() { return model; }

    public String getLicensePlate () { return licensePlate; }

    public void setLicensePlate(String licensePlate) {
        this.licensePlate = licensePlate;
    }

    public String getVIN() { return VIN; }

    public boolean isSameLicensePlate(String licensePlate) {
        return this.licensePlate.equals(licensePlate);
    }

    public boolean isValidLicensePlate() {
        String licensePlate = car.getCarID().getLicensePlate();

        if(licensePlate != null && licensePlate.length() != 7)
            return false;

        if(!Character.isDigit(licensePlate.charAt(2))
            || !Character.isDigit(licensePlate.charAt(3))
            || !Character.isDigit(licensePlate.charAt(4)))
            return false;

        if(!Character.isUpperCase(0))
            || !Character.isUpperCase(licensePlate.charAt(1))
            || !Character.isUpperCase(licensePlate.charAt(5))
            || !Character.isUpperCase(licensePlate.charAt(6)))
```

```
        return false;

    if(!licensePlate.contains("I")
        && !licensePlate.contains("O")
        && !licensePlate.contains("EE"))
        return false;

    return true;
}
}
```

```
package it.unive.dais.pol.id;

public class MotorcycleID {

    private final String make;
    private final String model;
    private String licensePlate;
    private final String VIN;

    public CarID(String make, String model, String licensePlate, String VIN){
        this.make = make;
        this.model = model;
        this.licensePlate = licensePlate;
        this.VIN = VIN;
    }

    public String getMake() { return make; }

    public String getModel() { return model; }

    public String getLicensePlate () { return licensePlate; }

    public void setLicensePlate(String licensePlate) {
        this.licensePlate = licensePlate;
    }

    public String getVIN() { return VIN; }

    public boolean isSameLicensePlate(String licensePlate) {
        return this.licensePlate.equals(licensePlate);
    }

    public boolean isValidLicensePlate() {
        String licensePlate = car.getCarID().getLicensePlate();

        if(licensePlate != null && licensePlate.length() != 7)
            return false;

        if(!Character.isDigit(licensePlate.charAt(2))
            || !Character.isDigit(licensePlate.charAt(3))
            || !Character.isDigit(licensePlate.charAt(4))
            || !Character.isUpperCase(licensePlate.charAt(5))
            || !Character.isUpperCase(licensePlate.charAt(6)))
            return false;

        if(!Character.isUpperCase(0))
```

```
        || ! Character.isUpperCase(licensePlate.charAt(1)))  
        return false;  
  
        if(!licensePlate.contains("I")  
            && !licensePlate.contains("O")  
            && !licensePlate.contains("EE"))  
            return false;  
  
        return true;  
    }  
}
```

Chapter 8: Subtyping

We have seen that when we extend a class, the subclass provides an interface that is larger than the superclass. This means that it provides the same components of the superclasses (maybe redefining the behavior of some of them), and it potentially adds some more components. This leads to a contract in the subclass that implies the contract in the superclass.

Now let's go back to the example with vehicles. We want to implement a method to run a race among different vehicles. Our implementation must be agnostic on the specific type of vehicle that is moving. This means that we want to provide an implementation that can run the race with any type of vehicle just relying on the interface of vehicles. We therefore implement the following class.

```
public class Race {  
    /**  
     *  
     * @param v1 the first vehicle  
     * @param v2 the second vehicle  
     * @param length the length of the race  
     * @return the id of the winner of the race, or -1  
     */  
    public static int race(Vehicle v1, Vehicle v2, double length) {  
        v1.fullBrake();  
        v2.fullBrake();  
        double distancev1 = 0, distancev2 = 0;  
        while(distancev1 < length && distancev2 < length) {  
            v1.accelerate(Math.random()*10.0);  
            v2.accelerate(Math.random()*10.0);  
            distancev1 += v1.getSpeed();  
            distancev2 += v2.getSpeed();  
        }  
        if(distancev1 >= length) {  
            if(distancev2 >= length)  
                return -1;  
            else return 1;  
        }  
        else return 2;  
    }  
}
```

Now the next question is: how can we use this? We can freely create different types of

vehicles, and pass them to the race method.

```
Car myCar = new Car(0, new FuelType("diesel", 1.4, 0.01));  
Bicycle myBicycle = new Bicycle(10);  
Truck myTruck = new Truck(0, diesel);  
myCar.refuel(new FuelTank(diesel, 2));  
myTruck.refuel(new FuelTank(diesel, 3));  
Race.race(myBicycle, myCar, 100);  
Race.race(myBicycle, myTruck, 100);
```

How is it possible that this works out of the box? Indeed, when we extend a class, the subclass is a subtype of the superclass, and it can substitute the superclass. This is what we'll explain in this chapter.

8.1 The substitution principle

The substitution principle simply states that if a class C1 exposes an interface that is wider than C2, then we can have instances of C1 wherever an instance of C2 is expected. What do we mean by wider interface? That the interface of C1 defines all the fields and methods of C2, plus something else.

For instance, in the example above each time we have a Vehicle we might receive a Car, Bicycle and Truck, since all these classes define the methods of Vehicle (to accelerate and brake) plus something else (management of the fuel, tire pressure, load and unload some goods). This is exactly what happened with Race.race: we were passing bicycles, cars and trucks where vehicles were expected.

In Java, when a class C1 extends a class C2 it contains all the components of the interface of C2, and it can add something more. Therefore, in Java we can substitute a class with any of its (direct or indirect) extensions. To understand how this is supported, we need to first discuss the type system adopted by Java.

8.2 The Java type system

Java programs need to declare the types of fields, parameters, local variables and returned values¹⁵. For instance, when we define the method Race.race, we specify that the first and second parameters must be Vehicles, the third parameter must be a double value, and the returned value is an integer. We refer to declared types as static types, since these are known at compile time (statically) without the need to run (dynamically)

¹⁵ In this part we ignore the var keyword introduced in Java 10

the program).

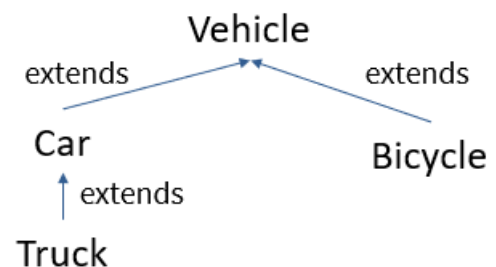
Starting from those types, the Java compiler infers the static type of all expressions of a program. For instance, `distanceV1 < length` is an expression with static type `boolean`, as it is the result of a comparison (less than) of two numerical values (`distanceV1` and `length`).

Finally, Java is strongly typed. This means that thanks to its type system we know that, if a program compiles correctly (and in particular the type check succeeds) then each time we call a function (or we execute other operations such as arithmetic expressions) we know that the types (of receiver and parameters) are correct. This means that if the OO program compiles, then the method we call exists. A consequence of this is that each area of allocated memory has a (static) type.

Note that all these are choices of Java, and other OO programming languages might adopt other approaches. A programming language might not have declared types (and this is always the case for dynamic programming languages such as JavaScript and Python, that check the correctness of types only during the execution of a program), or apply a weakly typed approach (and thus one might arbitrarily convert from a type to another one - e.g., we can assign a pointer to an integer and vice versa in C by explicitly casting the values).

8.3 The subtype relation

In Java inheritance defines a subtype relation among classes. In particular, a class `C1` is subtype of `C2` if `C1` extends (directly or indirectly) `C2`. For instance, `Truck` is subtype of `Car` and `Vehicle`, and `Car` and `Bicycle` are subtypes of `Vehicle`.



What does it mean for `C1` to be subtype of `C2`? Briefly, it means that wherever we need an instance of `C2`, we can pass an instance of `C1`. For instance, in the race between vehicles we were able to pass cars, trucks and bicycles, since all these classes were subtypes of `Vehicle`. This approach preserves the substitution principle: a subclass has an interface that is the same or wider than the superclass, and then we can replace the superclass with the subclass.

8.4 Polymorphism

Polymorphism is defined as “the quality or state of existing in or assuming different

forms”¹⁶. Such a concept is applied to several different areas, such as biology (where it means that there are several forms between the individuals of the same species) and genetics (where it means that one species contains individuals with varying chromosome counts or shapes).

In computer science, polymorphism means that the same symbol might represent different implementations based on a specific execution of a program. This is exactly what we obtained by overriding method `accelerate` in class `Car`: if we have a car (or any subclass such as `truck`) and we `accelerate`, we consume fuel, if we have a bicycle we simply `accelerate`.

The subtype relation enables polymorphism. In fact, when our code invokes a method, the static type checker validates that the static type of the expression used as receiver of the invocation contains the invoked method. However, at runtime our code might receive a subclass of the static type we declared, and such subclass might override the method we are invoking, redefining in this way its implementation and behavior. This is the case of method `race` presented at the beginning of this chapter: our code invoke method `accelerate` over a `Vehicle` parameter, but when an instance of class `Car` is passed, we execute the implementation of `accelerate` in `Car` and not in `Vehicle`.

8.5 Static and dynamic type

At this point, it should be clear that for each expression in our program we have two distinct types: (i) a static type that is declared (e.g., on fields, parameters and local variables) or inferred by the compiler, and (ii) a dynamic type that is the concrete type of the expression when we execute the program. Note that the static type is known at compile time and it does not vary based on the execution, while the dynamic type is known only during the execution and it can vary from one execution to another one. In addition, a method, like `Race.race`, might be invoked once with some parameter dynamic types, one with others.

The dynamic type is always the same type or a subtype of the static type. This is ensured by the Java compiler that refuses to compile any program that does not satisfy such property. Thanks to the enforcement of this property, we know that each time we access an object component this component exists.

8.6 Type checks and casts

¹⁶ <https://www.merriam-webster.com/dictionary/polymorphism>

However, one might want to (i) check the dynamic type of an expression, and (ii) cast a variable to some given types. Java supports this through

- `<expression> instanceof <type>` returns a boolean value, that is true if and only if the dynamic type of the expression is compatible (that is, it is equal or subtype of) with the given type, and
- `(<type>) <expression>` casts the static type of the given expression to the given type. If the dynamic type of the given expression is not compatible with the given type, a `ClassCastException` is thrown raising an execution error.

For instance, let's imagine that we want to refuel a Car before starting the race. However, method `race` receives two vehicles, that can be cars or not. This functionality can be implemented as follows.

```
public static int race(Vehicle v1, Vehicle v2, double length) {  
    v1.fullBrake();  
    v2.fullBrake();  
    if(v1 instanceof Car) {  
        Car c = (Car) v1;  
        c.refuel(new FuelTank(...));  
    }  
    //same code for v2  
    ...  
}
```

This piece of code first checks if the vehicle is indeed a car, and in this case it casts the local variable to `Car`, and then it refuels the car. Note that this works also if `v1` is `Truck`, since this is a subtype of `Car`.

The compiler knows the static type of each expression, and we know that the dynamic type must be equal to or a subtype of this type. Therefore, the compiler produces some errors and warnings if the type checks and casts apply wrong types. In particular, if we cast to an incompatible type (that is, a type that is not a subtype of the static type of the expression) then a compiler error is raised and the code does not compile. For instance, if we cast to `Car` an expression whose static type is a `Bicycle`, we already know that this will raise an exception, and therefore the Java compiler prevents its compilation through the error. The same happens if we check an incompatible type through `instanceof`.

Instead, we might type check and cast to a supertype of the static type. For instance, we might check if a `Car` is a `Vehicle`, or we might cast a `Car` to a `Vehicle`. In the first case, we know that the `instanceof` statement will always return true, while in the second case is not needed since we can already access all the components of a supertype from the

subtype, or assign a subtype to a supertype without the need of casting it. In these cases, the compiler raises a warning (since the operations are useless) but it allows to compile and execute this code.

Chapter 9: Interfaces

Up to now the only way we have to define that a class is a subtype of something else is by extending another class. This generates a subtype relation that can be represented as a tree (as we have done in the previous chapter). However, this might be quite restrictive, since we might want to build up a more complex hierarchy. For instance, in our examples with vehicles we have identified cars (that have fuel) and trucks (that have fuel and some load). However, we might want to represent vehicles that have load but not fuel, like a cart pulled by some animals. How could we proceed to support such a scenario?

9.1 Single and multiple inheritance

Java supports single inheritance. In practice, this means that after the `extends` keyword we can specify only one class. In the example above, we might think about writing a class that represents a loadable entity, that is, a class on which we can load and unload some goods. The implementation might look as follows.

```
class Loadable {  
    private double loadedCharge = 0.0;  
    public void chargeLoad(double l) {  
        if(l>0.0)  
            this.loadedCharge += l;  
    }  
    public double unload() {  
        double value = this.loadedCharge;  
        this.loadedCharge = 0.0;  
        return value;  
    }  
}
```

You might notice that this implementation is exactly the same we had in `Truck`. At this point, we might think to define class `Truck` as an extension of both `Car` and `Loadable`, and `HorseCart` as extension of `Vehicle` and `Loadable`. This would be possible if we had multiple inheritance (as in C++), but it is not possible in Java. So why Java does not support multiple inheritance?

The main reason is that single inheritance allows a much easier reasoning on the program than multiple inheritance. In particular, multiple inheritance implies that we

have multiple subclasses. Therefore, the Java keyword `super` might point to one superclass or the other, and when we invoke a method in a class that inherits it we would need to know if the called method stays in one class or in the other. In addition, what happens if the same method is implemented in both superclasses? Different programming languages provide different solutions (C++ requires to explicitly state what method to execute, while Scala follows the order of inheritance). Java decided to take a shortcut and support only single inheritance (like many other OO programming languages like C#). However, it still provides a mechanism to declare that a class is a subtype of several other types.

9.2 Interfaces

An interface defines a list of public method signatures without providing an implementation of the method. Like classes, it is declared in a Java file, it belongs to a package, and it defines a type we can use in our program. For instance, we can define an interface called `Loadable` that contains the methods that allow us to load and unload some goods from a truck or a horse chart.

```
package it.unive.dais.pol.vehicles;  
public interface Loadable {  
    public void chargeLoad(double amount);  
    public double unload();  
}
```

Classes can then implement interfaces. In particular, a class can declare the list of implemented interfaces using the keyword “implements”. A list of interfaces follow this keyword. Note that this approach is different from `extends`, where only at most one class can follow this keyword. In fact, Java supports single inheritance, but it allows a class to implement several interfaces. When a class implements an interface, it needs either to implement either (i) all the methods defined in the interface, or (ii) few or none of the methods, but then it must be declared as abstract. The class is then a subtype of all the implemented interfaces. Let's go back to our `Loadable` example. We now (i) make `Truck` implements `Loadable`, and (ii) implement a class `HorseChart` that represents a chart trained by horses (that is, a loadable vehicle without fuel).

```
public class Truck extends Car implements Loadable {  
    private double loadedCharge;  
    public void chargeLoad(double l) {  
        if(l>0.0)  
            this.loadedCharge += l;  
    }  
    public double unload() {
```

```
        double value = this.loadedCharge;
        this.loadedCharge = 0.0;
        return value;
    }
}

public class HorseChart implements Loadable {
    private double loadedCharge;
    public void chargeLoad(double l) {
        if(l>0.0)
            this.loadedCharge += l;
    }
    public double unload() {
        double value = this.loadedCharge;
        this.loadedCharge = 0.0;
        return value;
    }
}
```

We can now use Loadable as a type of our program like we did with Vehicle. In particular, we can add to our method Race.race a check on Loadable objects and unload the goods before starting the race as follows.

```
public static int race(Vehicle v1, Vehicle v2, double length) {
    ...
    if(v1 instanceof Loadable) {
        Loadable l = (Loadable) v1;
        l.unload();
    }
    //same code for v2
    ...
}
```

This code is almost identical to the one that refuelled the cars in the previous chapter. However, at this point of the book I hope that it is obvious that the implementations of Truck and HorseChart are not acceptable, since they completely duplicate the code that manages the load. Unfortunately, this is the only solution we have with single inheritance¹⁷.

9.3 Default implementations in interfaces

Java 8 added support for implementing (a part of) interfaces. In particular,

¹⁷ Indeed, we might define a Vehicle as an interface and Loadable as a class. In this way, we would not duplicate the code of Loadable, but we would need to duplicate the code of Vehicle, and this is even worse!

implementations of methods in interfaces are called default implementations. One can define a method in an interface with the keyword `default`, and he/she then provides the implementation of the method. In addition, one can add fields to the interface. Therefore, one might think to implement a `Loadable` interface as follows.

```
interface Loadable {  
    private double loadedCharge = 0.0;  
    default void chargeLoad(double l) {  
        if(l>0.0)  
            this.loadedCharge += l;  
    }  
    default double unload() {  
        double value = this.loadedCharge;  
        this.loadedCharge = 0.0;  
        return value;  
    }  
}
```

This solution does not work for several reasons and it raises the following errors:

- modifier `private` is not allowed on field `loadedCharge`. In fact, all components declared in an interface are public.
- cannot assign a value to final variable `loadedCharge`. This might sound surprising, since `loadedCharge` looks like a normal field and not a final variable. Indeed, fields declared in interfaces are all static and final, and therefore they cannot be assigned.

However, we can try a different solution: define a getter and setter method for `loadedCharge`, and then rely on these methods in `chargeLoad` and `unload`. The implementation will be as follows.

```
public interface Loadable {  
    double getLoad();  
    void setLoad(double l);  
    default public void chargeLoad(double amount) {  
        if(amount>0.0)  
            this.setLoad(this.getLoad()+ amount);  
    }  
    default public double unload() {  
        double value = this.getLoad();  
        this.setLoad(0.0);  
        return value;  
    }  
}
```

Then we need to implement these getter and setter in `Truck` and `HorseCart` as follows.

```
public class HorseCart extends Vehicle implements Loadable {  
    private double loadedCharge;  
    public double getLoad() {  
        return loadedCharge;  
    }  
    public void setLoad(double l) {  
        this.loadedCharge=l;  
    }  
}  
  
public class Truck extends Car implements Loadable {  
    private double loadedCharge;  
    public double getLoad() {  
        return loadedCharge;  
    }  
    public void setLoad(double l) {  
        this.loadedCharge=l;  
    }  
}
```

If on the one hand this solution still duplicates some code (field `loadedCharge`, getter, and setter), it looks more reasonable than the initial one where the whole logic of `chargeLoad` and `unload` was duplicated. However, this code is still far from being ideal: the getter and setter are public methods, and therefore external code might read and write this field. This does not encapsulate the code well, since it exposes unneeded methods. Unfortunately, this is the best solution we have for now in Java, that is, with single inheritance and default implementations of interfaces. And this is also why we believe default implementations are the wrong solution to the right problem.

9.4 Implementing several interfaces

The main difference between extending classes and implementing interfaces is that we can extend at most one class, but we can implement many interfaces. In this way, we can build up a rather complex subtype relationship, where a single class is a subtype of many other types defined by interfaces. The class implementing interfaces must either implement all the methods defined in all the interfaces, or be abstract. Let's for instance imagine that we want to define a `Printable` interface, that defines only a `print()` method. Such an interface might be implemented by several classes at different points of our type hierarchy, as, for instance, `Truck` and `Bicycle`. The implementation would look as follows.

```
public interface Printable {  
    public void print();  
}  
  
public class Truck extends Car implements Loadable, Printable {
```

```
public void print() {  
    System.out.print("I am a truck with "+loadedCharge+" kgs of goods");  
}  
}  
public class Bicycle extends Vehicle implements Printable {  
    public void print() {  
        System.out.println("I'm a slow bicycle");  
    }  
}
```

We can see here that class Truck implements two interfaces (Loadable and Printable) while Bicycle implements only Printable. At this point, we might print all the Printable objects in an array of vehicles as follows.

```
void printAll(Vehicle[] vehicles) {  
    for(int i=0; i<vehicles.length; i++)  
        if(vehicles[i] instanceof Printable)  
            ((Printable) vehicles[i]).print();  
}
```

In this way, we will print a message for each truck or bicycle in the array.

9.5 Extending interfaces

We can also build up a hierarchy among interfaces. In particular, an interface can extend an interface. In this way, it will inherit all the methods defined in the superinterface, and it can add more methods (or provide default implementations). This is achieved by using the `extends` keyword in the declaration of the interface. However, please note that while classes can extend at most one other class, interfaces can extend multiple interfaces. We now decouple the Loadable interface in two steps: a Loadable interface defining the `loadCharge` method with its default implementation, an Unloadable interface defining the `unload` method with its default implementation, and a LoadableUnloadable interface extending both interfaces.

```
public interface Loadable {  
    double getLoad();  
    void setLoad(double l);  
    default public void chargeLoad(double amount) {  
        if(amount>0.0)  
            this.setLoad(this.getLoad()+ amount);  
    }  
}  
public interface Unloadable {  
    double getLoad();  
    void setLoad(double l);  
}
```

```
default public double unload() {  
    double value = this.getLoad();  
    this.setLoad(0.0);  
    return value;  
}  
}  
  
public interface LoadableUnloadable extends Loadable, Unloadable {}
```

But what happens when we have two interfaces that implement a default method with the same signature? How does Java decide which one of the two implementations should be executed? The answer is simple: Java prevents such a situation by raising a compiler error. For instance, let's imagine that we have a `Loadable2` interface with exactly the same body of `Loadable`. If we declare a class that implements both `Loadable` and `Loadable2`, the compiler will raise an error saying that the class “inherits unrelated defaults for `chargeLoad(double)` from types `Loadable` and `Loadable2`”.

9.6 Abstract classes or interfaces? The hard choice

In this chapter, we deeply studied what we can do nowadays with Java interfaces. We have seen that in an interface we can declare methods and provide default implementations, while we cannot define fields. Instead, abstract classes can define fields and provide only the definition but not the implementation of abstract methods. On the other hand classes and interfaces can implement and extend, respectively, several interfaces, while a class can extend only at most another class. Now the hard question is: when should we use abstract classes, and when instead interfaces?

There is no general or fixed rule that can be applied to any context. The experience will show what are the benefits of one choice or the other. However, generally speaking, if what we are defining describes a property or some aspects of the class, then this should be an interface. If instead it represents the main entity of the class, then this should be a class in order to define not only the functionalities but also the state of the objects.

9.7 Exercise

Given the following classes, refactor the code according to the principles of aggregation and inheritance, exploiting the notions of abstract classes and interfaces of Section 8 and 9.

```
package it.unive.dais.pol.id;

public class CarID {

    private final String make;
    private final String model;
    private String licensePlate;
    private final String VIN;

    public CarID(String make, String model, String licensePlate, String VIN){
        this.make = make;
        this.model = model;
        this.licensePlate = licensePlate;
        this.VIN = VIN;
    }

    public String getMake() { return make; }

    public String getModel() { return model; }

    public String getLicensePlate () { return licensePlate; }

    public void setLicensePlate(String licensePlate) {
        this.licensePlate = licensePlate;
    }

    public String getVIN() { return VIN; }

    public boolean isSameLicensePlate(String licensePlate) {
        return this.licensePlate.equals(licensePlate);
    }

    public boolean isValidLicensePlate() {
        String licensePlate = car.getCarID().getLicensePlate();

        if(licensePlate != null && licensePlate.length() != 7)
            return false;

        if(!Character.isDigit(licensePlate.charAt(2))
            || !Character.isDigit(licensePlate.charAt(3))
            || !Character.isDigit(licensePlate.charAt(4)))
            return false;

        if(!Character.isUpperCase(0))
            || !Character.isUpperCase(licensePlate.charAt(1))
            || !Character.isUpperCase(licensePlate.charAt(5))
            || !Character.isUpperCase(licensePlate.charAt(6)))
```

```
        return false;

    if(!licensePlate.contains("I")
        && !licensePlate.contains("O")
        && !licensePlate.contains("EE"))
        return false;

    return true;
}
}
```

```
package it.unive.dais.pol.id;

public class BikeID {

    private final String make;
    private final String model;
    private final String chassisNumber;

    public BikeID(String make, String model, String chassisNumber){
        this.make = make;
        this.model = model;
        this.chassisNumber = chassisNumber;
    }

    public String getMake() { return make; }

    public String getModel() { return model; }

    public String getChassisNumber() { return chassisNumber; }
}
```

```
package it.unive.dais.pol.id;

public class MotorcycleID {

    private final String make;
    private final String model;
    private String licensePlate;
    private final String VIN;

    public MotorcycleID (String make, String model, String licensePlate, String VIN){
        this.make = make;
        this.model = model;
        this.licensePlate = licensePlate;
        this.VIN = VIN;
    }

    public String getMake() { return make; }

    public String getModel() { return model; }

    public String getLicensePlate () { return licensePlate; }

    public void setLicensePlate(String licensePlate) {
        this.licensePlate = licensePlate;
    }

    public String getVIN() { return VIN; }

    public boolean isSameLicensePlate(String licensePlate) {
        return this.licensePlate.equals(licensePlate);
    }

    public boolean isValidLicensePlate() {
        String licensePlate = car.getCarID().getLicensePlate();

        if(licensePlate != null && licensePlate.length() != 7)
            return false;

        if(!Character.isDigit(licensePlate.charAt(2))
            || !Character.isDigit(licensePlate.charAt(3))
            || !Character.isDigit(licensePlate.charAt(4))
            || !Character.isUpperCase(licensePlate.charAt(5))
            || !Character.isUpperCase(licensePlate.charAt(6)))
            return false;

        if(!Character.isUpperCase(0))
```

```
        || ! Character.isUpperCase(licensePlate.charAt(1)))  
        return false;  
  
        if(!licensePlate.contains("I")  
            && !licensePlate.contains("O")  
            && !licensePlate.contains("EE"))  
            return false;  
  
        return true;  
    }  
}
```

```
package it.unive.dais.pol.id;  
  
public class EBikeID {  
  
    private final String make;  
    private final String model;  
    private final String chassisNumber;  
    private String batteryNumber;  
  
    public EBikeID (String make, String model, String chassisNumber,  
                    String batteryNumber){  
        this.make = make;  
        this.model = model;  
        this.chassisNumber = chassisNumber;  
        this.batteryNumber = batteryNumber;  
    }  
  
    public String getMake() { return make; }  
  
    public String getModel() { return model; }  
  
    public String getChassisNumber() { return chassisNumber; }  
  
    public String getBatteryNumber () { return batteryNumber; }  
}
```


Chapter 10: Method dispatching

So far we have seen how we can override an inherited method, that is, how we can redefine the behavior of a method in a subclass. We also deeply discussed how this enables polymorphism, that is, the fact that the same symbol (a method call in this case) can have different behaviors (that is, leads to the execution of different code). However, we did not yet study how we decide what implementation of the method is executed, that is, how the method call is dispatched. This is exactly the goal of this chapter.

10.1 Static dispatching

When we invoke a method, we have an expression like `<receiver>.<method_name>(<parameters>)`. The code we want to execute is attached to the class of the receiver, and therefore we need to decide from which class we want to start the research of the implementation. Obviously, if that class does not contain the implementation, we will then look into the superclass, and so on.

We have seen that each expression has a static and a dynamic type, where the static type is known at compile time, while the dynamic type is the concrete type an expression has during the execution of the code. The receiver is nothing else than an expression, even if for the most part of the examples we encountered so far, it was a local variable, parameter or field, with a declared static type. Anyway, the receiver has both a static and dynamic type. If we apply static dispatching, we start by looking for the method implementation from the static type of the receiver. For instance, let's consider again the implementation of `Race.race`. When we invoke `v1.accelerate(...)`, we know that `v1`'s static type is `Vehicle`. Therefore, we know that we will always execute the implementation of `accelerate` in class `Vehicle` (that is, the implementation that does not consume fuel) despite the fact that the vehicle passed to the method is a car, truck, bicycle, or horse chart.

Unfortunately, in this way we do not support polymorphism (and I hope you already noticed it!): the static type is known at compile time, and it is the same for all the possible executions of the program. Therefore, we do not have a symbol that has different

behaviors, since its behavior is known once and for all¹⁸. On the other hand, this solution would be very efficient: we do not need at runtime to search for the method, since this is statically bounded.

In addition, please note that such an approach does not work with abstract methods, since we would have invocation to methods that are not implemented. The only way to be sure the method is implemented is that the class is instantiable. So if we would start the search of the implementation from the dynamic type, we would be sure that such a class is instantiable since it has been instantiated to create the object associated with the receiver of the call.

10.2 Dynamic dispatching and polymorphism

Therefore, in order to have a support for polymorphism we need to apply dynamic dispatching. When dealing with the receiver, this means that we start the search for the code to be executed from its dynamic type. Going back to the race example, this means that when we invoke the method passing a car as the first parameter, we execute the implementation of `accelerate` in class `Car`. When instead we pass a truck, we first look into class `Truck`, we do not find an implementation, we search into the superclass `Car`, and there we execute its implementation of method `accelerate`. Finally, if we pass a bicycle, we first look into class `Bicycle`, and then we move to class `Vehicle` executing its implementation of `accelerate`.

Thanks to this approach, the same symbol (aka, method call, for instance `v1.accelerate(...)`) has different behaviors, thus supporting polymorphism. Even more, such behaviors might be completely unknown at compile time, as we might receive instances of subclasses of the static type of the receiver that are not part of the program we compiled and shipped together.

10.3 Method dispatching in the presence of overloading

The subtype relation we discussed in the previous chapter might increase the complexity in the presence of overloading. In fact, we might have overloaded methods where the parameters' types are one subtype of the other, and it might not be clear what implementation of the method should be executed. For instance, let's imagine that we

¹⁸ Note that the implementation might not be present in the static type and in this case it should be searched in the superclass, but anyway it is statically known.

overload the method `Race.race` to consider the case when the two parameters are both cars, and when the first parameter is a car and the second is a vehicle. In such a scenario, we would have the following methods:

```
public class Race {  
    public static int race(Vehicle v1, Vehicle v2, double length) {...}  
    public static int race(Car v1, Car v2, double length) {...}  
    public static int race(Car v1, Vehicle v2, double length) {...}  
}
```

We have now the following invocations.

```
Car c1 = new Car(...), c2 = new Car(...);  
Bicycle b3 = new Bicycle(...);  
Vehicle v1 = c1, v2 = c2, v3 = b3;  
Race.race(c1, c2);  
Race.race(c1, v3);  
Race.race(v1, b3);  
Race.race(v1, v2);
```

The first invocation is pretty simple: the static and dynamic types of the two parameters are `Car`, therefore we invoke `race(Car, Car)`. The same applied to the second case: we have a car, and then a bicycle pointed by a vehicle variable. The method executed will be `race(Car, Vehicle)`. But what about the last two cases? If we apply dynamic dispatching of parameters, we would invoke `race(Car, Vehicle)` and `race(Car, Car)`, respectively. If instead we apply static dispatching, we will execute `race(Vehicle, Vehicle)` in both cases. The last case compared with the first one looks very odd: we are passing exactly the same runtime value to `race`, but in one case we execute on implementation, in another we choose a different implementation!

10.4 The Java solution

Java adopted dynamic dispatching on the receiver, and static dispatching on the parameters. Therefore, in the case above the third and forth invocation of method `race` will both execute `race(Vehicle, Vehicle)`. Such a choice might seem rather odd for the reasons we discussed above.

Therefore one might question: why such a difference? As already discussed, dynamic dispatching of the receiver is needed to have polymorphism, and in an object-oriented programming language this is not a choice, but it is mandatory. Instead, the parameters might be statically or dynamically dispatched. The runtime overhead of static dispatching is minimal, while dynamic dispatching might occur in significant overhead. On the other hand, dynamic dispatching would allow one to choose the “nearest”

implementation w.r.t. the effective types of values passed as parameters. Therefore, there are different tradeoffs to consider, and Java went in the direction of static dispatching of parameters, while other OO programming languages might instead adopt dynamic dispatching of parameters.

However, there might still be ambiguous calls. Let's overload once more `Race.race`, adding a `race` method whose parameters are a `Vehicle` and a `Truck`.

```
public class Race {  
    public static int race(Vehicle v1, Vehicle v2, double length) {...}  
    public static int race(Car v1, Car v2, double length) {...}  
    public static int race(Car v1, Vehicle v2, double length) {...}  
    public static int race(Vehicle v1, Truck v2, double length) {...}  
}
```

The four definitions of `race` are accepted by the compiler since the static type of the parameters are different. We want now to execute the following code.

```
Car c = new Car(...);  
Truck t = new Truck(...);  
Race.race(c, t);
```

What implementation of `Race.race` is executed in this context? All the four definitions are applicable. So, what is the “best” one? I think we all agree that the first and third signatures are worse than the second and the fourth one. In particular, the second and fourth signatures are just one step from the static types of the call: in one case we have a truck instead of a car as second parameter, in the other case we have a vehicle instead of a car as first parameter. Instead, the first signature compared to the fourth would require also to move from vehicle to truck as second parameter, and the third one would require the same compared with the second signature. Therefore, let's focus our attention on the second and fourth signature. What is the best choice?

The answer is easy: none! They are “equivalently distant” from the parameters of the caller, so they are “equivalently good”. The Java compiler prevents the compilation of such code, and returns an error message stating that the call is ambiguous. The idea is that the runtime environment chooses the “best” signature, where by best we mean the nearest signature in terms of distance between the declared types of the parameters, and the static types of the expressions passed by the caller. Since those types are static (and therefore known at compile time), the compiler can check if the call is ambiguous, raising an error.

10.5 Dispatching of static methods

What happens when we invoke a static method? In such a case we do not have a receiver, and we have seen that if we invoke a static method through a receiver we get a warning. This happens because with static methods the compiler substitutes the receiver with its static type, and it statically invokes that method. Therefore, static methods are not polymorphic, since when we invoke a static method we know exactly what implementation we will be executing at runtime. Consider for instance the following code.

```
public class Racing {  
    public static void foo() { System.out.println("Racing 1");  
}  
}  
public class Racing2 extends Racing {  
    public static void foo() { System.out.println("Racing 2");  
}  
}  
Racing2.foo(); //Racing 2  
Racing.foo(); //Racing 1  
Racing racing = new Racing();  
racing.foo(); //Racing 1  
Racing2 racing2 = new Racing2();  
racing2.foo(); //Racing 2  
Racing racing3 = racing2;  
racing3.foo(); //Racing 1
```

First of all, please note that only the first two method invocations (`Racing2.foo()` and `Racing.foo()`) do not raise a compiler warning. In those cases, it is clear and evident what code is executed, since there is no ambiguity as we have only the name of the class containing the method. Also the following two cases are clear, since the static and dynamic type of the object used to invoke the method coincide. The last case indeed is problematic: the dynamic type of `racing3` is `Racing2`, while its static type is `Racing`. Since Java (for non-static methods) applies dynamic dispatching on the receiver, we would expect to execute the code in `Racing2`. Instead, with static methods Java executes the code in the static type (aka, applies static dispatching), and therefore the implementation in `Racing` is executed.

This behavior might look (and indeed is from our point of view) odd, because this approach is different from the standard dispatching. However, the compiler clearly states that the fact we rely on a receiver is inherently wrong for static methods, since, as the name suggests, those methods are statically bound at compile time. Therefore, we should always avoid to invoke static methods on receiver as the result might seem unexpected if compared with invocation of object methods.

10.6 Java dispatching algorithm

Let's try now to summarize how the Java algorithm resolves method calls at runtime. The following algorithm contains the main intuition of such an algorithm.

1. Extract the dynamic type of the receiver of the method call
2. Collect all the methods in the current class and in the superclasses
3. Return the method in this class whose signature matches the given one, where the matching is performed by
 - a. Same name
 - b. “Nearest” arguments looking to the static type
 - i. No ambiguity is possible: if two method signatures match the static arguments with the same “distance”, the code does not compile

However, please note that this is a rough approximation of the real algorithm. In particular, we ignored quite a lot of details and other aspects that make method dispatching more complex, such as generic types, and variable number of arguments. The Java language specification contains the complete and detailed description of this algorithm¹⁹, but we think it contains a lot of details that are not useful to understand how OO programming languages work when dispatching method calls, and they are specific for the Java technology.

10.7 Exercise

Given the following snippets of code, implement the TODOs:

¹⁹ <https://docs.oracle.com/javase/specs/jls/se11/html/jls-15.html#jls-15.12>

```
package it.unive.dais.pol.vehicle;

public class VehicleFactory {

    public static VehicleID createDefaultVehicleID(String type){

        switch(type){
            case "Bicycle":
                return ; // TODO: return a default bicycle
            case "Truck":
                return ; // TODO: return a default truck
            case "Car":
                return ; // TODO: return a default car
            default:
                return null;
        }
    }

    public static boolean haveTheSameIdentity (RegisteredVehicleID v1,
                                                RegisteredVehicleID v2) {

        ...
    }
}
```

```
public abstract class VehicleID {

    ...
}
```

```
public abstract class RegisteredVehicleID extends VehicleID {

    ...
}
```

```
public class CarID extends RegisteredVehicleID {

    ...
    public CarID(){ ... }
}
```

```
public class TruckID extends CarID {  
    ...  
    public TruckID(){ ... }  
}
```

```
public class BikeID extends VehicleID {  
    ...  
    public BikeID () { ... }  
}
```


Chapter 11: Generics

In this chapter we introduce another form of polymorphism called generic programming, that is the ability of a (piece of a) program to run with many different types passed as arguments. Intuitively, generics can be seen as type parameters. For instance, imagine that we want to implement a method that returns a parameter and nothing else. In order to be sure that we get back exactly the same type of the given parameter, we would need to implement a method for each type in our program. Instead, with generics we can parametrize our method on a type, and specify that the type of the parameter and the returned value is exactly the given type. The implementation of such a method might look as follows.

```
<T> T identity(T value) {  
    return value;  
}
```

The definition of the method should be read as: (i) `<T>` specifies that the method receives as parameter a type `T`, (ii) it returns a value of type `T`, (iii) the method name is `identity`, and (iv) the method receives a parameter of type `T`. Intuitively, everything that is between chars `<` and `>` is a list of type parameters (separated by commas), and these parameters can be used to define the type of method parameters and returned value. This code can be then executed as follows.

```
String s = <String>identity("Hello world");  
Vehicle v = <Vehicle>identity(new Vehicle(...));  
Truck t = <Truck>identity(new Truck(...));
```

Note that the type passed as parameter must be aligned with the type of the parameters and the assigned variable. For instance, a call `<String>identity(new Vehicle(...))` won't compile.

In this chapter, we will refer to generic types as generics, following the Java terminology. However, other programming languages adopted a different terminology (for good reasons). For instance, in C++ we talk about templates. Their syntax is almost identical to Java, but how they are supported is extremely different.

11.1 Classes

Classes can be parametrized on generics when they are declared, and these generics can

be used as the static type of (i) fields, (ii) parameters, and (ii) returned values. Generics are widely used when defining data structures, since these aim at containing any type of object and value. For instance, let's imagine we want to implement a list that allows us to add, get and check if the list contains an element. This list will be parametrized on a generic, that is, the type of the objects contained in the list.

```
public class List<V> {  
    private V[] elements = ...;  
    public void add(V el) {  
        int n = elements.length+1;  
        elements = Arrays.copyOf(elements, n);  
        elements[n-1] = el;  
    }  
    public boolean contains(V el) {  
        for(int i=0; i < elements.length; i++)  
            if(elements[i]==el)  
                return true;  
        return false;  
    }  
    public V get(int i) {  
        return elements[i];  
    }  
}
```

The generics can be declared after the name of the class using < and > characters as parenthesis. For instance, the above List class is parametrized on a generic type V, and this type is used to declare the type of the elements stored in the array, of the parameter of add and contains, and of the value returned by get. We can pass multiple generic types by separating them with commas in their declaration.

When we instantiate the class, we need to specify the generics we want to use. This is done by declaring them in the new statement, after the name of the instantiated class and before the list of parameters passed to the constructor. The generics are specified by using < and > characters as parentheses. For instance, the following snippet of code creates a list of vehicles.

```
List<Vehicle> v = new List<Vehicle>();
```

11.2 Methods

Sometimes we might want to parameterize a single method on a generic, and not a whole class. An example could be the above identity method, where the return type is the same as the type of the parameter, but it is not bound to the whole class. Generics are widely

applied to static methods. In fact, those methods are not bound to a specific instance of the class, and therefore they cannot rely on the generics defined in the class declaration. For instance, let's imagine that we want to implement a static method `toList` that, given an element, returns a list containing such element. Being static, this method does not have knowledge about the generics `V`. Therefore, we parameterize it on a generic `T`.

```
public class List<V> {  
    public static <T> List<T> toList(T value) {  
        List<T> result = new List<T>();  
        result.add(value);  
        return result;  
    }  
}
```

The generic is declared after the various modifiers (such as `public` and `static`) and before the return type. As with classes, the generics are declared between `<` and `>` characters, and they are separated by commas. Then they can be used as static types of parameters, returned values, and local variables. When we invoke this method, we need to specify the generics before the name of the method. For instance, the following snippet of code uses this method to get a list of vehicles containing only the freshly created car.

```
List<Vehicle> l = List.<Vehicle>toList(new Car(...));
```

11.3 Generics inference

Indeed, we do not really need to always specify the generics when instantiating class or invoking methods parameterized on types, since the Java compiler infers those types for us using information from the context of the `new` statement or the method call. First of all, we might pass an empty list of generics when instantiating a class, and the compiler will infer the generics for us. For instance, we can remove the parameter `Vehicle` when creating a `List` as follows.

```
List<Vehicle> v = new List<>();
```

However, note that this syntax is different from removing the `<` and `>` characters. In fact, the following statement causes a warning for a raw use of parameterized class `List`.

```
List<Vehicle> v = new List();
```

Intuitively, in the first case we are asking the Java compiler to infer the generic type. In the second case, we are using the class without passing any generics. How is this possible? Essentially, we have this flexibility because of backwards compatibility: Java did not support generics until version 1.5. Therefore, in order to compile programs written before this version with newer compilers, it is necessary to support instantiation

of classes with generics without any generic. However, nowadays we should always refrain from using such syntax, and we must therefore adopt the instantiation with `<` and `>` parentheses. Instead, we can invoke a method parameterized with generics without opening and closing such parentheses, and this won't lead to any warning, since it is not a raw use of a parameterized class. For instance, the previous call of `toList` can be rewritten as follows:

```
List<Vehicle> l = List.toList(new Car(...));
```

Note that the compiler infers `Vehicle` as the generic passed to `toList` since we assign the returned value to a variable of type `List<Vehicle>`, and the parameter (a new `Car`) is a subtype of `Vehicle`.

11.4 Generics invariance

Generics in Java are invariants. This means, for instance, that we can assign only a `List` of vehicles to a variable whose static type is a list of vehicles, and nothing else. For instance, one might think that it should be allowed to assign a list of cars to a list of vehicles (that is, to assign a list whose generic is a subtype of the other one), since once we get a value from a list of cars we definitely get a vehicle as well. However, if we add a bicycle to a list of vehicles, this would break the fact that the list of cars (assigned to the list of vehicles) contains only cars or subtypes of cars. This is shown by the following snippet of code.

```
List<Car> c = new List<Car>();  
List<Vehicle> l = c; //not allowed, this raises an error  
l.add(new Bicycle(...)); //allowed, since a Bicycle is a Vehicle  
Car e = c.get(0); //here we would get a Bicycle object instead of a car  
//this shows by the second assignment is not allowed by the Java compiler
```

However, the Java approach with arrays²⁰ is different. In fact, the following code compiles.

```
Car[] c = new Car[1];  
Vehicle[] l = c;  
l[0] = new Bicycle(...);  
Car e = c[0]; //here we would get a Bicycle object instead of a car
```

In fact, arrays in Java are covariant. This means that we can pass an array of elements of type `T1` where an array of elements of type `T2` is expected if `T1` is a subtype of `T2`. However, this incurs exactly in the problem pointed out with generics. So, what happens

²⁰ Note that one might think of an array as a class with a generic type, like lists. Since they are natively supported by the programming language they do not rely on generics, but the approach and issues are identical.

when we execute such code? An `ArrayStoreException` is raised when `l[0]` is assigned. Essentially, the Java runtime checks if the dynamic type of the element assigned to the array is compatible (aka, the same type or a subtype of) with the type declared when the array was created, and if it is not the case it raises an exception.

11.5 Bounding generics

So far, when we rely on generics we do not have any information or restrictions about the types that might be passed to the class or method. This works perfectly for data structures like lists, that aim at containing any kind of value, but it might be too flexible in others. For instance, let's imagine that instead of having a static method to run a race, we want to build a class that receives two objects that will run a race, and then implements a method `race` that receives only the length of the race, and it returns the object that won the race. Such an implementation might look as follows.

```
public class Race<T> {  
    private final T v1, v2;  
    public Race(T v1, T v2) {  
        this.v1 = v1;  
        this.v2 = v2;  
    }  
    public T race(double length) {  
        v1.fullBrake();  
        v2.fullBrake();  
        double distanceV1 = 0, distanceV2=0;  
        while(true) {  
            distanceV1 += v1.getSpeed();  
            distanceV2 += v2.getSpeed();  
            if(distanceV1 >= length || distanceV2 >= length) {  
                if(distanceV1 > distanceV2) return v1;  
                else return v2;  
            }  
            v1.accelerate(Math.random()*10.0);  
            v2.accelerate(Math.random()*10.0);  
        }  
    }  
}
```

Unfortunately, this code does not compile. In particular, the compiler raises an error all the times that we invoke a method (e.g., `fullBrake`) of class `Vehicle` on fields `v1` and `v2`. In fact, the type `T` might be any type: a `Vehicle`, a `Car`, or a `String`! Therefore, no assumption can be made on its interface. Java gives the possibility to bound generics to a given supertype. This can be achieved by following the generic declaration with a clause

extends <type>. In this way, the compiler enforces that when the class is instantiated the generics passed is Vehicle or one of its subtypes.

```
public class Race<T extends Vehicle> { ... }
```

The advantage of having a generic bound to Vehicle instead of using Vehicle is that, for instance, we can instantiate class Race with two trucks, and then method race will return a truck (and not a generic vehicle). This is shown by the following snippet of code.

```
Race<Truck> r = new Race<>(new Truck(...), new Truck(...);  
Truck t = r.race(100);
```

11.6 Wildcards

So far, we have seen that once we declare a static type that is parameterized with generics we must specify a type. Java allows one not to exactly specify such type, and instead pass a so-called wildcard (represented by a question mark) in the declaration. The wildcard means that there exists a type that allows that code to compile, but it does not provide any specific guarantee on such type. Therefore, we might declare a variable l with type List<?>, but once we retrieve elements from such a list we do not know if they are vehicles, cars, or what else. Consider for instance the following code.

```
List<?> l = new List<Vehicle>();  
l.add(new Car(...));  
Vehicle v = l.get(0);
```

However, neither the second nor the third line compiles. In fact, the compiler knows that l is a list of something, but if we add a Car, it cannot be sure that such a list can contain cars (e.g., it might be a list of strings). Similarly, when we retrieve something, we do not know if it might be a Vehicle, a string, or anything else. However, we can bound wildcards through the extends keyword as we did previously with generics.

```
List<? extends Vehicle> l = new List<Vehicle>();  
l.add(new Car(...));  
Vehicle v = l.get(0);
```

Such a bound states that the generic of List l is Vehicle or one of its subtypes. With this restriction, the third line compiles since any List with a generic that is Vehicle or a subtype will return a value compatible with type Vehicle. However, the second line still does not compile. In fact, l might be a list of any subtype of Vehicle, and, for instance, a list of bicycles. In such a situation, we are not guaranteed that we can add a Car to the list.

With wildcards we also have a different type of bound. In particular, the extends

keyword states that the generic must be the same or a subtype of the given type. We can instead specify that the generic must be the same or a supertype of the given type through the super keyword.

```
List<? super Vehicle> l = new List<Vehicle>();  
l.add(new Car(...));  
Vehicle v = l.get(0);
```

Here we declared that `l` is a list of Vehicles or any supertype of vehicles. Therefore, the second line now compiles, since a `Car` is definitely a subtype of `Vehicle` or any of its supertypes. Instead, the third line does not: since we might have a supertype of `Vehicle` (e.g., the class `Object` we will introduce in the next chapter), we are not guaranteed that method `get` will return a vehicle.

11.7 Java support for generics

For almost a decade Java did not support generics. This was a deliberate design choice, since when Java was born there were already OO programming languages that supported this programming pattern (such as C++ with templates). Generics were added only later, and the backward compatibility imposed several restrictions on their usage and flexibility, as well as a rather complex runtime support. Several of these limitations (and in particular invariance of generics) might restrict their applications and usage. For these reasons, some other constructs, such as wildcards, were provided to make generics more flexible, but they still suffer, at least from a theoretical perspective, by other limits. Covariance and contravariance of generics goes beyond the scope of this book, and therefore we do not treat it in detail. We suggest to the interested reader to study other programming languages that have a richer support for generics (and Scala in particular) to better understand and practice the full opportunities that generics offer.

11.8 Exercise

Implement from scratch a generic class `MyStack` and a `main` method that use it. `MyStack` is a class that represents a stack data structure. A stack is a collection of elements, with two main principal methods:

- `push`, which adds an element to the collection
- `pop`, which removes the most recently element from the collection

The order in which elements come off a stack is LIFO (last in, first out).

Part 3: Java in action

Chapter 12: Object

So far we have seen how we can build up our type hierarchy of the classes we define. We can define that a class extends (thus inherits all components and is a subtype of) another class, or that it implements (thus must implement all components defined in and is a subtype of) an interface. This creates a local class hierarchy among the types defined by the interfaces and classes of our program. However, the Java platform provides us a richer environment, as well as several classes that are needed by almost all Java programs. In this chapter, we will discuss the `Object` class, while in the next chapter we will present the main classes of the core package of the Java library.

12.1 The superclass of all classes in Java

Consider the `Vehicle` class we previously defined: it does not have an `extends` clause, therefore it looks like it is not a subclass of any other class. Indeed, if we do not specify that a class extends another one, the Java compiler implicitly makes it extend the `java.lang.Object` class²¹. This means that all Java classes extend directly or indirectly `Object`, and we can assign any object to a component that has static type `Object`. Consider for instance the following code.

```
Object[] array = new Object[3];  
array[0] = new Car(...);  
array[1] = new Bicycle(...);  
array[2] = "hello world";  
Object element = array[<random>];
```

This snippet of code compiles and executes correctly (assuming we get a random index between zero and two). In the array, we store a bunch of different objects (a car, a bicycle, and a string) and then we retrieve a random element from the array. But what we can do with this element? Essentially, we can access only the components that are available in the `Object` class.

This is a generic class provided by the Java platform that contains some standard methods that are useful for any class we might think of. Some of these methods (`wait`, `notify` and `notifyAll`) provide utilities for concurrency that are outside the scope of this book. The method `finalize` has been deprecated starting from Java 9. Method `getClass`

²¹ <https://docs.oracle.com/en/java/javase/11/docs/api/java.base/java/lang/Object.html>

instead is useful for reflection purposes that we will introduce later. Therefore, in this chapter we will focus on the methods that provide semantic information and operations about a generic object, and in particular on equals, hashCode and clone. All these methods are non-final and therefore can (and sometimes should) be overridden in order to redefine their behaviour.

12.2 equals

This method receives an object as a parameter, and it returns true if and only if the current object is equal to the given one. First of all, why should we rely on such a method instead of using the equality comparison ==? Essentially, this comparison returns true if and only if the references of the objects are the same. If we had two distinct objects that indeed represent exactly the same entity (and therefore they are semantically equals) this operator returns true. However, class Object implements this method by returning the result of this comparison. Such a choice is reasonable since at a very generic level (as the one of class Object) we can only state that two objects are equal only if they are exactly the same object. On the other hand, we can redefine the notion of equality in our classes by overriding method equals.

Let's go back to the definition of our class FuelType. This time, we establish that two fuel types are equal if all their fields are equal, that is, if the name, cost per liter, and speed increment per liter of fuel are the same. This can be implemented as follows.

```
class FuelType {  
    public boolean equals(Object o) {  
        if (o == null || !(o instanceof FuelType) )  
            return false;  
        FuelType f = (FuelType) o;  
        return f.costPerLiter==this.costPerLiter && f.litresPerKmH==this.litresPerKmH &&  
            this.type.equals(f.type);  
    }  
}
```

Intuitively, we first check if the given object is null or not an instance of FuelType. In these cases, we know it is different from the current object, and we return false. Otherwise, we cast the object to FuelType and we check if the three fields contain the same values²². Note that in order to check the equality of fields pointing to objects (such as field type that is a string) we rely on the implementation of method equals in their

²² Remember that private fields are accessible inside the same class. Therefore, we can access private fields not only of the object on which the method was invoked (this), but also of the object passed as parameter.

classes by calling this method.

However, an equality operator must satisfy some properties. In particular, it must be (i) reflexive (meaning that `x.equals(x)`, that is, an object must be equal to itself), (ii) symmetric (`x.equals(y)` if and only if `y.equals(x)`), and (iii) transitive (if `x.equals(y)` and `y.equals(z)` then `x.equals(z)`). If our implementation of method `equals` does not respect these properties we might encounter unexpected behaviors by method relying on `equals`. Nevertheless, these properties cannot be neither checked at compile time nor enforced through the syntax of the program. Therefore, they are documented in the Javadoc of class `Object`²³, and any class overriding method `equals` should provide an implementation satisfying those requirements.

12.3 clone

Another method defined by `Object` is `clone`. This method does not receive any parameter, and it returns an `Object`. Its semantics is to produce an object that is (i) identical to the current one, but (ii) it is a different object. Formally, this means that (i) `o.equals(o.clone())`, and (ii) `o != o.clone()`. Therefore, the implementation of method `clone` is strictly related to the implementation of method `equals`. For instance, `clone` can be implemented as follows in class `FuelType`.

```
class FuelType {  
    protected Object clone() {  
        return new FuelType(type, costPerLiter, litresPerKmH);  
    }  
}
```

However, note that `clone` returns an object, and therefore when we use it we should check the type and cast the returned value. However, we can rewrite the code as follows, and this still compiles and executes correctly.

```
class FuelType {  
    protected FuelType clone() {  
        return new FuelType(type, costPerLiter, litresPerKmH);  
    }  
}
```

When cloning an object we have a choice: clone all the objects pointed by fields (deep cloning), or not (shallow cloning). In the first case, we avoid that the cloned object aliases

²³ The documentation specifies some more properties, such as consistency, that we left out from our notes. The full documentation can be retrieved at: [https://docs.oracle.com/en/java/javase/11/docs/api/java.base/java/lang/Object.html#equals\(java.lang.Object\)](https://docs.oracle.com/en/java/javase/11/docs/api/java.base/java/lang/Object.html#equals(java.lang.Object))

some fields of the original object, but we allocate more memory. In the second case, we alias the objects pointed by the fields of the two objects (thus the modification of the content of the object pointed by a field will be visible in the other object), but we do not allocate further memory.

In `FuelType`, the only field pointing to an object is `type` that points to a string. In this case, we performed a shallow copy. Since, as we will discuss in the next chapter, `String` class is immutable (that is, there is no way to change the data contained in the object after it has been instantiated), it does not make any difference to clone it, and in this way we avoid allocating memory.

Remember that when we override a method, we should have a definition that gives the same guarantees of the definition in the superclass. Since `FuelType` is a subtype of `Object` (like all classes in Java), if a method guarantees to return a `FuelType` object, it also guarantees to return an `Object` instance. Therefore, it is allowed to override a method with another one whose returned type is a subtype of the one in the superclass.

Unfortunately, we have a second issue: `method clone` is protected. This means that we cannot clone a generic object, while instead we can check equality among generic objects. However, also in this case we can define the clone method as follows when overriding it.

```
class FuelType {  
    public FuelType clone() {  
        return new FuelType(type, costPerLiter, litresPerKmH);  
    }  
}
```

We already discussed that we can give a less strict visibility to an overridden method for the same reasons explained above, since such a method will be visible from all the locations of the method in the superclass plus something more. Therefore, it is up to us to make the return type more specific and/or the method public.

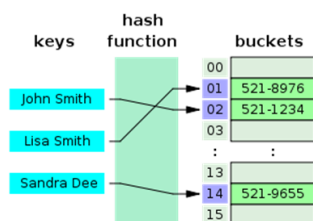
A workaround might be to rely on the `Cloneable` interface²⁴. “By convention, classes that implement this interface should override `Object.clone` (which is protected) with a public method”. However, this interface is empty and it does not define a public clone method. “Therefore, it is not possible to clone an object merely by virtue of the fact that it implements this interface. Even if the clone method is invoked reflectively, there is no guarantee that it will succeed.” So, why do we have this interface? Short answer: that’s

²⁴ <https://docs.oracle.com/en/java/javase/11/docs/api/java.base/java/lang/Cloneable.html>

just a poorly-designed interface²⁵.

12.4 hashCode

Object's hashCode method is public, does not receive any parameter, and it returns an integer value. Its goal is to provide a hash code to be used by hash tables to index elements in those data structures. Therefore, in order to understand the semantics and the constraints of this method we need to understand how hash tables work (even if this is not a topic related to object oriented programming).



“A hash table uses a hash function to compute an index, also called a hash code, into an array of buckets or slots, from which the desired value can be found. During lookup, the key is hashed and the resulting hash indicates where the corresponding value is stored.”²⁶

Intuitively, this data structure usually is more efficient in practice, but it requires more memories than other data structures. Method hashCode is used in order to know the index where an element should be stored, or where we should lookup for a given element.

Therefore, an essential property for a hash code is that if two objects are equals, then their hash code is the same. Otherwise, we might look up for an element in an index while it is stored in another one, or we might store an element in the wrong index. Note that the vice-versa does not hold, and it should not be the case that two objects with the same hash code are guaranteed to be equals. In fact, in such a case we would have an index for each single possible element, and this probably would consume and waste a lot of memory to store the elements.

A trivial implementation of hashCode would be to return a constant value (e.g., 0). In this way, if two elements are equal, they surely have the same hash code. However, such implementations would compromise any computation benefit of using hash tables, since at that points all elements will be stored and looked up on the same index. Generally speaking, implementing correctly and efficiently the method hashCode requires some

²⁵

<https://stackoverflow.com/questions/9981796/why-does-java-lang-cloneable-not-override-the-clone-method-in-java-lang-object>

²⁶ https://en.wikipedia.org/wiki/Hash_table

knowledge on the data structure and the number of elements that we might have that goes beyond the scope of this book (that is to introduce OO programming languages). Common IDEs already provide several utilities to auto-generate this method, even if such implementations might not be optimal. Another option might be to rely on the hashCode of one or some fields. For instance, we might implement hashCode on FuelType as follows.

```
class FuelType {  
    public int hashCode() {  
        return type.hashCode();  
    }  
}
```

Apart from the various aspects of hash tables and their complexity, something we must be always aware and remind is that if we override method equals, we always must override method hashCode, since otherwise the requirement that if `o1.equals(o2)` then `o1.hashCode()==o2.hashCode()` might not hold. The Java compiler produces a warning if only one of the two methods is overridden.

12.5 toString

The last method of class Object we discuss in this chapter is toString. This public method does not receive any parameter, and it returns a string representing the current object. The idea is that the string provides a succinct representation of the state of the object. However, there is no restriction in the documentation about how the object should be represented, and about relationships between this method and other methods in Object such as equals. Indeed, the toString method provides only a utility that allows to represent textually the object in any context, and it is widely used in debuggers (so we do not need to inspect the fields of an object to see its content but we can simply look to its string representation), user output (e.g., through console), and sometimes to dump the state of objects to text (e.g., XML or JSON) formats. The implementation of the toString method in Object returns the name of the class of the object concatenated with a @ character followed by a representation of the address of the object. Usually, such representation does not tell much about the state of the object. Therefore, it's always a good idea to override this method, since it will ease the development and debugging process, but it won't affect the coherence of libraries' behaviors (while instead this was the case for equals and hashCode).

For instance, we can implement the method toString of FuelType as follows.

```
class FuelType {  
    public String toString() {  
        return type+", cost " + costPerLiter + ", performance "+litresPerKmH;  
    }  
}
```

In this way the string representing a fuel type will contain all the information of the object (name, cost per liter, and performance of the specific type) in a very concise way. Note that with such representation we have that if two objects have the same string representation then they are equals, and vice versa. However, this is not a property we must guarantee, and indeed when we have rich and complex objects we often do not want since representing the full data contained in such objects might be particularly verbose.

12.5 Exercise

Given the following classes, implements the methods equals, hashCode and toString.

```
public abstract class VehicleID {  
  
    protected final String make;  
    protected final String model;  
    ...  
}
```

```
public class BikeID extends VehicleID {  
  
    private final String chassisNumber;  
    ...  
}
```

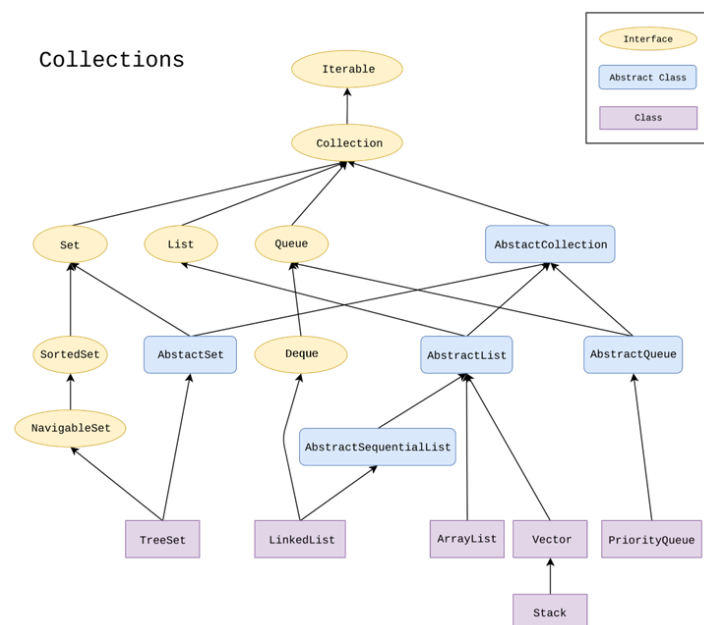
```
public class EBikeID extends BikeID {  
    private String batteryNumber;  
    ...  
}
```

Chapter 13: The core Java type hierarchy

In this chapter, we introduce the main classes of the core packages of the Java platform, and the relations among them. When we talk about core packages, we mostly mean (i) all the content of `java.lang` (that is implicitly imported by any Java class), and (ii) a part of `java.util` (that needs to be explicitly imported).

13.1 Collections and iterators

When we introduced and discussed generics, we stated that data structures are a common application of generics without bounds, since they are aimed at containing and managing any type of object. Therefore, they rely on the methods provided by class `Object`. For instance, if we try to retrieve an element from a collection or check if an element is contained, the implementation of the collection will rely on method `equals` to check if an element contained in the collection is the same of the given one. Similarly, hash tables rely on `hashCode` as discussed above.



Collections provide a deep and complex example that show how complex a type hierarchy can become. The figure above shows only a small part of this. The root of this hierarchy is interface `Iterable` that provides a method returning an iterator (that is, an object that allows to check if there is a next element and get it). Collection provides many more functionalities, such as methods to add and remove elements. When we move down in the hierarchy, we add more and more functionalities as well as some implementations. For instance, an `AbstractSet` implements several methods by following the semantics of sets (that is, an element can be contained at most once in a set), while a `LinkedList` contains a sequence of (potentially equal) elements.

Since `Iterable` is part of the core packages of the Java platform, the programming language relies on that, and it contains a construct that exploits the interface of `Iterable`. In particular, let's assume that we have an iterable variable of type `Iterable<Vehicle>`. We can iterate on this iterable with the following for each loop.

```
for(Vehicle v : iterable)
    v.fullStop();
```

This for each loop is compiled as follows.

```
Iterator<Vehicle> it = iterable.iterator();
while(it.hasNext()) {
    Vehicle v = it.next();
    v.fullStop();
}
```

13.2 Sorted set and Comparable

Let's focus on a small subset of the collection hierarchy, and in particular on `SortedSet`²⁷ and on one of its implementations (`TreeSet`²⁸). Why is it interesting to have a sorted collection? The most popular reason is to obtain deterministic programs, that is, programs that, given an input, produce always the same output. If the collection is not sorted, we do not have any guarantee on the order of the elements when we iterate over them. For instance, given a collection with two elements `o1` and `o2`, this means that in one execution we could start from `o1`, and in another one from `o2`. If the output depends on how we iterate over the elements (e.g., if we print the objects one by one on the console), we would get different results in different executions, and this for instance might fail some tests. Therefore, sorted collections became quite popular.

²⁷ <https://docs.oracle.com/en/java/javase/11/docs/api/java.base/java/util/SortedSet.html>

²⁸ <https://docs.oracle.com/en/java/javase/11/docs/api/java.base/java/util/TreeSet.html>

However, so far, we can only check if an object is equal to another one, but not if one is less than the other. If we want to sort a collection of objects, we need an operator that allows us to compare two objects. Java defines the `java.util.Comparable`²⁹ interface. This interface is parametrized on a generics `T`, and it defines only a method `compareTo` that, given an object of type `T`, returns an integer value. This method “returns a negative integer, zero, or a positive integer as this object is less than, equal to, or greater than the specified object”. As for `equals` and `hashCode`, we have several requirements that this method must satisfy. Since the sign of the value returned by the method indicates if the object is less than the other one, then the sign of `x.compareTo(y)` should be the opposite of `y.compareTo(x)`. In addition, if `x.compareTo(y)` returns zero, then the two objects are equals³⁰ and vice versa.

We now make `FuelType` implementing `Comparable<FuelType>`, and implement method `compareTo` as follows.

```
public class FuelType implements Comparable<FuelType> {
    public int compareTo(FuelType o) {
        if(this.equals(o)) return 0;
        else if(! this.type.equals(o.type))
            return this.type.compareTo(o.type);
        else if(this.litresPerKmH!=o.litresPerKmH)
            return (int) (this.litresPerKmH-o.litresPerKmH);
        else return (int) (this.costPerLiter-o.costPerLiter);
    }
}
```

Let's look at this implementation. First of all, if they are equals, we return zero, satisfying part of the last condition we mentioned. Then, if the string containing the type of the fuel is different, we return the results of method `compareTo` implemented in class `String`, and this guarantees us that the various constraints are respected. The last three code lines simply checks if the value in the corresponding fields are different (note that the last line is reachable only if `costPerLiter` is different, since otherwise all the three fields would be equal and this case is covered by the initial equality check), and it returns the difference between value. However, since `compareTo` returns an integer and the fields contain a double value, we need to cast to `int` the result of the subtraction. Unfortunately, this creates a bug in our implementation: if the difference between the two values is between

²⁹ <https://docs.oracle.com/en/java/javase/11/docs/api/java.base/java/lang/Comparable.html>

³⁰ Indeed, the Java documentation states that “it is strongly recommended, but not strictly required that `(x.compareTo(y)==0) == (x.equals(y))`. Generally speaking, any class that implements the `Comparable` interface and violates this condition should clearly indicate this fact. The recommended language is “Note: this class has a natural ordering that is inconsistent with `equals`.”” For the sake of simplicity, we left out from the notes this option.

0 and 1 or -1, this returns zero, but the two objects would be different! This shows also how subtle it is to implement these methods. Nowadays, modern IDEs often provide functionalities that automatically implement these standard methods in order to take care of all standard corner cases. Anyway, we fix our implementation as follows.

```
public class FuelType implements Comparable<FuelType> {  
    public int compareTo(FuelType o) {  
        if(this.equals(o)) return 0;  
        else if(! this.type.equals(o.type))  
            return this.type.compareTo(o.type);  
        else if(this.litresPerKmH!=o.litresPerKmH)  
            return roundItUp(this.litresPerKmH-o.litresPerKmH);  
        else return roundItUp(this.costPerLiter-o.costPerLiter);  
    }  
    private int roundItUp(double v) {  
        if(v>=0.0 && v<1.0) return 1;  
        else if (v<=0 && v>-1) return -1;  
        else return (int) v;  
    }  
}
```

Now we can create a `TreeSet` and add to it several elements. However, class `TreeSet` (as well as `SortedSet`) is parametrized on a generic type `T` that does not need to be a subtype of `Comparable`. What happens if we add an object that is not comparable to a `TreeSet`? We get a `ClassCastException` stating that an object is not an instance of `Comparable`. The choice of the developer of the Java platform was not to restrict the generic to instance of `Comparable`, since we might have classes that do not implement comparable, but we might define our own comparator and build up an ordered collection using this. Therefore, we are not ensured statically that all the elements can be compared, and the library at runtime crashes in the case it is unable to compare objects inside the ordered collection.

13.3 String

Another class that is used by any Java program is `java.lang.String`³¹. This class represents character strings, and it internally contains all the information required to encode and decode strings. In particular, it contains a private field pointing to an array of bytes, and it relies on the UTF-16 encoding with some extensions to support additional characters.

However, a developer should not take care of all those aspects. In fact, string literals are part of the Java programming language, and one can create a string by simply writing

³¹ <https://docs.oracle.com/en/java/javase/11/docs/api/java.base/java/lang/String.html>

some characters between double quotes as follows.

```
String s = "abc";
```

However, please note that this is a shortcut supported by the Java compiler that indeed compiles it into the creation of a string object as follows.

```
char[] value = {'a', 'b', 'c'};  
String s = new String(value);
```

String objects are immutable. This means that once they have been created, their internal state cannot be changed. Instead, `StringBuffer`³² supports mutable strings. Immutability is a desired property when we want to share an object with many others. In fact, even if an object is aliased by many locations (see Section 3.7 about aliasing), we might have side effects hard to track and detect, since the modification of the internal state of the object from a location would be visible to all aliases. Since immutable objects cannot change their internal state, we cannot have side effects through aliases. How to achieve an immutable object in Java is a rather complex process. In particular, it is not enough to define all fields as final. In fact, a final field pointing to an object might still be mutable since we might modify the state of the referenced object. Therefore, one needs to properly encapsulate the state of the object and prevent it from modifications. Other programming languages provide different mechanisms (such as the `const` modifier for fields in C++) to ensure that an object is immutable.

Apart from string literals, Java natively supports the concatenation of strings through the `+` operator. In particular, given two strings `s1` and `s2`, `s1+s2` returns a sequence of characters starting with the ones of `s1` and followed by the ones of `s2`. Note that `s1` and `s2` are not modified by this operation, and this would not be possible anyway since strings are immutable.

```
String s1 = "abc";  
String s2 = "def";  
String s = s1+s2; //"abcdef"
```

While the implementation of the concatenation of strings is left to the compiler, `String` class provides a `concat` method that concatenates two strings. For instance, `s1+s2` is semantically equivalent to `s1.concat(s2)`. Since strings are immutable, when we concatenate two strings the runtime environment needs to (i) create a string buffer, (ii) add all the characters of the first string, (iii) add all the characters of the second string, and (iv) return the string represented by this buffer. Therefore, handling string operations might be particularly expensive both in terms of memory and computations. For instance, let's look again to `toString` method we implemented in the previous chapter

³² <https://docs.oracle.com/en/java/javase/11/docs/api/java.base/java/lang/StringBuffer.html>

in the FuelType class.

```
class FuelType {  
    public String toString() {  
        return type+"", cost " + costPerLiter + ", performance "+litresPerKmH;  
    }  
}
```

Even if this is just one line of code, we have two string constants, and four applications of the concat operators. Overall, we have the creation of two String objects for the literals and four other objects for the concatenation, and four string buffers for the concatenation³³. Therefore, strings often represent a critical resource in terms of memory and resource allocation and consumption in Java programs, and they should be carefully treated (e.g., by having a unique string buffer instead of applying several times the concatenation through the + operator in the example above).

13.4 Primitive types

In Section 3.5 and 3.6 we already discussed value and reference types. In the previous chapter, we saw that all classes are subtypes of Object. Therefore, all reference types are subtypes of Object. But what about value types?

Generally speaking, we talk about primitive types to indicate all the types that are not references, and they are usually adopted to deal with numerical values. In Java we have boolean, byte, char, short, int, long, float, and double. All those types are not a subtype of Object. The table below summarizes the different primitive types and their information.

Type	Bits	Floating Point	Literals
boolean	1		true, false
byte	8		
char	16		'a', 'b', '\n', '\t'
short	16		
int	32		12, 564, -436
long	64		12L, 564L, -436L

³³ Starting with Java 9 the Java compiler deeply revised how strings are treated during compilations, and in the latest releases we noticed significant differences between how the Oracle compiler and OpenJDK compiles them. Therefore, this description might not be accurate for Java 9+ compilers.

float	32	X	1.23F, -54.3F, 1F
double	64	X	1.23D, -54.3D, 1D

First of all, note that those types (or similar ones) exist more or less in any (not necessarily object-oriented) programming language. Their semantics and behavior in Java reflects their common understanding. In particular, several implicit type conversions apply. When we assign a value that has a "lower" precision that has a "higher" prevision, the statement is compiled and the value is implicitly converted. But what do we mean by precision? Two simple general rules apply: (i) floating point values are more precise, and (ii) if we have a higher number of bits, we are more precise. Consider, for instance, the following assignments.

```
byte a = 3; //int -> byte
char b = 'a'; // char -> char
int c = b; //char -> int
c = a; // byte -> int
a = c; // int -> byte not allowed
long d = c; //int -> long
c = d; //long -> int not allowed
float e = c; // int -> float
e = d; //long -> float
double f = e; //float -> double
e = f; //double -> float not allowed
```

We cannot assign an integer value (32 bits) to a byte variable (8 bits), a long value (64 bits) to an integer variable (32 bits), a double value (64 bits) to a float variable (32 bits). In those cases, we can either (i) explicitly cast the expression to the desired type, or (ii) use the methods provided by the wrapper classes. In the example above, we might fix the three cases as follows.

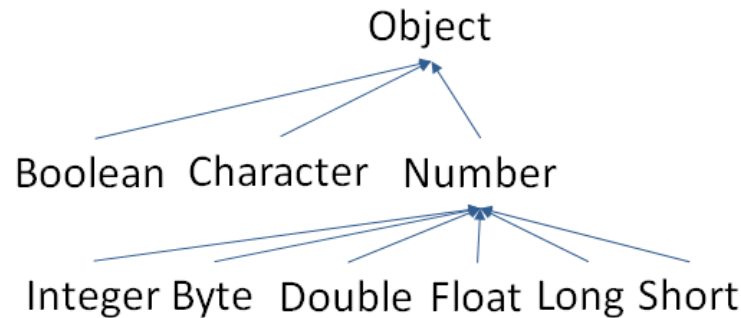
```
a = Integer.valueOf(c).byteValue(); // int -> byte
c = Long.valueOf(d).intValue(); //long -> int
e = Double.valueOf(f).floatValue(); //double -> float
a = (byte) c; // int -> byte
c = (int) d; //long -> int
e = (float) f; //double -> float
```

Note that in this way we are forced to convert the results in all the cases in which we might lose some information and potentially fall into numeric overflows.

13.5 Wrappers

In the code snippet above, we used classes Integer, Long, and Double. All those classes

belong to the `java.lang` package and they are therefore part of the core libraries of Java. In particular, we have one of these classes per each primitive type we discussed in the previous section. Those classes are intended to wrap a primitive value inside an object, in order to use them in all the places where a reference type (and not a primitive type) is needed.



The figure above reports the full type hierarchy of these classes. In particular, we have a two-level structure in order to deal with numbers in an uniform way, while booleans and characters are left out. Those classes provide various utilities to deal with primitive numbers, such as converting a primitive value to a wrapper object (as shown in the previous section), minimal and maximal values, conversion from a primitive type to another primitive type (instead of using explicit dynamic casting directly), etc...

Wrapper classes are needed in all the contexts where we need a generic object instead of a primitive value. The most common use case is with generics. For instance, let's imagine that we want to create a list of integers. Unfortunately, the statement “`new ArrayList<int>()`” does not compile since primitive types are not allowed as type arguments. Overall, this makes sense: when we have generics without bounds we rely on the methods provided by the `Object` class in order to compare and represent as strings those objects. However, creating a list of integer values is definitely something that we must be able to do. Therefore, we can create a List of Integer objects, and use this to manage a collection of integer values as in the following snippet of code.

```
List<Integer> list = new ArrayList<>();  
list.add(Integer.valueOf(1));  
Integer element = list.get(0);  
list.add(2);  
int element = list.get(0);
```

The first three lines of code are pretty straightforward: we create an `ArrayList` of `Integer` objects and we assign it to the `list` variable, we add an instance of `Integer`, and we

retrieve the first element in the list, storing it in an Integer variable. Indeed, the last two lines might look surprising. In particular, we call method `add` passing an `int` value. We would have expected, following the specification of `List`, that the `add` method required an `Integer` object, and not an `int` value. How is it possible that this line compiles and executes correctly? Similar considerations apply to the last line: we retrieve an element of a `List<Integer>` and we assign it to an `int` variable!

Intuitively, objects representing primitive data types are automatically boxed and unboxed when needed by the compiler. Therefore, when we compile `list.add(2)`, the Java compiler detects that we have a list of `Integer` objects, we pass an `int` value, and therefore it compiles this statement to `list.add(Integer.valueOf(2))`. Similarly, when we compile `element = list.get(0)`, the Java compiler sees that we assign an `Integer` object to an `int` variable, and therefore compile the statement to `element = list.get(0).intValue()`. However, remember that all these operations come at a computational and memory consumption cost, and therefore we should minimize the creation of wrapper objects when possible. The Java compiler by default produces warnings in many contexts where we have a useless boxing or unboxing of these values.

In addition, implicit conversion rules do not apply when automatically boxing or unboxing values. For instance, we cannot assign an integer value to a `Double` variable, or a byte value to an `Integer` value. All the following statements do not compile, and they raise a type error.

```
Integer i = 1.0;  
Double d = 1;  
d = i;
```

13.6 Exercise

Given the following class, make it comparable and print the license plate of the vehicles contained in the `main` method sorted by in alphabetical order (ascending order) and license plate length (descending order).


```
public abstract class RegisteredVehicleID {

    protected final String make;
    protected final String model;
    protected String licensePlate;
    protected final String VIN;

    public RegisteredVehicleID (String make, String model,
                                String licensePlate){

        this.make = make;
        this.model = model;
        this.licensePlate = licensePlate;
    }

    ...

    public abstract boolean isValidLicensePlate();

    public boolean isSameLicensePlate(String licensePlate) {
        return this.licensePlate.equals(licensePlate);
    }

    public String getMake() { return make; }

    public String getModel() { return model; }

    public String getLicensePlate () { return licensePlate; }

    public void setLicensePlate(String licensePlate) {
        this.licensePlate = licensePlate;
    }

    public String getVIN() { return VIN; }
}
```

```
public void main(String[] args) {  
  
    RegisteredVehicleID v1 = new CarID("Fiat", "500", "AA999ZZ");  
    RegisteredVehicleID v2 = new CarID("Ferrari", "F40", "BC111XX");  
    RegisteredVehicleID v3 = new MotorcycleID("Ducati", "Monster", "AA99999") ;  
    RegisteredVehicleID v4 = new CarID("Bugatti", "Veyron", "BA111XP");  
    RegisteredVehicleID v5 = new MopedID("Piaggio", "Vespa", "11222X") ;  
  
    //TODO: code implementation here  
    // expected print result: "AA99999", "AA999ZZ", "BA111XP", "BC111XX", "11222X"  
}
```

Chapter 14: Exceptions

The normal execution of a program might be suspended because an error occurs. There are several reasons that might cause the abruptive termination of a program, such as an access to an invalid reference, a division by zero, or out-of-memory errors. Java (and in general all OO programming languages) manage such situations through exceptions, that are particular objects that collect and represent useful information in such contexts.

14.1 Runtime errors and exceptions

When an error occurs, the normal execution of a program is suspended, and the error is propagated. If this error reaches the “external level”, the program is terminated abruptly and an error is reported. When we talk about normal execution, we mean that the statements are sequentially executed following the flow of the instructions (e.g., taking into account conditional jumps, loops, etc..). When an error occurs, it is not possible to follow this flow as we are not able to compute a valid state after a statement. Consider for instance the following class and code.

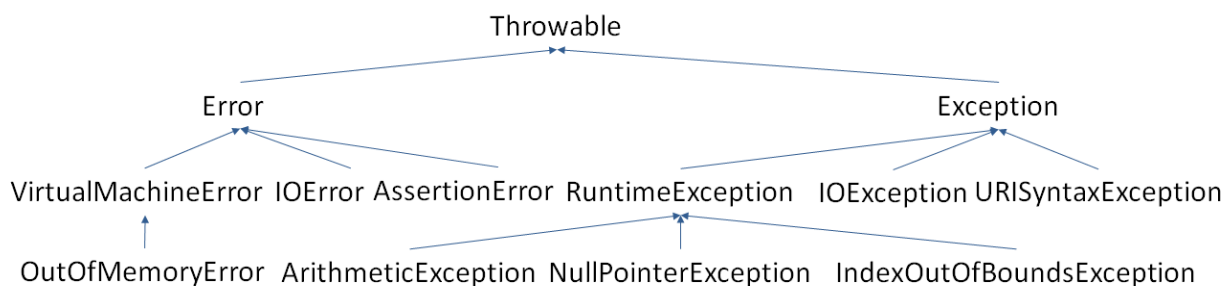
```
class Car {  
    private double fuel = 0;  
    private final FuelType fuelType;  
    public Car(double initialSpeed, FuelType f) {  
        super(initialSpeed);  
        fuelType = f;  
    }  
    public void accelerate(double amount) {  
        double f = computeConsumedFuel(amount, fuelType.getLitresPerKmH());  
        if(f < fuel) {  
            super.accelerate(amount);  
            fuel = fuel - fuelConsumed;  
        }  
        else {  
            double increaseSpeed = fuel / fuelType.getLitresPerKmH();  
            super.accelerate(increaseSpeed);  
            fuel = 0;  
        }  
    }  
}
```

```
Car c = new Car(0, null);
c.accelerate(10);
```

What happens when we execute the invocation of method `accelerate`? The first line of this method dereference `fuelType` that indeed contains a null pointer. Therefore, the normal execution is suspended and the program crashes. In particular, we get the following output in the console (assuming that the executed code stays in the main method of class `Car`).

```
Exception in thread "main" java.lang.NullPointerException
    at it.unive.dais.pol.vehicles.autovehicles.Car.accelerate(Car.java:102)
    at it.unive.dais.pol.vehicles.autovehicles.Car.main(Car.java:119)
```

This output reports to us the type of error that occurred (a `NullPointerException`), where (line 102 of file `Car.java`), and the stack of method calls (main called `accelerate` at line 119 of `Car.java`). Therefore, exceptions allow us to distinguish between different types of runtime errors. Errors are represented through standard Java classes that implement some specific functionalities.



The figure above illustrates the main families of errors we might have (in particular, `Error`, `Exception` and `RuntimeException`) and some specific types that are widely adopted by the Java libraries and runtime environment. In particular, the root of all classes representing errors is `Throwable`³⁴. This class encompasses all methods to store and retrieve an error message, set and print a stack trace, and to nest several errors.

14.2 Throwing exceptions

Java allows throwing exceptions explicitly through the `throw` statements. This statement is followed by an expression that must return an object that is subtype of `Throwable`. Usually, a `throw` keyword is followed by a new expression that instantiates a class that is subtype of `Throwable`. For instance, we want to modify method `accelerate` in `Vehicle` in

³⁴ <https://docs.oracle.com/en/java/javase/11/docs/api/java.base/java/lang/Throwable.html>

order to throw an `ArithmeticException` if the speed passed as parameter is negative.

```
class Vehicle {  
    public void accelerate(double a) {  
        if(a >= 0)  
            this.speed += a;  
        else throw new ArithmeticException("Negative values for speed are not allowed");  
    }  
}
```

We now execute the following snippet of code.

```
Car a = new Car(0, new FuelType("diesel", 0.01));  
a.accelerate(-10);
```

This raises the arithmetic exception we introduced before. In particular, we get the following output.

```
Exception in thread "main" java.lang.ArithmeticException: Negative values for speed  
are not allowed  
    at it.unive.dais.pol.vehicles.Vehicle.accelerate(Vehicle.java:27)  
    at it.unive.dais.pol.vehicles.autovehicles.Car.accelerate(Car.java:104)  
    at it.unive.dais.pol.vehicles.autovehicles.Car.main(Car.java:119)
```

Note that we passed a string to the constructor of the `ArithmeticException`. This string is the message of the exception, and it is printed when the program crashes in order to give more details about what problem happened.

14.3 Creating our own exceptions

Different classes represent different types of possible runtime errors. We want now to define our own types of error. In particular, in the example above we want to represent the situation when a negative speed is passed as parameter through a `NegativeSpeedException`. The constructor of this exception receives a double value with the negative speed value, and initializes the exception with a message that specifies what error occurred reporting also the negative value.

```
class NegativeSpeedException extends Exception {  
    public NegativeSpeedException(double acceleration) {  
        super("You passed a negative speed here. Its value was "+acceleration);  
    }  
}
```

We can now modify the implementation of `accelerate` in order to throw this exception.

```
class Vehicle {  
    public void accelerate(double a) {  
        if(a >= 0)
```

```
        this.speed += a;  
    else throw new NegativeSpeedException(a);  
}  
}
```

14.4 Declaring and documenting thrown exceptions

Unfortunately, the method above does not compile. In particular, the Java compiler states that we have an “unhandled exception: NegativeSpeedException”. In fact, all methods must explicitly declare the exceptions they throw. This can be done by adding a throws clause after the signature of the method and before its body. Therefore, we need to rewrite the method above as follows.

```
class Vehicle {  
    public void accelerate(double a) throws NegativeSpeedException {  
        if(a >= 0)  
            this.speed += a;  
        else throw new NegativeSpeedException(a);  
    }  
}
```

Note that once a method declares to throw an exception, then all methods invoking it will need to declare the same (e.g., method `accelerate` in `Car` since it calls `super.accelerate`). Therefore, adding exceptions to the method definition is often tedious and requires modifying quite a lot of method definitions.

Now that we added the thrown exceptions in the method definition, we need to document it. Javadoc allows to document those exceptions through the `@throws` clause³⁵ in methods’ documentation. This clause should be followed by an explanation about when the exception might be thrown. For instance, we can document the method above as follows.

```
class Vehicle {  
    /**  
     * Accelerate the car of the given amount of km/h.  
     * @throws NegativeSpeedException when the value passed as parameter is  
     *                               strictly less than zero  
     * @param a speed in km/h. Must be greater or equal than zero.  
     */  
    public void accelerate(double a) throws NegativeSpeedException {...}  
}
```

³⁵ Indeed, one might also use `@exception` that has exactly the same semantics of `@throws`. We prefer to use `@throws` since it is more coherent with the keywords used in the method definition.

14.5 Overriding methods throwing exceptions

Now that thrown exceptions are part of the method definition, we should take them into account when overriding methods. Let's recall the substitution principle: “if a class C1 exposes an interface that is wider than C2, then we can have instances of C1 wherever an instance of C2 is expected”. What do we mean by a wider interface when talking about exceptions? One might think about more exceptions, and therefore we might think that we can substitute a method m1 with a method m2 if m2 throws more exceptions than m1. Indeed, this is wrong. Let's imagine this scenario in terms of substitution, and let's assume m1 declares to throw a `FirstException`. With what methods could we substitute it? If a method m2 declares to throw `FirstException` and `SecondException`, it would expose more behaviors than the ones allowed by m1. Instead, if it declares to throw no exception, it will expose fewer behaviors, and therefore it might substitute m1.

Therefore, the general rule is: an overriding method can declare to throw at most all the exceptions of the overridden method. Unfortunately, this poses several problems: if I am extending a class and I want to throw more exceptions in a method I'm overriding, I cannot. I should be able to modify the superclass, but if it is part of a library I would not be able to do this. Generally speaking, we need to carefully consider exceptions (even if we might not need to throw them in our implementation) when we provide classes that are intended to be extended.

Let's go back once again to our vehicle example. This time we want to throw a `FuelNotSufficientException` when we accelerate a `Car` that does not have enough fuel to accelerate. This might be implemented as follows.

```
class FuelNotSufficientException extends Exception {
    public FuelNotSufficientException(double fuelConsumed, double fuel) {
        super("Not enough fuel: consumed "+fuelConsumed+", available "+fuel);
    }
}

class Car {
    public void accelerate(double amount)
        throws NegativeSpeedException, FuelNotSufficientException{
        double f = computeConsumedFuel(amount, fuelType.getLitresPerKmH());
        if(f < fuel) {
            super.accelerate(amount);
            fuel = fuel - fuelConsumed;
        }
        else
            throw new FuelNotSufficientException(fuelConsumed, fuel);
    }
}
```

```
}  
}
```

This code does not compile since method `accelerate` in class `Car` clashes with `accelerate` in class `Vehicle` but it cannot override it, since it declares more thrown exceptions. How can we solve this problem? A first idea might be to add `FuelNotSufficientException` in the exceptions thrown by `Vehicle.accelerate`. However, such a solution would imply that `Vehicle` has a notion of fuel (that we wanted to avoid since the beginning), and in general it might not be always possible (e.g., if `Vehicle` is in a library as discussed above).

A cleaner solution would be to articulate the exceptions type hierarchy in order to further extend it. In particular, we can define an abstract class `ImpossibleAccelerationException`, aimed at representing all the cases where it is not possible to accelerate, but not indicating a specific cause (and therefore abstract). Then method `Vehicle.accelerate` would declare that it throws this exception, and `NegativeSpeedException` and `FuelNotSufficientException` will extend both `ImpossibleAccelerationException`. This can be implemented as follows.

```
abstract class ImpossibleAccelerationException extends Exception {  
    public ImpossibleAccelerationException(String s) {  
        super(s);  
    }  
}  
class NegativeSpeedException extends ImpossibleAccelerationException {  
    public NegativeSpeedException(double acceleration) {  
        super("You passed a negative speed here. Its value was "+acceleration);  
    }  
}  
class Vehicle {  
    public void accelerate(double a) throws ImpossibleAccelerationException {  
        if(a >= 0)  
            this.speed += a;  
        else throw new NegativeSpeedException(a);  
    }  
}
```

```
class FuelNotSufficientException extends ImpossibleAccelerationException {  
    public FuelNotSufficientException(double fuelConsumed, double fuel) {  
        super("Not enough fuel: consumed "+fuelConsumed+", available "+fuel);  
    }  
}  
class Car {  
    public void accelerate(double amount)  
        throws ImpossibleAccelerationException {
```



```
double f = computeConsumedFuel(amount, fuelType.getLitresPerKmH());  
if(f < fuel) {  
    super.accelerate(amount);  
    fuel = fuel - fuelConsumed;  
}  
else  
    throw new FuelNotSufficientException(fuelConsumed, fuel);  
}
```

Note that the constructor in `ImpossibleAccelerationException` is needed to dispatch the message to the superclass.

14.6 Catching exceptions

We might want not only to throw an exception (to represent that some error occurred), but also to manage the situations where an exception has been thrown. For instance, method `race` might want to go ahead with a race even if one of the two vehicles threw a `ImpossibleAccelerationException`, since this represents a situation where the vehicle failed to speed up, but this would be fine during a race (let's imagine it as something like a mechanical issue). We can catch exceptions through the try-catch statement. This statement is structured as follow:

```
try {  
    <try-block>  
}  
catch(<exception-type> e) {  
    <catch-block>  
}
```

First of all, <try-block> is executed. Then we have three cases: (i) the block terminates normally, and this ends the execution of the whole try-catch statement, (ii) the block throws an exception that is subtype of <exception-type>, and then the <catch-block> is executed and the statement ends, or (iii) the block throws an exception that is not a subtype of <exception-type>, and then the exception is propagated to the outer level. Using this statement, we can rewrite the `race` method as follows.

```
public class Race<T> {  
    public T race(double length) {  
        v1.fullBrake();  
        v2.fullBrake();  
        double distanceV1 = 0, distanceV2=0;  
        while(true) {  
            distanceV1 += v1.getSpeed();  
            distanceV2 += v2.getSpeed();  
        }  
    }  
}
```

```
if(distanceV1 >= length || distanceV2 >= length) {  
    if(distanceV1 > distanceV2) return v1;  
    else return v2;  
}  
try {  
    v1.accelerate(Math.random()*10.0);  
    v2.accelerate(Math.random()*10.0);  
    catch(ImpossibleAccelerationException e) {  
        System.err.println("Problem when accelerating:"+e.getMessage());  
    }  
}  
}
```

Note that in this way the exception is caught and not propagated, and method race does not need to declare to throw an `ImpossibleAccelerationException`. In addition, the caught exception can be manipulated inside the catch block: the catch clause, in addition to the caught type, provides the name of the local variable that will point to the caught exception. In this way we can, for instance, get the message of the thrown exception and print it in the standard error.

14.7 Finally

A try-catch block can also be followed by a finally clause. This clause is always executed, (aka, both in case the code contained in the try block throws an exception that is caught, that it is not caught, or it does not throw any exception). After the execution of the code contained in the finally block, the execution proceeds in its previous state (that is, normal execution if there was no propagating exception, or it propagates the exception). Therefore, the syntax of a try-catch-finally block is as follows.

```
try {  
    <try-block>  
}  
catch(<exception-type> e) {  
    <catch-block>  
}  
finally {  
    <finally-block>  
}
```

Note that the try block needs to be followed by at least one catch block, or by a finally block, but it does not need to have both. The semantics of the overall block can be described as follows:

- 1) Execute the try block
- 2) If it ended without any exception, execute the finally block
- 3) If it threw an exception that is not caught, then execute the finally block and re-throw the exception
- 4) If it threw a caught exception, execute the catch block, and then the finally block (and in case the catch block throw an exception, re-throw the exception)

Let's go back again to our race method. We want now to always stop the car at the end of the race, both in case it ended normally or with an exception. We can encode this as follows.

```
public class Race<T> {  
    public T race(double length) {  
        v1.fullBrake();  
        v2.fullBrake();  
        double distanceV1 = 0, distanceV2=0;  
        try {  
            while(true) {  
                distanceV1 += v1.getSpeed();  
                distanceV2 += v2.getSpeed();  
                if(distanceV1 >= length || distanceV2 >= length) {  
                    if(distanceV1 > distanceV2) return v1;  
                    else return v2;  
                }  
                v1.accelerate(Math.random()*10.0);  
                v2.accelerate(Math.random()*10.0);  
            }  
        }  
        catch(ImpossibleAccelerationException e) {  
            throw new IllegalArgumentException("Random returned a negative number, e);  
        }  
        finally {  
            v1.fullBrake();  
            v2.fullBrake();  
        }  
    }  
}
```

The fullBrake method will be executed on the two vehicles whatever it happens, that is, if the race ended normally, or with an exception (re-thrown by the catch block).

14.8 Checked and unchecked exceptions

One might have noticed that in the previous snippet of code we throw explicitly an

exception (`IllegalArgumentException`³⁶) without declaring that the method throws it. How is this possible? Briefly, there are two types of exceptions in Java: the checked exceptions that need to be explicitly declared by the methods that might throw them, and the unchecked exceptions. Going back to the exception type hierarchy we reported in section 14.1, the unchecked exceptions are all subclasses of `RuntimeException` and `Error`, while the checked exceptions are all subclasses of `Exception` but not of `RuntimeException`.

We have already seen many examples of unchecked exceptions: `NullPointerException`, `ArithmeticException`, etc.. The idea is that those exceptions “can be thrown during the normal operation of the Java Virtual Machine”³⁷. This means that they do not represent a logic error inside the flow of the program, but that they are caused by some inconsistent operations (such as dividing a number by zero, accessing a null pointer or accessing an array outside its bounds) performed by the program. Why are they not declared? Essentially, the most part of code might throw those exceptions: any heap dereference might dereference a null value, any division might cause a division by zero, and any array access might access the array outside its bound. Declaring all those exceptions would be quite verbose and anyway pretty useless. Therefore, Java decided to introduce unchecked exceptions.

However, please always keep in mind that those exceptions should be used only in very specific cases. It would be tempting, for instance, to declare that our `ImpossibleAccelerationException` is a subclass of `RuntimeException` and in this way avoid declaring that quite a lot of methods might throw it. This should never be done (and even considered): this exception represents a logical error in the usage of our library, and therefore it should be checked and documented. Instead, note that in case of method `race` the situation is different: we are sure that the speed of vehicles is always greater than or equal to zero, and we pass a value that multiplies a positive number (10) with a random number that, by specification, is between zero and one. Therefore, if we get such an exception using method `accelerate` with that value, this means that `Math.random` returned a negative number, and this is an error that happened during the normal execution of the JVM. Therefore, we are allowed to throw an unchecked exception to stop the normal execution of the program.

³⁶<https://docs.oracle.com/en/java/javase/11/docs/api/java.base/java/lang/IllegalArgumentException.html>³⁷ <https://docs.oracle.com/en/java/javase/11/docs/api/java.base/java/lang/RuntimeException.html>

14.9 Errors

Apart from `RuntimeExceptions`, there is another type of exceptions that are unchecked: `Error`³⁸. `Error` “indicates serious problems that a reasonable application should not try to catch. Most such errors are abnormal conditions.” Intuitively, when a subclass of `Error` is thrown, this means that the runtime environment is definitely compromised, and we should not attempt to continue the execution of the program. For instance, if the program goes out of memory when allocating new objects, an `OutOfMemoryError`³⁹ is thrown. At that point, the runtime environment might be extremely slow, and the memory in a corrupted state. Similarly, a `StackOverflowError`⁴⁰ is thrown when a program recurses too deeply, that means, the call stack exceeds the maximal number of nested stacks that can be allocated by the JVM⁴¹.

Errors are unchecked for the same reason of `RuntimeExceptions`: they might be thrown by most existing methods, and there is no interest in declaring and/or documenting them. In addition, they should never be explicitly thrown by our program, since they indicate problems that are outside the logical and the operation of our code.

14.10 Chains of exceptions

When we throw the `IllegalArgumentException` in section 14.8, we passed the `ImpossibleAccelerationException` to the constructor.

```
public class Race<T> {  
    public T race(double length) {  
        ...  
        try {...}  
        catch (ImpossibleAccelerationException e) {  
            throw new IllegalArgumentException("Random returned a negative number, e);  
        }  
        finally {...}  
    }  
}
```

In particular, class `Throwable` provides constructors that receive an exception that represents the cause of the exception we are building up. In this way, we can chain

³⁸ <https://docs.oracle.com/en/java/javase/11/docs/api/java.base/java/lang/Error.html>

³⁹ <https://docs.oracle.com/en/java/javase/11/docs/api/java.base/java/lang/OutOfMemoryError.html>

⁴⁰ <https://docs.oracle.com/en/java/javase/11/docs/api/java.base/java/lang/StackOverflowError.html>

⁴¹ The number of nested stack can be specified to the JVM through the `-Xss` option

together several exceptions, in order to provide explanation not only at the outer level (random returned a negative number), but also internally (we discovered it because we passed a negative value to the accelerate method). In the example above, the throw of that exception would lead to the following output.

```
Exception in thread "main" java.lang.IllegalArgumentException: Random returned a
negative value
    at it.unive.dais.pol.vehicles.Race.race(Race.java:114)
    at it.unive.dais.pol.vehicles.Race.main(Race.java:125)
Caused by: it.unive.dais.pol.vehicles.NegativeSpeedException: You passed a negative
speed here. Its value was -2.83246176723153
    at it.unive.dais.pol.vehicles.Vehicle.accelerate(Vehicle.java:28)
    at it.unive.dais.pol.vehicles.Race.race(Race.java:111)
    ... 1 more
```

We can see that first of all we have the “external” explanation, followed by the internal details. In this way, a user of the Race.race method would focus on the error about Random (and then maybe double check what went wrong in the JVM and the Java libraries), while the developer of Race.race would go into the internal implementation of the method to see if effectively this was the cause of the problem.

14.11 Assertions

Assertions are special Java statements that allow to check some testing conditions. In particular, Java provides the assert statement. This is followed by a Boolean condition, and optionally by a string message. When executing the assert statement, the Boolean condition is checked. If it is evaluated to false, then an AssertionError⁴² (with the optional message in case) is thrown. For instance, we might want to avoid that somebody can invoke accelerate passing a negative value using the assert statement.

```
class Vehicle {
    public void accelerate(double a) {
        assert a >= 0 : "The speed should be always positive"
        this.speed += a;
    }
}
```

When invoking accelerate with passing a negative value, we get the following output.

⁴² <https://docs.oracle.com/en/java/javase/11/docs/api/java.base/java/lang/AssertionError.html>

```
Exception in thread "main" java.lang.AssertionError: The speed should be always positive
    at it.unive.dais.pol.vehicles.Vehicle.accelerate(Vehicle.java:23)
    at it.unive.dais.pol.vehicles.Race.main(Race.java:125)
```

Did you try to run that? If so, you would have discovered that you didn't get the exception. Why? The idea is that assertions are checked only when testing a program. Therefore, by default assertions are disabled. In order to check them, we need to pass to java the option `-ea` (enable assertions). This also explains why when failing an assert statement returns an `Error` and not an (unchecked) `RuntimeException`: if we are in a testing environment and an assertion fails, we want to definitely stop the execution of our program in order to debug it. Raising an `Error` will prevent anybody (if following the Java guidelines) to catch and manage it, and therefore it will make the program to crash

However, since we cannot be sure assertions will be checked in our code, we need always rely on the worst case scenario when developing our methods. This means that, for instance, `accelerate` should still consider the case when the speed is negative. Therefore, `accelerate` can be implemented as follows.

```
class Vehicle {
    public void accelerate(double a) throws ImpossibleAccelerationException {
        assert a >= 0 : "The speed should be always positive"
        if(a >= 0)
            this.speed += a;
        else throw new NegativeSpeedException(a);
    }
}
```

14.12 Exercise

Given the following code, create an ad hoc exception and throw it when ever a registered vehicle ID is instantiated with an invalid license plate. Handle these exceptions in the main method.

```
package it.unive.dais.pol.id;

public class Main {

    public static void main (String[] args) {

        OptionParser p = new OptionParser(args);

        ...

        String make = p.getMake();
        String model = p.getModel();
        String licensePlate = p.getLicensePlate();
        String VIN = p.getVIN();

        CarID carId = new CarID(make, model, licensePlate, VIN);

        try {
            CarInfo car = retrieveCarInfoFromDatabase(id);
            ...
        } catch (SQLException e){
            ...
        }
    }

    ...
}
```



```
package it.unive.dais.pol.id;

public final class CarID extends RegisteredVehicleID {

    public CarID(String make, String model, String licensePlate, String VIN) {
        super(make, model, licensePlate, VIN);
    }

    protected final boolean isValidLicensePlate(){

        if(licensePlate != null && licensePlate.length() != 7)
            return false;

        if(!Character.isDigit(licensePlate.charAt(2))
            || !Character.isDigit(licensePlate.charAt(3))
            || !Character.isDigit(licensePlate.charAt(4)))
            return false;

        if(!Character.isUpperCase(0))
            || ! Character.isUpperCase(licensePlate.charAt(1))
            || !Character.isUpperCase(licensePlate.charAt(5))
            || !Character.isUpperCase(licensePlate.charAt(6)))
            return false;

        if(!licensePlate.contains("I")
            && !licensePlate.contains("O")
            && !licensePlate.contains("EE"))
            return false;

        return true;
    }

    ...
}
```

```
package it.unive.dais.pol.id;

public abstract class RegisteredVehicleID {

    protected final String make;
    protected final String model;
    protected String licensePlate;
    protected final String VIN;

    public RegisteredVehicleID (String make, String model,
                                String licensePlate, String VIN) {

        this.make = make;
        this.model = model;
        this.licensePlate = licensePlate;
        this.VIN = VIN;
    }

    protected abstract boolean isValidLicensePlate();
    ...
}
```

Chapter 15: Annotations

We have several ways to add information on a single component (e.g., class, field, method) of our OO programs. First of all, we have a set of given modifiers that we can apply to specify, for instance, the visibility of a component (i.e., public, protected, default or private). We can also add comments. If we follow the Javadoc standard, these comments are compiled into the documentation of our software. Finally, we might express what our code expects and guarantees through formal contracts.

In Java, we can also adopt annotations. Intuitively, annotations are similar to classes. In particular, they can have a state through some attributes, and they can be implemented and extended through a standard syntax. In addition, annotations can be read and managed by the compiler and the JVM. This means that this information is potentially available during both compilation and execution (while instead comments are discarded by the compiler). Annotations nowadays are a key component of Java programs and they are adopted by most existing Java software.

15.1 Syntax

Annotations can be attached to all the major components of our OO program. In this chapter, we will focus on fields, methods, and method parameters, but annotations can be applied also to classes, generics, etc.. An annotation always starts with a character @ followed by the name of the annotation. The annotation must always precede the component it annotates. Consider for instance the following code auto-generated by common IDEs when one overrides method hashCode.

```
public class Vehicle {  
    @Override  
    public int hashCode() {...}  
}
```

As we discussed in Chapter 13, class Vehicle (implicitly) extends Object inheriting method hashCode. Therefore, the implementation of hashCode in Vehicle overrides the one in Object. This method is annotated with Override in order to indicate this.

15.2 @Override, @Deprecated, and @SuppressWarnings

What's the purpose of annotating method `hashCode` in this way? For sure this provides more information about our implementation since it explicitly states that we are aiming at overriding a method. However, this could have been achieved with some comments. The advantage of using the `@Override` annotation⁴³ is that we can make this information available to the compiler and/or the runtime environment. In particular, compilers are required to raise an error if the annotated method does not override anything. For instance, let's imagine that, by mistake, we write the following code.

```
public class Vehicle {  
    @Override  
    public int hashCode(int i) {...}  
}
```

This `hashCode` method does not override any method, since its signature is different from the one in `Object`. In this case, the Java compiler raises an error stating that the “method does not override method from its superclass”. If we had not annotated the method, we would have not got any error without discovering a bug in our code.

Another annotation widely used in methods is `Deprecated`⁴⁴. In section 12.1 we have already discussed one of its applications, in particular in class `Object` where it states that method `finalize` has been deprecated, and it should not be used anymore. The aim of `@Deprecated` is to indicate an element that “programmers are discouraged from using”. For instance, we defined a method `refuel` in `Car` that received only a `double` value. Later, we introduced the class `FuelTank` and a method `refuel` using such objects, in order to check the type of the fuel and not only the quantity. Now we would like to remove the first implementation of this method. However, this method was public, and if we remove it the code using our library and calling this method would not compile anymore. Therefore, usually instead of removing we first tag the method as deprecated through this annotation as follows.

```
public class Car extends Vehicle {  
    @Deprecated  
    public void refuel(double amount) {  
        fuel += amount;  
    }  
}
```

In this way, the code calling `Car.refuel(double)` will still compile. However, the compiler sees this annotation, and it raises a warning when it is called stating that “`refuel(double)` is deprecated”.

⁴³ <https://docs.oracle.com/en/java/javase/11/docs/api/java.base/java/lang/Override.html>

⁴⁴ <https://docs.oracle.com/en/java/javase/11/docs/api/java.base/java/lang/Deprecated.html>

Another quite popular annotation is `SuppressWarnings`⁴⁵. The Java compiler produces a lot of different types of warnings. However, we might be interested only in a few of them, and not in all the code. First of all, we need to set up our compiler properly in order to focus only on the types of warnings that matter for us. However, we might still have warnings that in general are interesting, but not in some specific portions of code. `@SuppressWarnings` state that some warnings should not be produced on some OO components (usually, methods). This annotation can also receive an array of strings specifying what type of warnings we want to suppress. For instance, class `Car` contains a public method `getFuelCost` that is not used by our own code. The Java compiler produces a warning stating that “Method 'getFuelCost()' is never used”. We believe this warning is useless since we plan this method to be an interface of our library, and therefore we suppress it on this specific method as follows.

```
public class Car extends Vehicle {  
    @SuppressWarnings("unused")  
    public double getFuelCost() {  
        return this.fuelType.getFuelCost();  
    }  
}
```

In particular, this annotation specifies to suppress all warnings concerning unused components on method `getFuelCost`. Another annotation that would have suppressed that warning is `@SuppressWarnings("all")`. In this way, we suppress warnings of any kind on the annotated method.

15.3 Defining our own annotations

So far we have seen how we can use existing annotations. However, we can also define our own annotations with attributes. For instance, let's imagine that we want to define an annotation that specifies if a given field, parameter, or returned value is a speed. Such annotation should also specify the direction of the speed (forward or backward) and the unit of measure (kmh, kph, ms, ...). This can be implemented as follows in a .java file.

```
public @interface Speed {  
    String type() default "kmh";  
    boolean forward();  
}
```

Like interfaces and classes, the declaration starts with the visibility (public) with a specific keyword (`@interface`). Then, we can specify a list of attributes. Each attribute has

⁴⁵ <https://docs.oracle.com/en/java/javase/11/docs/api/java.base/java/lang/SuppressWarnings.html>

a static type, a name (followed by parentheses) and it can have a default value. The annotation can be then used on any OO component. For instance, we can annotate class Vehicle as follows.

```
public abstract class Vehicle {  
    @Speed(forward = true) private double speed;  
    public void accelerate(@Speed(forward = true) double a) {...}  
    @Speed(forward = true, type = "kph") public double getSpeed() {...}  
}
```

First of all, all annotations must specify the value of the attribute forward. This is required since we did not specify any default value. Instead, attribute type has a default value (kmh), so when it is not defined it assumes such value. In order to specify attribute values, we need to specify the name of the attribute followed by character ‘=’ and the desired value among parentheses after the name of the annotation.

The code snippet above annotates (i) field speed with a forward speed in kmh, (ii) the parameter of method accelerate with a forward speed in kmh, and (iii) method getSpeed (aka, the value returned by the method) with a forward speed in kph.

15.4 Target

We want to apply annotation Speed only on fields, methods and method parameters, and to prevent it from being applied to other components (e.g., classes). We can annotate our annotation with @Target⁴⁶ in order to specify to what components it is applicable. In particular, @Target has an attribute value⁴⁷ that consists of an array of ElementType⁴⁸ representing the different OO components. For instance, annotation Speed can be annotated as follows.

```
@Target( {  
    ElementType.METHOD,  
    ElementType.FIELD,  
    ElementType.PARAMETER  
})  
public @interface Speed {...}
```

In this way, if we try to annotate a class with Speed the compiler raises an error stating that “‘@Speed’ not applicable to type”.

⁴⁶ <https://docs.oracle.com/en/java/javase/11/docs/api/java.base/java/lang/annotation/Target.html>

⁴⁷ In this way, we can pass the value for the attribute without the need of explicitly referring to the name of the attribute as we did for Speed

⁴⁸ <https://docs.oracle.com/en/java/javase/11/docs/api/java.base/java/lang/annotation/ElementType.html>

15.5 Retention

Annotations might be interesting only in source code (e.g., like comments), in the .class files (e.g., to discover that we are using a deprecated method in a library), or also at runtime. Annotations can be annotated with `@Retention`⁴⁹ to specify this. In particular, `@Retention` receive a single `RetentionPolicy`⁵⁰ value that indicates that the annotation should be retained only in the source code (`RetentionPolicy.SOURCE`), also in the .class files (`RetentionPolicy.CLASS`), or at runtime as well (`RetentionPolicy.RUNTIME`). For the purposes of annotation Speed, we can annotate it to be retained only in the source code as follows.

```
@Retention(RetentionPolicy.SOURCE)
public @interface Speed {...}
```

15.6 JUnit

We now briefly discuss a couple of examples of libraries/frameworks that rely on annotations. We start from JUnit⁵¹, a testing framework for Java. Intuitively, a test is a small method that runs some portions of our code in order to check if it behaves correctly (i.e., it does not throw exceptions and it computes the expected results). The goal of tests is to have an automatic means to check if our software is correct each time we modify, compile, release, deploy, ... it. Without entering into the details of testing, we introduce how JUnit supports tests. In particular, JUnit provides a `@Test` annotation⁵² that can be applied to methods. This indicates that a non-static, non-private method that does not return any value. For instance, we can develop the following test for our Car. Its main goal is to check that when we accelerate and we have enough fuel, we effectively speed up the car of the correct amount.

```
public class CarTester {
    @Test public void testAccelerate() throws ImpossibleAccelerationException {
        Car myCar = new Car(0, new FuelType("diesel", 0.015, 0.01));
        myCar.refuel(10.0);
        myCar.accelerate(10.0);
        assert myCar.getSpeed() == 10.0;
    }
}
```

⁴⁹

<https://docs.oracle.com/en/java/javase/11/docs/api/java.base/java/lang/annotation/Retention.html>

⁵⁰

<https://docs.oracle.com/en/java/javase/11/docs/api/java.base/java/lang/annotation/RetentionPolicy.html>

⁵¹ <https://junit.org/>

⁵² <https://junit.org/junit5/docs/5.0.0/api/org/junit/jupiter/api/Test.html>

```
}  
}
```

JUnit is a framework. This means that JUnit requires to run a Java program with its own runtime environment, and not with the standard Java one (whose execution starts from a main method). The JUnit runtime environment extracts all the methods that are annotated with `@Test` (and in fact the retention of this annotation is `RUNTIME`), executes all of them, and reports if the tests were successful or not.

15.7 JAXB

The Jakarta XML Binding (JAXB)⁵³ is a library that allows mapping Java classes to XML files. In this way, we can dump the state of Java objects to XML files, and load XML files as Java objects. JAXB heavily relies on annotations in order to allow a developer to specify what he/she wants to represent of a class in an XML file. In particular, we have (i) `@XmlType` to annotate classes that we want to represent in XML files, (ii) `@XmlRootElement` to annotate classes that can be the root of XML files, (iii) `@XmlElement` to annotate fields or getters/setters that we want to represent as XML elements, and (iv) `@XmlAttribute` to annotate fields or getters/setters that we want to represent as XML attributes. For instance, we can annotate class `FuelType` as follows.

```
@XmlRootElement  
@XmlType  
public class FuelType {  
    @XmlElement private String type;  
    @XmlAttribute private double costPerLiter;  
    @XmlAttribute private double litresPerKmH;  
}
```

Once annotated, objects instances of `FuelType` can be easily marshalled to and unmarshalled from XML files as follows.

```
static void marshal(FuelType fuelType) throws JAXBException {  
    JAXBContext context = JAXBContext.newInstance(FuelType.class);  
    Marshaller mar= context.createMarshaller();  
    mar.marshal(fuelType, new File("./fuelType.xml"));  
}  
static FuelType unmarshall() throws JAXBException, IOException {  
    JAXBContext context = JAXBContext.newInstance(FuelType.class);  
    return (FuelType) context.createUnmarshaller().  
        unmarshal(new FileReader("./fuelType.xml"));  
}
```

⁵³ JAXB was part of the Java standard library until Java 8. Starting from Java 9, one needs to add to the classpath an API and an implementation library in order to use it.

However, the unmarshalling causes an exception since class `FuelType` does not have a public constructor without parameters. This happens because JAXB needs to create an instance of the class through reflection (see next chapter). Once implemented, the class is correctly marshalled to and unmarshalled from the following XML file.

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<fuelType costPerLiter="0.015" litresPerKmH="0.01">
  <type>diesel</type>
</fuelType>
```

15.8 Exercise

Implement a new annotation *CarID* with the string fields *model* and *make*, where the value are set as “none” by default. The annotation must be applicable only on methods and must be readable at runtime. Then, create a class that use the annotation *CarID*.

Chapter 16: Reflection

How is it possible that JUnit executes a method whose signature was unknown to him? And how can JAXB create a file assigning arbitrary values to private fields? So far the only way to read and write fields, and invoke methods was through explicit statements in our code. The Java static type system and compiler guarantees that all these statements are well formed (i.e., fields and methods exist and they are accessible). While the program might still crash during the execution for many different good reasons (from logical errors in the executed code to irrecoverable issues in the JVM), the compiler guarantees us that the components our code accesses exist.

However, there is another way to access those components, that is, by reflection. Reflection is the ability of a program to access programmatically (aka, through code) information about its structure, such as classes, fields, methods and constructors. This ability is needed to “accommodate applications such as debuggers, interpreters, object inspectors, class browsers, and services such as Object Serialization and JavaBeans that need access to either the public members of a target object (based on its runtime class) or the members declared by a given class”⁵⁴. Indeed, nowadays it is widely used by frameworks (aka, libraries that provide a specific execution model) such as JUnit in order to decide what components execute and how.

16.1 Class

The main entrypoint of reflection is the `java.lang.Class` class⁵⁵. This class is aimed at representing all the types of our programs. There are two main ways to get an instance of `Class`: (i) through `Object.getClass`, and (ii) through `<type_name>.class`. In the first way, we can have the `Class` representing the concrete type of an object. However, if some types are not instantiable (i.e., primitive types, interfaces, abstract classes, etc.), or if we want

⁵⁴

<https://docs.oracle.com/en/java/javase/11/docs/api/java.base/java/lang/reflect/package-summary.html>

⁵⁵ <https://docs.oracle.com/en/java/javase/11/docs/api/java.base/java/lang/Class.html>

to get the Class object representing a known type, we can rely on `<type_name>`. Being able to get a Class object for any type of our program is essential in order to fully exploit the possibilities of reflection (e.g., to specify the types of parameters of a method we are looking at in a class).

Class `class` provides several methods to access all the information available about the class:

- `isPrimitive`, `isInterface`, `isAnnotation`, `getModifiers`, etc.. provide information about the definition of the type (such as if it is a primitive type, an interface, an annotation, and its modifiers⁵⁶)
- `getInterfaces`, `getSuperclass`, `getPackage`, etc.. provide information about the type hierarchy (the interfaces implemented by the current class, its superclass, and the package it belongs to, respectively)
- `getFields`, `getMethods` and `getConstructors` return all the accessible fields, methods, and constructors. Those might be declared in the represented type, or inherited, but they are public and accessible from the current context
- `getDeclaredFields`, `getDeclaredMethods`, and `getDeclaredConstructors` return all the fields, methods and constructors that are defined in the represented types. Those might not be accessible (e.g., private) and they must be declared exactly in the given type (and not inherited)

```
Class<Vehicle> c = Vehicle.class;
for(Constructor t : c.getDeclaredConstructors())
    System.out.println(t);
for(Method m : c.getDeclaredMethods())
    System.out.println(m);
for(Field f : c.getFields())
    System.out.println(f);
for(Field f : c.getDeclaredFields())
    System.out.println(f);
System.out.println(c.getSuperclass());
System.out.println(c.getPackage());
```

For instance, the code above will (i) print the only constructors we defined (public and with one double parameter) in `Vehicle`, (ii) print `accelerate`, `fullBrake`, `brake`, and `getSpeed` (all the methods declared in `Vehicle`), (iii) print nothing (there is no accessible

⁵⁶ `getModifiers` return an int value representing all the modifiers of the current type. Class `Modifier` (<https://docs.oracle.com/en/java/javase/11/docs/api/java.base/java/lang/reflect/Modifier.html>) contains several utilities to query this value to know what modifiers it represents.

field in class Vehicle), (iv) print the private field speed, (v) print Object (implicit superclass of Vehicle), and (vi) print the package where Vehicle is (it.unive.dais.po1.vehicles).

Indeed, point 4 prints also another private static field, that is, `Vehicle.$assertionsEnabled`. From where did this field pop out? This field was added by the Java compiler when we added an assertion to our Vehicle class (to check if the value passed to `accelerate` was greater than or equal to zero). This is used in order to check if the assertions are enabled through the `-ea` option to the JVM (and therefore they should be checked) or not. When we inspect our classes through reflection, we also see all the instrumentations the compiler added to our code. Field `$assertionsEnabled` (and in general any field starting with `$`) is one of these instrumentations.

Therefore, `Class` provides us means to access all information and components of a class. We can also look for a specific field, method or constructor by invoking methods `getField`, `getMethod`, and `getConstructor` and passing the desired field name or method signature.

16.2 Field

We now discuss the main class components we can inspect starting from fields. Generally speaking, we can read or write fields. Therefore, `Field` class⁵⁷ provides methods to get the value of a field, and to set its value. How do we programmatically read and write fields? For instance, `Vehicle.getSpeed` returns `this.speed`, while `accelerate` assigns `this.field=...` In both cases, the Java static type checker guarantees that the value read and written is a `double`. Instead, class `Field` supports the read and write of any type of value. Therefore, its interface provides methods to read and write values of all kinds. We already discussed that `Object` is a superclass for all reference types. Unfortunately, primitive types are outside this type hierarchy. So `Field` provides a `get` and a `set` method for each primitive type.

Generally speaking, a `get<type>` method receives an object (that is, the receiver of the field read), and it returns a `<type>` value (that is, the value contained in the field). Similarly, `set<type>` receives an object (the receiver of the write), and a `<type>` value (to be written in the field). For instance, we might set field speed of a `Vehicle` as follows.

⁵⁷ <https://docs.oracle.com/en/java/javase/11/docs/api/java.base/java/lang/reflect/Field.html>

```
Car c = new Car(0, null);  
Class classCar = c.getClass();  
Class classVehicle = classCar.getSuperclass();  
Field speed = classVehicle.getDeclaredField( "speed");  
speed.setDouble(c, 10.0);
```

In order to retrieve a `Field` object that represents a field of a class, we invoke method `get[Declared]Field` passing the name of the field we are looking for. First of all, note that we can ask the class about any field, even one that does not exist. In this case, method `getDeclaredField` throws a `NoSuchFieldException`. Similarly, we might get and set values with arbitrary types (e.g. try a `setBoolean` on field `speed`). In this case, the method will throw an `IllegalArgumentException`. In addition, the field might not be accessible. Indeed, the code above raises an `IllegalAccessException` since field `speed` is private. When we accessed the field through code, none of these scenarios was possible, since the compiler prevented us from accessing a field that does not exist, or reading and assigning values of the wrong type. With reflection, we take a more dynamic approach, where all these constraints are checked at runtime and therefore they do not raise a compiler error, but they raise a runtime exception.

Is there any way to set an arbitrary value to field `speed`? When we unmarshalled an xml to a `FuelType` instance, the private fields of that object were set to arbitrary values. `Field` class (as well as `Method` and `Constructor`) provide a method `setAccessible` that receives a `Boolean` value⁵⁸. If we invoke `speed.setAccessible(true)`, we can then execute method `setDouble` without raising an `IllegalAccessException`, thus modifying the value stored in the field. However, in this way we completely break the concept of encapsulation: anybody from outside might read and inject arbitrary values into our fields, and we may not rely on any assumption about what they can do with our object. Clearly, reflection breaks several constraints that Java imposes through its compiler and strong static typing. Practice showed that there are many contexts (JAXB being just one notable example) where such possibility is desirable. Anyway, we should always keep in mind that when we perform such aggressive operations we might be breaking completely the invariants of the objects we deal with.

16.3 Method

⁵⁸ Indeed, this method works only if some constraints about modules' structure are respected. Since we do not discuss modules in this book, we omit this discussion. Please refer to the documentation at [https://docs.oracle.com/en/java/javase/11/docs/api/java.base/java/lang/reflect/AccessibleObject.html#setAccessible\(boolean\)](https://docs.oracle.com/en/java/javase/11/docs/api/java.base/java/lang/reflect/AccessibleObject.html#setAccessible(boolean)) for the full details.

The Method class has a structure quite similar to Field. In particular, it contains all methods to get information about the OO component, it can be retrieved from the containing class, and it might be used to access the OO component. However, methods are very different from fields: while we can read and write in the latter case, in the first case we can invoke the component and get back the results of the computation. Therefore, how we access the component is rather different.

First of all, when we invoke a method we need to specify the method signature, that is, the method name, and the number and static types of parameters. Class provides a method `getMethod(String name, Class... parameterTypes)` (and an identical `getDeclaredMethod`) that given the name of the method, and a sequence of Class instances (representing the types of the parameters) returns a Method instance representing the method with the given signature. Its behaviour is identical to `get[Declared]Field`: if the given OO component does not exist, it throws a `NoSuchMethodException`,

Once we get the Method instance, we can execute the method by invoking `Method.invoke(Object obj, Object... args)`. This method receives the receiver object of the invocation (like for fields), a sequence of objects (the parameters), and returns an object (the value returned by the method). Note that everything goes through objects and we do not have distinct methods to pass/retrieve primitive values like we had for fields, and this approach was not possible since the sequence and types of parameters is unknown a-priori. Therefore, Java relies on the auto-boxing and unboxing of primitive values as described in section 13.5. `invoke` might throw three different exceptions: (i) `IllegalAccessException` (if we do not have access to the invoked methods like for fields, and we can circumvent this by invoking `setAccessible(true)`), (ii) `IllegalArgumentException` (if we pass a wrong number or type of parameters like when we passed a wrong value to `setField`), and (iii) `InvocationTargetException` (if the invoked method throws an exception - note that this exception will be chained inside the `InvocationTargetException`).

```
Car c = new Car(0, null);
Class classCar = c.getClass();
Class classVehicle = classCar.getSuperclass();
Method accelerate = classCar.getDeclaredMethod("accelerate", double.class);
Method getSpeed = classVehicle.getDeclaredMethod("getSpeed");
result = getSpeed.invoke(c);
Object result = accelerate.invoke(c, 2.0);
```

Consider, for instance, the code above. We retrieve two methods: (i) `getSpeed` in class

Vehicle, and (ii) `accelerate` in class `Car`. We then first invoke `getSpeed` on a fresh new instance of a car with speed zero. This invocation returns an instance of `Double` with a `0.0` value wrapped inside. The invocation of `accelerate` instead raises an `FuelNotSufficientException` exception, since we accelerate of `2.0` km/h but we did not previously refuel the `Car`. Therefore, the following exception is thrown by `accelerate.invoke(2.0)`.

```
Exception in thread "main" java.lang.reflect.InvocationTargetException
at sun.reflect.NativeMethodAccessorImpl.invoke0(Native Method)
at sun.reflect.NativeMethodAccessorImpl.invoke(NativeMethodAccessorImpl.java:62)
at sun.reflect.DelegatingMethodAccessorImpl.invoke(...)
at java.lang.reflect.Method.invoke(Method.java:498)
at it.unive.dais.pol.vehicles.Main.main(Main.java:155)
Caused by: it.unive.dais.pol.vehicles.autovehicles.fuel.FuelNotSufficientException: The
fuel was not enough: consumed 0.02, in the tank there was 0.0
at it.unive.dais.pol.vehicles.autovehicles.Car.accelerate(Car.java:112)
... 5 more
```

In this example the utility of having chained exceptions is clear: the outer exception simply reports the exception on the reflective call, while the inner one fully explains the reasons for this exception.

16.4 Constructor

Constructors are special methods that can be invoked only once when a class is instantiated. They do not have a name, and they are distinguished by their parameters. Therefore, the interface for retrieving and invoking constructors is slightly different from standard methods. First of all, `get[Declared]Constructor` receives only a sequence of Class instances (and not a name). Then we do not have an `invoke` method whose first parameter is the receiver of the call, but instead a `newInstance` method, that receives only a sequence of objects (the parameters' values) and returns a new instance of the class containing the constructor after its execution. This method returns a generic type `T`, that is, the generic representing the class that contains the constructor. All the rest (exceptions, access, etc..) is identical to methods. For instance, the following code creates a new car running at `2.0` km/h with diesel fuel.

```
Class<Car> classCar = c.getClass();
Constructor<Car> cst = classCar.getDeclaredConstructor(
    double.class, FuelType.class);
Car created = cst.newInstance(2.0, new FuelType("diesel", 0.015, 0.01));
```

16.5 Annotations

Reflection can be used to access annotations at runtime. Only annotations with retention `RUNTIME` (see section 15.5) can be seen by reflection. The interface `AnnotatedElement`⁵⁹ provides all the methods to access those annotations. In particular, given an `AnnotatedElement` object, we can check if it is annotated with a given annotation through `isAnnotationPresent`, retrieve all annotations through `getAnnotations` and `getDeclaredAnnotations` (with the same behavior we described for fields and methods), and get a specific annotation through `getAnnotation` and `getDeclaredAnnotation`. All the classes in `java.lang.reflect` that represent OO components (such as `Class`, `Field`, `Method`, and `Constructor`) implement this interface.

When an annotation is retrieved from an OO component, we get an object of type `Annotation`⁶⁰. This interface defines very few methods (it overrides few `Object` methods, and defines an `annotationType` method that returns the class representing the annotation type). What is represented by this interface? If we look into its documentation we see that all the annotations of the standard Java library we discussed in the previous chapter (`Deprecated`, `SuppressWarnings`, `Override`) are implementations of `Annotation`. Indeed, when we define an annotation (through the `@interface` keyword) the Java compiler creates a class (named as the defined annotation) that implements `Annotation`. In addition to the methods defined in the annotation, we find also the methods that retrieve the values of the attributes specified in the definition of the annotation. For instance, `Speed` contains method `forward()` (returning the Boolean value of attribute `forward`) and `type()` (returning a string that represents the unit of measure of the speed).

```
Class classVehicle = Vehicle.class;
Field s = classVehicle.getDeclaredField("speed");
boolean annotated = s.isAnnotationPresent(Speed.class);
Speed speedAnnotation = (Speed) s.getAnnotation(Speed.class);
String type = speedAnnotation.type();
boolean forward = speedAnnotation.forward();
```

Consider, for instance, the code above. First of all, we retrieve the field `speed` of the class `Vehicle`. We then check if it is annotated as `Speed`. If we keep the definition of `Speed` given in section 15.5, `isAnnotationPresent` will return `false`, and `getAnnotation` will return a null value. In fact, if the annotation is not retained at runtime, it is not visible by

⁵⁹

<https://docs.oracle.com/en/java/javase/11/docs/api/java.base/java/lang/reflect/AnnotatedElement.html>

⁶⁰

<https://docs.oracle.com/en/java/javase/11/docs/api/java.base/java/lang/annotation/Annotation.html>

reflection means. If we modify the retention to `RUNTIME`, then `isAnnotationPresent` returns true, and the retrieved annotation object has “kmh” for attribute type, and true for forward. This information can be then used for various purposes (e.g., to convert from a unit of measure to another one based on the annotation of the various components).

16.6 Pros and cons of reflection

Through reflection we can access components that are not accessible through plain Java code (where by plain we mean code not using reflection). For instance, we can read and write private fields, and invoke private methods. However, this breaks any principle of encapsulation: if somebody can access components that we wanted to keep invisible (and we might assume some invariants hold on these components), then we cannot modularly reason about our classes. Reflection should be conceived as an exceptional means to perform some operations in extreme cases. For instance, without reflection we might not marshall and unmarshall classes into xml files through JAXB. Nevertheless, we should reason about our code without taking into account what reflection can break, and assume that such means is used only by careful developers and for well delimited tasks.

If on the one hand reflections allow access to components not visible or that were unknown when the program was compiled, on the other hand it has several drawbacks. First of all, we do not have any static guarantee about what we are accessing: if a class, field, or method does not exist, the code still compiles, and exceptions are thrown at runtime in these cases. In addition, if we pass wrong values (e.g., we try to assign an int value to a String field), we get another exception. Generally speaking, reflection circumvents all the static checks of the Java compiler, and it relies on exceptions during the execution in case an access is not well-formed. In addition, accessing components through reflection is quite more verbose than doing this programmatically: we need to first retrieve the Class object, then the component, then use the correct method to access the components. Instead programmatically we can directly dereference an object and access its components.

16.7 An example using reflection

Let's now use reflection in a rather extensive way. In particular, we want to run a race among many different types of vehicles, potentially unknown when we write our code. Therefore, we want to use reflection in order to create those vehicles. First of all, we

need to implement a class that allows us to run a race among an indefinite number of vehicles. To do so, we extend `HashSet<Vehicle>` implementing a `race(double length)` method that runs the race as follows. In this way, we can use the interface of `Set` in order to add vehicles to our collection, and then simply invoke method `race` in order to run the race.

```
public class VehicleHashSet<T extends Vehicle> extends HashSet<T> {  
    public Vehicle race(double length) throws ImpossibleAccelerationException {  
        for(Vehicle v : this) {  
            v.fullBrake();  
        }  
        HashMap<Vehicle, Double> distance = new HashMap<>();  
        for(Vehicle v : this) {  
            distance.put(v, 0.0);  
        }  
        while(true) {  
            for(Vehicle v : this)  
                distance.put(v, distance.get(v)+v.getSpeed());  
            for(Vehicle v : this)  
                if(distance.get(v) >= length)  
                    return v;  
            for(Vehicle v : this)  
                v.accelerate(Math.random()*10.0);  
        }  
    }  
}
```

The code above is a slight variation of the `race` method among two vehicles we previously wrote. The main difference is that it needs to iterate over all the vehicles in the hashset, and we need a support structure (`distance`) to track how many kms each vehicle has already traveled. Anyway, for now nothing about reflection!

```
static Collection<Class> getAllVehiclesClasses() {  
    HashSet<Class> result = new HashSet<>();  
    result.add(Bicycle.class);  
    result.add(Car.class);  
    result.add(Truck.class);  
    result.add(HorseCart.class);  
    return result;  
}
```

First of all, we need a method that returns all the existing types of vehicles. For now we assume to have an oracle (such the one above) that returns a collection with the `Class` instances representing those vehicles. Later, we will see how we can use a library to retrieve them automatically.

Now that we have a collection with all the classes, we need to construct an instance of each class. We implement a method `createInstanceOf` that performs this operation. In particular, it traverses all the constructors of each class. For each parameter of the constructor, it tries to create an arbitrary value through method `getConcreteValue` (that given a `Class` returns an object instance of that class). If it succeeds to have values for all the parameters, it invokes the constructor through method `newInstance`, and it returns the constructed object. Otherwise, it iterates to the following constructor. If `createInstanceOf` was unable to find a suitable constructor, it returns a null value. These methods can be encoded as follows.

```
static Vehicle createInstanceOf(Class c)
    throws InvocationTargetException, InstantiationException, IllegalAccessException {
    external: for(Constructor constructor : c.getConstructors()) {
        constructor.setAccessible(true);
        Parameter[] parameters = constructor.getParameters();
        Object[] parametersValues = new Object[parameters.length];
        for(int i = 0; i < parameters.length; i++) {
            Object o = getConcreteValue(parameters[i].getType());
            if(o!=null)
                parametersValues[i] = o;
            else
                continue external;
        }
        //We have values for all parameters
        return (Vehicle) constructor.newInstance(parametersValues);
    }
    return null;
}

static Object getConcreteValue(Class type) {
    if(type.equals(FuelType.class))
        return new FuelType("diesel", 0.015, 0.10);
    if(type.equals(double.class) || type.equals(Double.class))
        return 0.0;
    if(type.equals(int.class) || type.equals(Integer.class))
        return 0;
    return null;
}
```

We are almost there! However, some vehicles (subtype of `Car`) need to be refuelled before starting the race as follows.

```
static void refuel(VehicleHashSet<Vehicle> allInstatiatedVehicles) {  
    for(Vehicle v : allInstatiatedVehicles)  
        if(v instanceof Car)  
            ((Car) v).refuel(20.0);  
}
```

Finally, we can implement the main method that constructs the objects and run the race.

```
public static void main(String[] args)  
    throws ImpossibleAccelerationException, InvocationTargetException,  
            InstantiationException, IllegalAccessException {  
    Collection<Class> allVehicles = getAllVehiclesClasses();  
    VehicleHashSet<Vehicle> allInstatiatedVehicles = new VehicleHashSet();  
    for(Class c : allVehicles) {  
        Vehicle v = createInstanceOf(c);  
        if(v!=null) allInstatiatedVehicles.add(v);  
        else System.err.println("Unable to instantiate vehicle "+c.getName());  
    }  
    refuel(allInstatiatedVehicles);  
    System.out.println("The winner is "+allInstatiatedVehicles.race(100.0));  
}
```

This short example exploits almost all the features of Java we have seen so far: standard methods of class Object (such as equals in method getConcreteValue), auto boxing of numerical values (getConcreteValue), collections, exceptions, reflection, etc.. In addition, it heavily relies on several components discussed in the part about polymorphism, from inheritance (to define class VehicleHashSet), to generics (in all the collections) and dynamic dispatching of method calls. Essentially, all the concepts we have seen up to now are almost always part of any non-trivial Java program we might think of, even if it is composed by few code lines and it performs a relatively simple task as the one presented in this section.

16.8 Exercise

Given the following classes, rewrite the main method using only the reflection.

```
package it.unive.dais.pol.id;

public class Main {

    public static void Main(String[] args) {

        CarID id = new CarID(args[1], args[2], args[3]);

        String model = id.getModel();

        if(model.equals("Lamborghini"))
            System.out.println("Nice choice!");

        if(id.isValidLicensePlate())
            System.out.println("The license plate \""+id.licensePlate() + "\" is valid!");
    }
}
```

```
package it.unive.dais.pol.id;

public final class CarID extends RegisteredVehicleID {

    public CarID(String make, String model, String licensePlate) {
        super(make, model, licensePlate);
    }

    public final boolean isValidLicensePlate(){
        ...
    }

    ...
}
```

```
package it.unive.dais.pol.id;

public abstract class RegisteredVehicleID {

    protected final String make;
    protected final String model;
    public final String licensePlate;

    public RegisteredVehicleID (String make, String model,
                                String licensePlate) {
        this.make = make;
        this.model = model;
        this.licensePlate = licensePlate;
    }

    public abstract boolean isValidLicensePlate();

    public String getModel() { return model; }
    ...
}
```



Chapter 17: Library management

When we write Java programs, we often use external libraries, where by library we mean an artifact containing the bytecode of some classes. So far, we focused the attention on a set of classes and packages that are part of the Java standard library. This library is automatically loaded when we execute a Java program, and we can import and use those components without the need of including them in the runtime environment. However, Java offers means to add more libraries to the runtime environment and use them in our applications.

17.1 Classpath

The classpath defines all the paths where the JVM looks for the classes that are loaded and used by the runtime environment. Therefore, the classpath is composed of a series of paths. Each path could be either (i) a directory containing class files structured following the packages classes belong to (e.g., if a class is in package `it.unive.dais`, then it should be placed in the subdirectory `it/unive/dais`), or (ii) jar files. When the execution is launched and the program uses some classes, the JVM will look into the various paths in order to find them. If a class is not found, then a `java.lang.NoClassDefFoundError` is thrown and the execution is terminated.

For instance, let's imagine that we want to split our code about vehicles into two separate libraries. On the one hand, we want to put all the classes defining vehicles. On the other hand, the classes to run a race. We then get two distinct artifacts, let's say in directory `/vehicle` and in `/race`, and we want to execute class `it.unive.dais.po1.Race`. The latter depends on the former, since it uses the interface of `Vehicle` in order to run the race. So far, we have seen that we can execute “`java it.unive.dais.po1.Race`” in order to execute this class. If we run from the directory `/vehicle`, we get the following exception:

```
Error: Could not find or load main class it.unive.dais.po1.Race
Caused by: java.lang.ClassNotFoundException: it.unive.dais.po1.Race
```

Indeed, such class is not present in that directory. However, if we run it from `/race`, we

get another error.

```
Error: Could not find or load main class it.unive.dais.pol.Vehicle  
Caused by: java.lang.ClassNotFoundException: it.unive.dais.pol.Vehicle
```

Indeed, when loading class `Race` the JVM looks for `Vehicle`, and it does not find it. Java allows passing to various sdk tools (such as `javac`, `java`, and `javadoc`) a `-classpath` option followed by several paths (separated by `:` or `;` depending on the operating system⁶¹). In this way, we can specify more than one path, and load classes from various sources. Therefore, we can execute our `Race` class as follows:

```
java -classpath /vehicle;/race it.unive.dais.pol.Race
```

And the execution will complete successfully.

17.2 Internal and external interfaces of libraries

In the part of this book about encapsulation, we largely debated about how one can hide a part of the implementation through several access modifiers (`public`, `protected`, `default/package` and `private`). This creates two levels of interface of a class: its internal and its external interface. The external interface is composed of all the elements that are externally visible, that is, at least all `public` components. There might be additional components (e.g., `protected` methods that must be overridden by the library client) that are part of the external interface, but for the sake of simplicity we will consider only `public` components⁶². Instead, the internal interface is composed of all the elements that are intended only for internal use. This usually comprises all `private`, `default/package`, and `protected` components.

17.3 Types of changes

Libraries evolve over time. In particular, we might add, modify, or remove components. Those components might be classes, interfaces, fields, methods etc.. and they might belong either to the external or the internal interface. This is a standard process: we first release a library with some functionalities, and then we improve it. Improving might mean that we discovered some bugs and we fix them, we add more functionalities, or we remove old functionalities that are not anymore needed or supported.

⁶¹ <https://docs.oracle.com/javase/8/docs/technotes/tools/windows/classpath.html>

⁶² <https://commons.apache.org/releases/versioning.html> reports a detailed and deep practical discussion about those interfaces, changes, as well as version numbers. In this book, we present a simplified view of this approach, but the interested reader should refer to it to get more details about the overall approach.

Libraries are released at a certain point. The release process consists in producing and distributing a unique artifact. Each release produces a specific version of the library. Therefore, given a library and a version, we have a unique artifact that represents it.

When we release a new version of our library, we would like to be *backward compatible*⁶³. This means that our new version can replace the old version without needing any change in the software that uses our library. Backward compatibility is a key concept not only in OO programs, but in technology in general as well. Looking into this concept, one might realize that it is just another instance of the substitution principle we introduced in section 8.1, and we deeply discussed in the part about polymorphism. In particular, a new version of our library can substitute an old version if it provides all the functionalities of the previous version, plus, in case, something else. We need therefore to define what we mean by providing a functionality in an OO library.

First of all, a functionality is an OO component that is visible from outside, that is, that is part of the external interface of the library. Roughly speaking, this means that functionalities are all public classes, fields, and methods of our program. How can we modify it? As previously discussed, we might add, modify, or remove a component. For the sake of simplicity, let's focus on methods. If we add a method to our external interface, essentially we are providing more functionalities, and therefore this won't break backward compatibility. Instead, if we modify it, it depends what we are doing exactly. If we change its interface (e.g., add, remove or change the types of parameters) or the semantics of its implementation (meaning that the method might have a different behavior), we definitely change its external interface. If instead we simply fix a bug, or optimize the code, we are still backward compatible. Finally, if we remove a method, we definitely change its external interface. This is the reason why the annotation `@Deprecated` is widely used: a method is not anymore supported or interesting, but it is not removed by its external interface as this would break the backward compatibility of the library.

From this discussion, it becomes clear that we might have changes that are interface compatible, that is, they do not make substantial changes to the external interface of the library, and others that are not⁶⁴.

⁶³ https://en.wikipedia.org/wiki/Backward_compatibility

⁶⁴ The Apache commons webpage about versioning mentioned above contains a more fine-grained discussion about this point

17.4 Versioning numbers

Finally, we can discuss how we can number different versions of our libraries. We have three different types of releases:

- Point or bug fixes, where the external interface of the library is untouched (not even adding more functionalities or changing the semantics of some components), and therefore we might only fix errors or optimize our components;
- Minor, where the changes are interface compatible, thus backward compatibility is preserved; and
- Major, where the external interface is modified and therefore backward compatibility is not guaranteed.

For this reason, a version number is composed of three distinct digits separated by dots. For instance, 30.0.1 represents the major release 30, minor release 0, and point or bug fix release 1.

Generally speaking, major releases should be rare and limited to the cases where we definitely need to modify the external interface of our library. In fact, the clients of our library will need to adjust their code and configuration in order to be compatible with the new version. If such effort is too high, they will be resistant to move to the new version, and in the long term might be stuck with extremely old versions of our library or move to another library. In practice, major releases are produced once every several months, minor releases once every several weeks, and point/bug fix releases are usually produced on demand (e.g., a few days after a minor or major release since an unexpected bug arised).

17.5 Build automation tools

You might have noticed that the complexity of the process of building and running a Java application is becoming more and more complex. Managing such complexity manually is risky and error prone. Therefore, several tools to automate the building process appeared. We can distinguish two main parts of this process: (i) the management of libraries (i.e., declare what libraries and versions are needed, and retrieve them), and (ii) the compilation and execution of the program. There are tools that support both, and tools that support only one task.

For instance, Apache Ant⁶⁵ is focused on building, while Apache Ivy⁶⁶ is focused on library management. Instead, Apache Maven⁶⁷ and Gradle⁶⁸ comprise both. Each technology has a different way to specify the same things (Ant, Ivy and Maven rely on XML files, while Gradle relies on Groovy or Kotlin files). The current trend leans towards solutions that support both build and library management, and Maven and Gradle are probably the two most popular technologies in this field nowadays. In the next section, we will focus on Gradle, but the same concepts (with a different syntax) apply to Maven and Ant+Ivy as well.

17.6 Gradle

Let's start by looking into a rather simple Gradle build file⁶⁹ composed of three main components (plugins, repositories, and dependencies).

```
plugins {  
    id 'application'  
}  
repositories {  
    mavenCentral()  
}  
dependencies {  
    testImplementation group:'junit', name:'junit', version:'4.13'  
}  
mainClassName = 'it.unive.dais.pol.Race'
```

Intuitively, `plugins`⁷⁰ defines what type of program we want to compile. Gradle supports many programming languages, and the same programming language might have different types of plugins. For instance, Java code might be built into an application (aka with a class containing a main method that should be executed when we want to run the application), a library (where there is no main method), a platform, an IntelliJ IDEA plugin, etc.. In addition, we might want to define several different types (e.g., we might have a piece of Java code that might be both an application and a library). Therefore, inside the `plugins` section we can specify several plugins. In our case, we define a Java application. In addition, we need to define the name of the class containing the main

⁶⁵ <https://ant.apache.org/>

⁶⁶ <https://ant.apache.org/ivy/>

⁶⁷ <https://maven.apache.org/>

⁶⁸ <https://docs.gradle.org/current/userguide/userguide.html>

⁶⁹ In this book, we do not enter into the details of Kotlin/Groovy, since the Gradle's build files can be easily understood without full knowledge of their syntax.

⁷⁰ https://docs.gradle.org/current/userguide/plugin_reference.html

method. This is done in the line of `build.gradle` by assigning `it.unive.dais.po1.Race` to variable `mainClassName`.

The second section of `build.gradle` regards repositories. We discussed libraries and their versions, but how do we share libraries? Through repositories! Despite the fact that everyone might open his own repository, nowadays Maven Central⁷¹ is the most common repository with more than 24M artifacts. However, please be aware that not all libraries are developed by careful people: as you can easily imagine, the most part of those libraries is not maintained, or even it was just published once quite a long time ago. Therefore, when choosing if and what libraries you want to use, always check if those libraries are safe, reliable, and maintained. For instance, projects like Apache Commons⁷² and Guava⁷³ are often a de-facto standard to support various functionalities. They are used by thousands of other artefacts, and downloaded millions of times. In our `build.gradle`, we use only the Maven central repository.

Finally, the third and last section of `build.gradle` lists all the dependencies of the software project. In particular, for each library, we need to first define the configuration (i.e., if we want to use it only to compile our code, to compile and run it, only for testing purposes, etc etc..). Then, we need to specify the group that produced the library, the name of the library, and the version. With this information, Gradle can retrieve from the repositories (Maven central in our example) the artefact, cache them, and use them when building, running and/or testing our system.

Gradle is fully integrated in modern IDEs. This means that we can exploit the GUIs in order to see what dependencies we have, run build scripts, manage repositories, etc.. This dramatically eases the effort required to correctly build, test and run our code, since we do not need to manually define commands (specifying the classpath, ...), and the build automation tool takes care of the whole process.

17.7 Using Gradle to exploit a reflection library

Let's be honest: the code we developed in section 16.7 was not so satisfying, since we had to manually add each single class in the result of our `getAllVehicleClasses` method. Therefore, we needed to know upfront what classes extended `Vehicle` in order to build up an exhaustive list of vehicles that can run our race. The standard library of Java does

⁷¹ <https://mvnrepository.com/>

⁷² <https://commons.apache.org/>

⁷³ <https://github.com/google/guava>

not provide means to retrieve all the subclasses of a given class. This happens mostly because of the class loading mechanisms that Java implements. However, there are libraries that support such functionality. In particular, reflections⁷⁴ is a rather popular library that provides a wide range of utilities. For our purposes `Reflections.getSubTypesOf(Class)` returns all the subtypes starting from a given package of a given class. First of all, we need to add to our Gradle dependencies this library in `build.gradle` as follows.

```
dependencies {  
    implementation group: 'org.reflections', name: 'reflections', version: '0.10.2'  
}
```

Then we can modify our `getAllVehicleClasses` as follows.

```
static Collection<Class> getAllVehiclesClasses() {  
    HashSet<Class> result = new HashSet<>();  
    Reflections reflections = new Reflections("it.unive.dais.pol.vehicles");  
    Set<Class<? extends Vehicle>> allClasses = reflections.getSubTypesOf(Vehicle.class);  
    result.addAll(allClasses);  
    return result;  
}
```

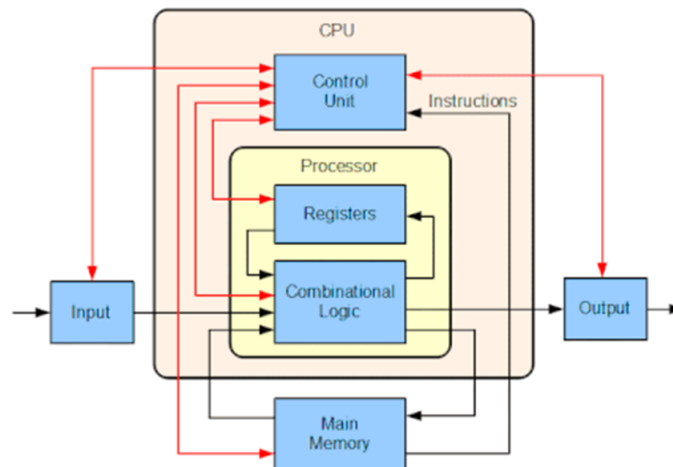
In this way, each time a class extending `Vehicle` will be added to the classpath, `getAllVehicleClasses` will retrieve it and put it inside the results of its computation.

⁷⁴ <https://github.com/ronmamo/reflections>

Chapter 18: Model View Controller and Spring

How we develop software today is different from how we developed it 1 year ago, that was different from 5 years ago, that was different from 10 years ago, and so on. Indeed, the architecture of software has changed quite dramatically over the last decades. Starting from mostly embedded software (executed on devices with limited capabilities whose main goal was to manage such devices through sensors and actuators), software moved to desktop applications (executed locally on a machine whose main goal was usually to manage enterprise tasks), and finally to Web and cloud software (executed on remote machines connected to Internet to the client and providing a wide range of different functionalities, from e-banking to social networking and messaging). And, obviously, we cannot know what will happen in 1, 5 and 10 years. However, in this chapter we present one of the main applications of OO software nowadays (meaning at the beginning of 2020s).

18.1 von Neumann architecture



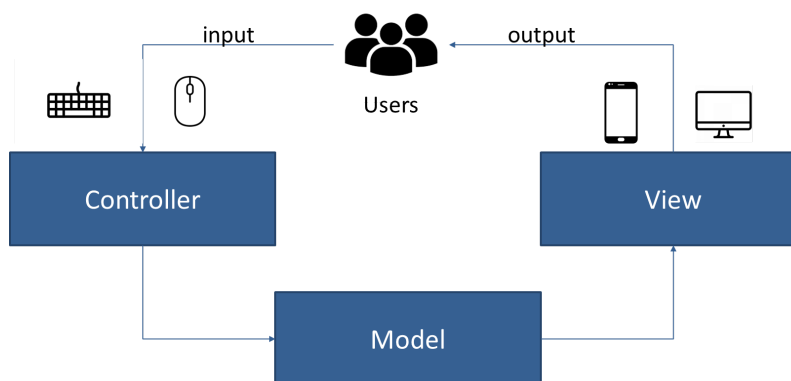
John von Neumann defined the computer architecture depicted in the picture above in

1945. This is still the architecture of our modern computers. The model is very simple: we have an input, something in the middle that performs computations and produces an output. The “something in the middle” is composed by a memory, a control unit that decides what to execute, and a combinational logic that performs the computations reading and writing values on some local registers. All modern devices fall into this simple model: smartphones, tablets, laptops, desktop computers, etc etc..

And any existing programming language produces programs to be executed on such machines. Even looking back to what we covered in this book, essentially Java programs allocate memory, perform some computation, store information in the memory, etc etc... Of course the means provided by OO programs to perform those operations are quite higher level than the fine grained operations effectively performed by the computer. For instance, we cannot allocate an arbitrary number of bytes, but we can allocate the memory needed to store the state of an object instance of a specific class.

The model of program we developed in the examples in this book mostly concerns the whole von Neumann architecture. Indeed, the applications we defined are all main methods that, starting from some input (e.g., the array of string values passed as parameter to the main method), we performed some computations producing some output (usually printed in the console). Therefore, in some ways our program took care of reading the input, implementing the logic of our model, and rendering the output. While this is fine for simple systems, such as part of the embedded software and the desktop applications, it becomes unmanageable when we move to Web applications and cloud systems, where several machines communicating through a shared network are involved.

18.2 MVC pattern

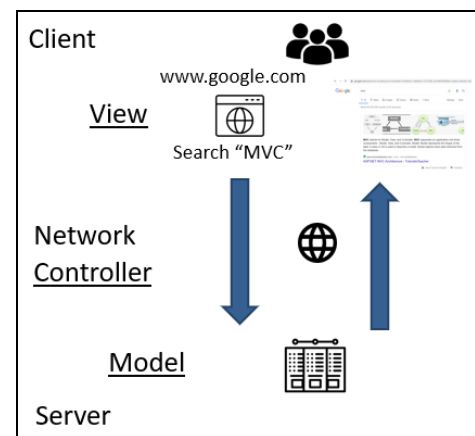


Modular reasoning is at the basis of developing complex technological systems, as we stressed out in the first chapter when talking about the Eiffel tower. In particular, we do not want to mix up how input is collected with the logic of our digital model and how the output is rendered. The model view controller (MVC) pattern is aimed exactly at defining such software architecture. The figure above depicts its flow. The controller receives the input from the user, and it dispatches them to the model. This then performs the core of the computation and it returns some output. This output is finally rendered through the view component.

Ideally, these three components can work completely independently without knowledge about the other parts. However, achieving this is not so straightforward. For instance, when there is a new input from the user, should it be the controller to inform the model, or the model to query the controller to see if there is new input? And should it be the view to ask the model if there is new output, or the model to inform the view? In any case, some dependencies would be created!

18.3 MVC for Web applications

The MVC model represents an effective approach to develop Web applications. The figure on the right depicts the MVC model applied to Web applications. Users interact with our application as remote clients. They have a view (usually a Web browser) that allows them to send a request to the server and receive a response. The controller manages the communications from the client to the server, while the model is fully executed server-side. Finally, the view allows the client to visualize the answer from the server.



As one can easily imagine, when we talk about such a complex system there are several different technologies that come into play. Nowadays, views are mostly represented as HTML pages (often containing some code, e.g., JavaScript), the model might be implemented with many different programming languages and patterns, and the same happens for the controller. In the rest of this chapter, we will focus on the main technologies available for developing such applications in Java, but please keep in mind that this is just one possibility (and, indeed, not necessarily the most popular one!).

18.4 J2EE and Spring

This book is about the Java programming language. However, there are also several Java platforms⁷⁵. For instance, Java Card is a platform for Java software on smart cards, Java Micro on embedded systems, Java Standard Edition a general purpose platform, and Java Enterprise Edition is focused on multi-tier client server enterprise applications. So far the core of the Java libraries we discussed (e.g., exceptions, annotations, collections) are part of all these platforms. Here we will focus on the Java Enterprise Edition.

Indeed, nowadays the Java Enterprise Edition evolved into the Jakarta Enterprise Edition⁷⁶. Instead, the Spring framework⁷⁷ is an evolution of Java Enterprise Edition. Those two platforms are nowadays the most popular choices when it comes to developing Web applications in Java. Indeed, in recent years Spring was a bit more popular than Jakarta EE, and therefore we will focus on this framework. Anyway, the same concepts can be applied to other platforms with mostly small adjustments to the syntax of the program.

However, we need first of all to discuss what is the difference between a library and a framework. A library is a piece of software that we import in our classpath, and that it is used to perform some computations. In the previous chapter we have seen how we can create dependencies between our code and libraries in order to correctly compile and execute our program through Gradle. However, the main concept is: our program uses the code of libraries.

With frameworks we apply the so-called inversion of control. What do we mean? So far, we mostly run programs where we implemented a main method that starts the execution of our code. Instead, inversion of control means that the framework takes the control of the execution, and it decides what to execute and how. We have already seen this in action when we run tests through JUnit: we annotated a method with `@Test` and “by magic” (that is, by reflection) JUnit executed it. This is an example of frameworks. Usually, modern Java frameworks rely on annotations in order to specify framework specific components through means that are structured and visible at runtime. Then they inspect what is available at runtime through reflection, and they execute some parts of our code.

⁷⁵ [https://en.wikipedia.org/wiki/Java_\(software_platform\)](https://en.wikipedia.org/wiki/Java_(software_platform))

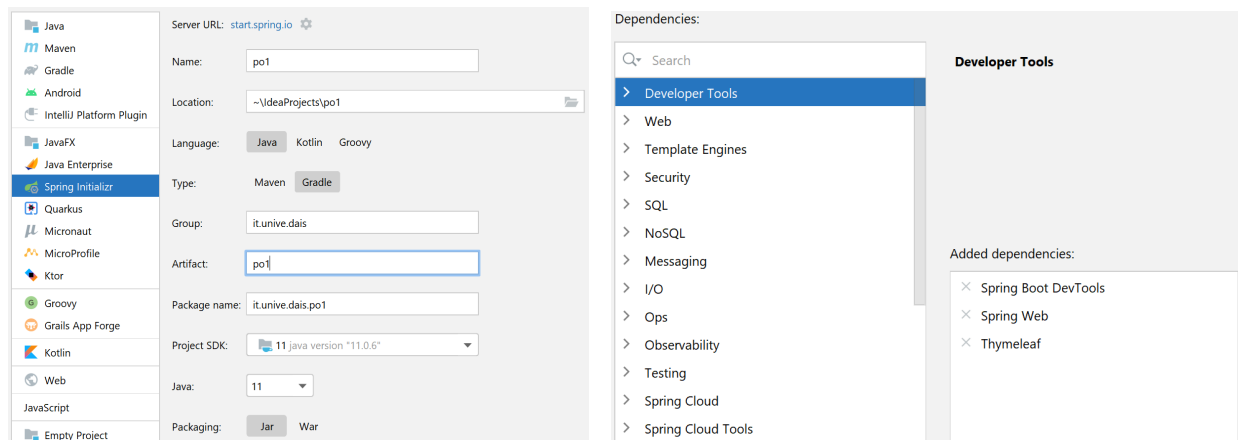
⁷⁶ https://en.wikipedia.org/wiki/Jakarta_EE

⁷⁷ <https://spring.io/>

18.5 A simple Spring application

The Spring framework is a rich technological ecosystem that allows the development of a large variety of applications such as microservices, Web applications, and cloud services. In this section, we give a quick overview⁷⁸ of some basic functionalities to develop Web applications based on the MVC architecture.

The infrastructure required by this architecture is rather complex: we need to run a server to test our code, connect through a browser, install and run our application on the server, etc... Luckily, Spring offers a tool, called Spring Boot, that eases this work: essentially, with Spring Boot we can just push a button and our code will run, since Spring Boot takes care of the whole infrastructure effort. In addition, Spring Boot is integrated with all the most popular IDEs⁷⁹, easing even more the effort required to start to play with Spring.



First of all, we start by creating a new Spring project. We need to specify some information about our project, such as the group it belongs to (that corresponds to the group of the dependencies we declared through Gradle), the name of the artifact, what version of Java we want to use, and what type of build automation tool we want to use (we selected Gradle since this is the tool we already discussed).

Then we need to choose the dependencies we are interested in. For the purposes of this demo we chose:

⁷⁸ Based on the tutorials published at <https://spring.io/guides/gs/spring-boot/> and <https://spring.io/guides/gs/serving-web-content/>

⁷⁹ In this section, we'll provide screenshots taken from IntelliJ IDEA, but other IDEs offer identical integrations with Spring.

- Spring Boot DevTools among the Developer tools. This provides some facilities to run, debug, restart, etc.. Spring applications during their development;
- Spring Web among the Web dependencies. This provides support for the MVC architecture; and
- Thymeleaf among the Template Engines. This provides a convenient template engine for creating HTML pages.

We can then start with our first Spring application!

```
import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;

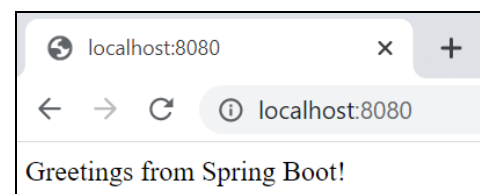
@SpringBootApplication
public class OurSpringApplication {
    public static void main(String[] args) {
        SpringApplication.run(PolApplication.class, args);
    }
}
```

Nothing so impressive: we have a main method that runs a Spring application that, essentially, does nothing. However, note that we must annotate our class with `@SpringBootApplication`: in this way, Spring Boot will see it at runtime (... through reflection!), and it will be able to build up the whole infrastructure running our application. Let's add a Web page that displays a message.

```
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RestController;

@RestController
public class HelloController {
    @RequestMapping("/")
    public String index() {
        return "Greetings from Spring Boot!";
    }
}
```

Also in this case, we relied on annotations: (i) `RestController` specifies that a class is a Web controller, and (ii) `RequestMapping` specifies that a method handles a request arrived at a given address. Therefore, method `index()` will be executed when somebody connects to the root of our server. When



Spring Boot runs, we can then connect to <http://localhost:8080/80> to see the message returned by this method.

Let's move to a more complex example... an “hello world” Web application! Indeed, we want to develop an application that if no value is passed, then it displays “Hello world”. Instead, if a value is given (e.g., Pietro), it displays “Hello Pietro”. First of all, we need to define another controller.

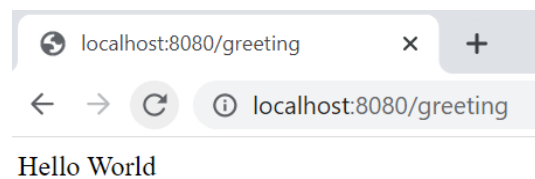
```
import org.springframework.stereotype.Controller;
import org.springframework.ui.Model;
import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.RequestParam;

@Controller
public class GreetingController {
    @GetMapping("/greeting")
    public String greeting(@RequestParam(name = "name", required = false,
                                     defaultValue = "World") String name, Model model) {
        model.addAttribute("name", name);
        return "greeting";
    }
}
```

This controller listens the path “/greeting”. However, the structure of the method is rather more complex than the previous example. In particular, we need a parameter with the value passed to our application, and a Model object to set up properly the results of our model in order to correctly visualize the response through a view. Therefore, our method receives (i) a parameter name annotated with @RequestParam to specify that it comes from the request parameter name, it is not required, and in case it is not specified its default value is World, and (ii) a parameter Model that will be used afterwards to produce the final view. The implementation is extremely simple: the request parameter name is added to the model. Finally, we need to implement the view as follows.

```
<html xmlns:th="http://www.thymeleaf.org">
<body>
<p th:text="Hello ' + ${name} + '" />
</body>
</html>
```

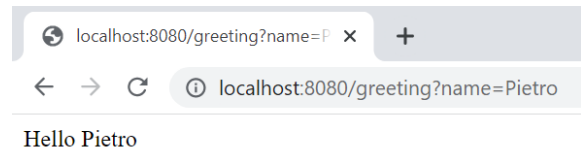
The view is implemented as an HTML page that produces a paragraph inside the body with the string “Hello “ concatenated with the value passed as name through the model. If we connect



⁸⁰ These are the default addresses and values where Spring Boot publishes the application.

to page `/greeting` without specifying a request parameter name, we obtain the message “Hello World” as depicted in the figure on the right. In fact, our controller defined through the `@RequestParam` annotation that the default value of such parameter was `World`.

We can also pass a request parameter by specifying a `name=<value>` through the URL. In this way, the value is passed to method `greeting` and then added to the model read by the view. Therefore, if we connect to `/greeting?name=Pietro` we get the message “Hello Pietro” as depicted in the figure on the right.



Even if these examples are extremely simple, we can already see all the three components in action: the controller specifies what methods should be executed when a request arrive, the implementation of these methods (should) perform some computation setting up the results in the Model instance, and finally the view displays these values to the client. We can already notice that, even for such a simple example, we need to rely not only on Java code and annotations, but also on HTML Web pages. This is quite common when we develop Web applications: several languages are involved, based on what is more convenient. Obviously, this demo covers probably 0.01% of the Spring framework, and the interested reader should refer to the Spring documentation (starting from <https://spring.io/guides>) for details on the different components.

18.6 Exercise

Considering MVC paradigm, create a program that allows to add, remove or search car identity (make, model, license plate, VIN). The actions (add, remove) will given as input from the keyboard. The information must be stored in a collection without duplicates. The various updates and requests be displayed in the console output.



Appendix A: The Java technology

The main goal of this book is to introduce the main concepts of object oriented programming. Many languages (such as Java, C#, C++, SmallTalk just to name a few) have been developed during the last decades following this paradigm. Among the many, we chose Java for all our examples and concrete discussions. This choice was driven by two main reasons: (i) Java has been the most popular OO programming language in the last 20 years, and (ii) Java cleanly supports all the OO concepts discussed in this book. However, we might have chosen another language obtaining exactly the same results, and indeed we aimed at exploring the extension of this book with a multi-language approach, where OO concepts are presented throughout various programming languages. In order to play with Java, it is indeed necessary to clarify this technology, and what exact pieces of this technology we are interested in.

A.1 Programming language and APIs

When we talk about Java technology we mean mainly two distinct things: the programming language, and the Application Programming Interface (API, that is, the standard library). The first version of Java appeared in 1995, and over the years both the language and the APIs changed in many different ways. Most of the OO concepts presented in this book were already supported by this first version. The following table reports all the versions of the Java technologies that appeared over the years, where LTS denotes versions that are long term support.

Version	Release date	Version	Release date	Version	Release date
JDK Beta	1995	Java SE 7	Jul-11	Java SE 15	Sep-20
JDK 1.0	Jan-96	Java SE 8 (LTS)	Mar-14	Java SE 16	Mar-21
JDK 1.1	Feb-97	Java SE 9	Sep-17	Java SE 17 (LTS)	Sep-21
J2SE 1.2	Dec-98	Java SE 10	Mar-18	Java SE 18	Mar-22

J2SE 1.3	May-00	Java SE 11 (LTS)	Sep-18	Java SE 19	Sep-22
J2SE 1.4	Feb-02	Java SE 12	Mar-19	Java SE 20	Mar-23
J2SE 5.0	Sep-04	Java SE 13	Sep-19	Java SE 21	Sep-23
Java SE 6	Dec-06	Java SE 14	Mar-20		

While each version introduced various features and novel APIs, only a few versions introduced relevant changes with respect to the concepts of OO programming we discussed in this book. In particular, Java 5 introduced generic types, Java 8 lambda expressions and functional interfaces, and Java 9 modules.

Note that while the first versions of Java (up to Java 8 included) were released every 2 or 3 years, starting from Java 9 new versions were released every 6 months. This was a change that Oracle decided in order to have more frequent releases, but only a few of them (1 every 6 releases) are long term support, while all the others are supported only until the next release is out. In practice, this means that the novel Java versions that are not LTS are intended only for a temporary use, and they should be replaced by the new releases as soon as they are out. Therefore, from the perspective of learning Java, we strongly suggest to install and use only LTS versions (and in general it is a good practice to rely on the latest version).

A.2 Compiling and executing Java programs

Java source code is compiled into bytecode, that is, an intermediate representation at a higher level of abstraction than assembly code, but less structured than source code. This bytecode consists of statements that perform basic arithmetic operations, reading and writing local variables, heap access and writes, method calls, jumps, and so on. The Java Virtual Machine Specification⁸¹ reports the complete set of instructions of this language. However, in order to execute such code a virtual machine is needed. In this way, Java bytecode can be ported and executed on different hardware platforms, while virtual machines will be specific for some given platforms.

Generally speaking, we can download and install two main distributions of Java. The Java Runtime Environment (JRE) comprises the Java Virtual Machine, and it allows us to run Java bytecode programs. A JRE is able to run Java programs of its version or any previous versions. The Java Development Kit (JDK) includes the JRE, and it contains the

⁸¹ <https://docs.oracle.com/javase/specs/jvms/se8/html/>

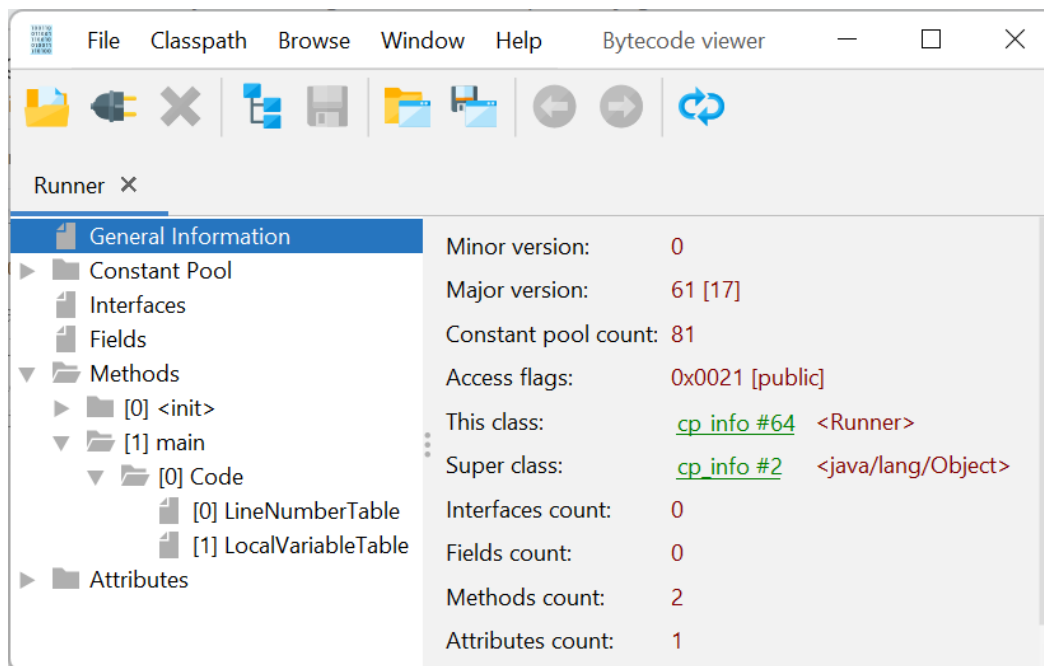
compiler as well as other development tools. Therefore, it is important to note that if we want to develop Java code we need to install a JDK, and not only a JRE.

A.3 Java bytecode

The Java bytecode is an intermediate language that is at a lower level than source code, but at a higher level than assembly code. In particular, bytecode is not structured (thus, we don't have if-then-else, while, for, etc.. statements but we have gotos and conditional jumps) and its execution model relies on an array of local variables, and an operand stack. All together, Java bytecode encompasses about 200 distinct statements. The full specification of this language can be found inside the Java Virtual Machine specification⁸². The rest of this section will intuitively introduce the main characteristics of Java bytecode.

How to inspect Java bytecode

Java bytecode is stored into .class binary files. Several tools allow you to inspect those files. In particular, jclasslib⁸³ is a graphical user interface that allows users to open, navigate and modify those files. There are various plugins that integrate this tool with different IDEs such as IntelliJ IDEA⁸⁴.



⁸² <https://docs.oracle.com/javase/specs/jvms/se17/html/index.html>

⁸³ <https://github.com/ingoegel/jclasslib>

⁸⁴ <https://plugins.jetbrains.com/plugin/9248-jclasslib-bytecode-viewer>

The figure above shows what can be navigated when a single .class file is opened. In particular, we get information about all the different components of the class, from its accessibility, name and super class, to the fields and methods defined inside it. For each method, we have its code, line number table (that is, some debug information that tells to from what line of the source code a specific bytecode instruction was obtained), and local variable table (that for each local variable reports its name in the source code).

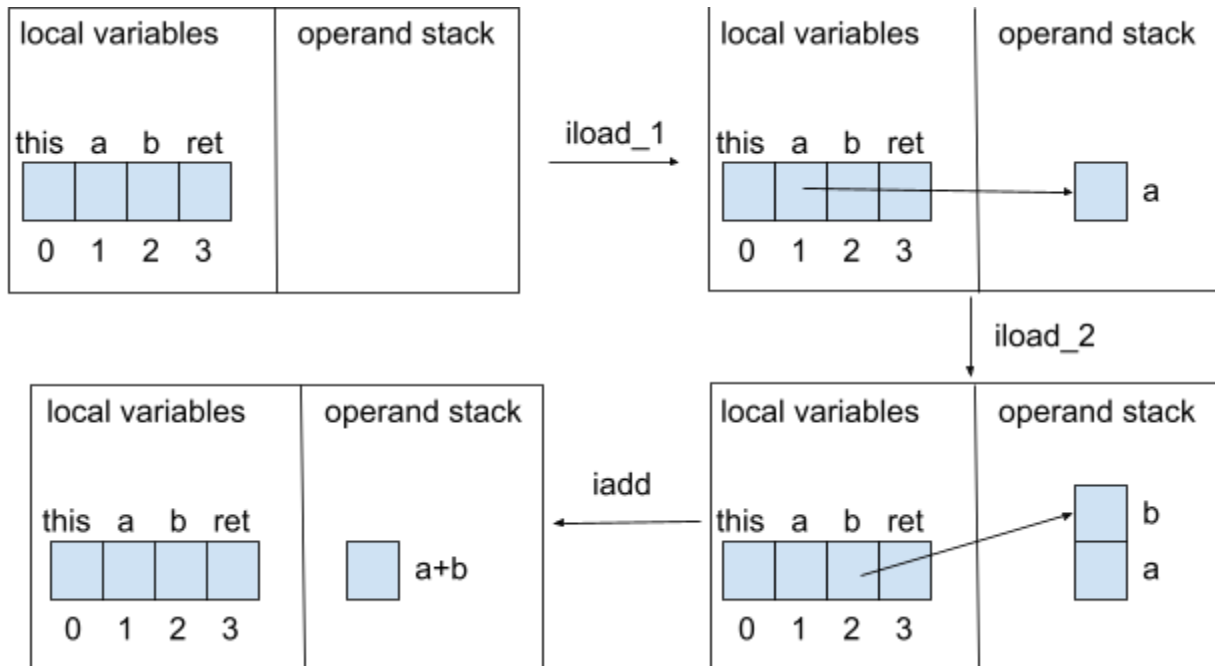
Arithmetic computations and local variables

Let's start with a minimal example: we implement a method that receives two integer parameters and it returns its sum. The source code looks as follows:

```
int sum(int a, int b) {  
    int res = a+b;  
    return res;  
}
```

This code is compiled into the bytecode on the right side. First of all, notice that all statements start with an i. This means that they all deal with integer values, and there are similar statements for other numerical types (float, double, long, ...). We have then statements that load the value of a local variable to the top of the operand stack (e.g., `iload_1` loads the integer value contained in the local variable at index 1), that store the value at the top of the operand stack into a local variable (e.g., `istore_3` stores the integer value at the top of the operand stack into local variable 3), that add the two integer values at the top of the operand stack (i.e., `iadd`), and that return the integer value at the top of the operand stack to the caller (`ireturn`). A graphical representation of the flow of execution for the first three statements is reported in the figure below.

```
0  iload_1  
1  iload_2  
2  iadd  
3  istore_3  
4  iload_3  
5  ireturn
```



Accessing the heap

Let us assume that our class `Runner` has an integer field, and we implement the following method.

```
void incrementByOne() {
    f++;
}
```

This code is compiled into the bytecode on the right side. First of all, note that the source code implicitly accesses the object pointed by `this`. Such operation is made explicit in the bytecode, since it starts by getting the reference of local variable 0 (that contains variable `this`) and duplicating its value. It then gets the value of field `f` through statement `getfield`: this statement gets the reference at the top of the stack, and replaces it with the value contained in the given field of the pointed object. The bytecode then loads the integer constant value 1, adds the two values at the top of the stack (that is, `this.f` and 1), and it finally put the values at the top of the stack into field `f` of the object pointed by the reference stored in the second slot of the operand stack through statement `putfield`. Finally, `return` ends the execution of the current method without returning a value.

```
0  aload_0
1  dup
2  getfield #7 <Runner.f : I>
5  iconst_1
6  iadd
7  putfield #7 <Runner.f : I>
10 return
```

Invoking a method

As the last example, let's consider the invocation of method `add` previously defined. In particular, we compile the following Java statement.

```
int x = sum(1, 2);
```

We obtain the bytecode reported on the right side. First of all, such bytecode loads a reference to `this` (through local variable zero), and then the two integer constant values passed to the method (1 and 2).

```
0 aload_0  
1 iconst_1  
2 iconst_2  
3 invokevirtual #7 <Runner.sum : (II)I>  
6 istore_1
```

The invocation of method `sum` follows: this statement, takes the values of the parameters from the top values of the operand stack, and then the reference to the object that is the receiver of the method invocation. After this statement, the value returned by the method is stored at the top of the operand stack. Our bytecode then stores this value into the local variable 1, that corresponds to local variable `x` in our source code.

Exercise Solutions

2.8 Exercise Solution

It is possible to identify a vehicle with various information, for example with make and model, through the license plate or more uniquely through the Vehicle Identification Number (VIN). All these pieces of information are data and can be represented using fields, since these identifiers are alphanumeric sequences. In addition, among the functionalities it could be interesting to be able to check the validity of some of these data, saying if they are valid or not. For instance, the license plate in Italy must have a length of 7 alphanumeric characters of the CCNNCC type, where N is a digit and C is a different alphabetic character from “I” and “O”, and the sequences CC must not be “EE” because the double “E” is not allowed.

An example of a solution is given below:

```
class CarID {  
  
    String make;  
    String model;  
    String licensePlate;  
    String VIN;  
  
    static boolean checkLicensePlateValidity(Car car ) {  
        ...  
    }  
  
    ...  
}
```

3.9 Exercise Solution

Below a possible solution:

```
class CarID {  
  
    final String make;  
    final String model;  
    String licensePlate;  
    final String VIN;  
  
    boolean isSameLicensePlate (Car car) {  
        return this.licensePlate.equals(car.licensePlate);  
    }  
  
    CarID(String make, String model, String licensePlate, String VIN){  
        this.make = make;  
        this.model = model;  
        this.licensePlate = licensePlate;  
        this.VIN = VIN;  
    }  
  
    ...  
}
```

For the sake of simplicity and as a design choice of the CarID class, only the licensePlate field is considered final. In fact, in case of theft, loss or destruction, or deterioration of the plate, it must be replaced with a new value. About isSameLicensePlate method, the this.licensePlate represents the plate of the current object and the car. licensePlate is the plate of the other car to compare. The equals is a method of the String class that returns true, if the string (which calls equals) is equal to the object passed as a parameter, otherwise returns false.

4.8 Exercise Solution

An example of a solution is given below:

```
package it.unive.dais.pol.id;

public class CarID {

    private final String make;
    private final String model;
    private String licensePlate;
    private final String VIN;

    public CarID(String make, String model, String licensePlate, String VIN){
        this.make = make;
        this.model = model;
        this.licensePlate = licensePlate;
        this.VIN = VIN;
    }

    public boolean isSameLicensePlate(String licensePlate) {
        return this.licensePlate.equals(licensePlate);
    }

    public String getMake() { return make; }

    public String getModel() { return model; }

    public String getLicensePlate () {return licensePlate; }

    public void setLicensePlate(String licensePlate) {
        this.licensePlate = licensePlate;
    }

    public String getVIN() { return VIN; }

    ...
}
```

The first part of the package (it.unive.dais.pol) is the same as the Car, but the second part is different because conceptually the vehicles and identity are two different things. We created an id package that contains all classes involved in the identification. For this reason, being a different package the CarID should be public to be imported in the Car class. About field access modifiers, they should be all private, since for instance we do not want to allow somebody to arbitrarily change the values. Instead, the methods can be public, in order to be used by the instances in other classes.

6.7 Exercise Solution

Below a possible solution:

```
/**
 * This method checks if the license plate of a car is compliant to Italian law.
 *
 * @param car the car to check
 * @return true if the car has a valid license plate for the Italian law
 * @requires car != null && car.getCarID() != null
 */
static boolean checkLicensePlateValidity(Car car ) {

    // get the license plate from the car
    String licensePlate = car.getCarID().getLicensePlate();

    // check if the plate exist and if it contains the exact number
    // of characters for a compliant license plate
    if(licensePlate != null && licensePlate.length() != 7)
        return false;

    // check if the digits are in the right places
    if(!Character.isDigit(licensePlate.charAt(2))
        || !Character.isDigit(licensePlate.charAt(3))
        || !Character.isDigit(licensePlate.charAt(4)))
        return false;

    // check if the uppercase letters are in the right places
    if(!Character.isUpperCase(0)
        || ! Character.isUpperCase(licensePlate.charAt(1))
        || !Character.isUpperCase(licensePlate.charAt(5))
        || !Character.isUpperCase(licensePlate.charAt(6)))
        return false;

    // check if the plate does not contain illegal letters or illegal sequences
    if(!licensePlate.contains("I")
        && !licensePlate.contains("O")
        && !licensePlate.contains("EE"))
        return false;

    return true;
}
```

We used the comments `//` to describe the meaning of operations inside the method, with a detailed description, in order to facilitate the code in case of future changes. Hence, we added a high level description using `/** */`, in order to generate java documentation and

to summarize the expected behavior of the method, without adding unnecessary verbosity. Note, we used `@requires` to warn method users to avoid null values, not to end up in exception (`NullPointerException`).

7.10 Exercise Solution

```
package it.unive.dais.pol.id;

public abstract class RegisteredVehicleID {

    protected final String make;
    protected final String model;
    protected String licensePlate;
    protected final String VIN;

    public RegisteredVehicleID (String make, String model,
                                String licensePlate, String VIN){
        this.make = make;
        this.model = model;
        this.licensePlate = licensePlate;
        this.VIN = VIN;
    }

    public abstract boolean isValidLicensePlate();

    public boolean isSameLicensePlate(String licensePlate) {
        return this.licensePlate.equals(licensePlate);
    }

    public String getMake() { return make; }

    public String getModel() { return model; }

    public String getLicensePlate () { return licensePlate; }

    public void setLicensePlate(String licensePlate) {
        this.licensePlate = licensePlate;
    }

    public String getVIN() { return VIN; }
}
```

```
package it.unive.dais.pol.id;

public final class CarID extends RegisteredVehicleID {

    public CarID(String make, String model, String licensePlate, String VIN){
        super(make, model, licensePlate, VIN);
    }

    public final boolean isValidLicensePlate(){

        if(licensePlate != null && licensePlate.length() != 7)
            return false;

        if(!Character.isDigit(licensePlate.charAt(2))
            || !Character.isDigit(licensePlate.charAt(3))
            || !Character.isDigit(licensePlate.charAt(4)))
            return false;

        if(!Character.isUpperCase(0)
            || !Character.isUpperCase(licensePlate.charAt(1))
            || !Character.isUpperCase(licensePlate.charAt(5))
            || !Character.isUpperCase(licensePlate.charAt(6)))
            return false;

        if(!licensePlate.contains("I")
            && !licensePlate.contains("O")
            && !licensePlate.contains("EE"))
            return false;

        return true;
    }
}
```

```
package it.unive.dais.pol.id;

public final class MotorcycleID extends RegisteredVehicleID {

    public MotorcycleID(String make, String model, String licensePlate, String VIN){
        super(make, model, licensePlate, VIN);
    }

    public final boolean isValidLicensePlate(){

        if(licensePlate != null && licensePlate.length() != 7)
            return false;

        if(!Character.isDigit(licensePlate.charAt(2))
            || !Character.isDigit(licensePlate.charAt(3))
            || !Character.isDigit(licensePlate.charAt(4))
            || !Character.isUpperCase(licensePlate.charAt(5))
            || !Character.isUpperCase(licensePlate.charAt(6)))
            return false;

        if(!Character.isUpperCase(0) || !Character.isUpperCase(licensePlate.charAt(1)))
            return false;

        if(!licensePlate.contains("I")
            && !licensePlate.contains("O")
            && !licensePlate.contains("EE"))
            return false;

        return true;
    }
}
```

The `RegisteredVehicleID` is the abstract class, and it contains all common fields and methods between the old `CarID` and `MotocycleID` classes. Now, the field modifiers are protected instead of private because they need to be visible in the classes that extend `RegisteredVehicleID`. About methods, getters, setters, and `isSameLicensePlate` method are in the `RegisteredVehicleID`. The only exception is `isValidLicensePlate` method because the signature of the method is the same, but the contents are different between `CarID` and `MotocycleID`. For this reason, `isValidLicensePlate` method is declared as abstract in `RegisteredVehicleID` and implemented in two different ways in `CarID` and `MotocycleID`. Note, in these two classes the method is final because we do not want that can be

arbitrarily redefined in other possible subclasses.

9.7 Exercise Solution

Below a possible solution:

```
package it.unive.dais.pol.id;

public class VehicleID {

    private final String make;
    private final String model;

    public VehicleID(String make, String model){
        this.make = make;
        this.model = model;
    }

    public String getMake() { return make; }

    public String getModel() { return model; }

}
```

```
package it.unive.dais.pol.id;

public interface LicenseValidity {

    public boolean isValidLicensePlate();

}
```

```
package it.unive.dais.pol.id;

public abstract class RegisteredVehicleID extends VehicleID
    implements LicenseValidity {

    private String licensePlate;
    private final String VIN;

    public RegisteredVehicleID (String make, String model, String licensePlate,
                                String VIN){
        super(make,model);
        this.licensePlate = licensePlate;
        this.VIN = VIN;
    }

    public String getLicensePlate () { return licensePlate; }

    public void setLicensePlate(String licensePlate) {
        this.licensePlate = licensePlate;
    }

    public String getVIN() { return VIN; }

    public boolean isSameLicensePlate(String licensePlate) {
        return this.licensePlate.equals(licensePlate);
    }

    public abstract boolean isValidLicensePlate();
}
```



```
package it.unive.dais.pol.id;

public class CarID extends RegisteredVehicleID {

    public CarID(String make, String model, String licensePlate, String VIN){
        super(make, model, licensePlate, VIN);
    }

    public boolean isValidLicensePlate() {
        String licensePlate = getLicensePlate();

        if(licensePlate != null && licensePlate.length() != 7)
            return false;

        if(!Character.isDigit(licensePlate.charAt(2))
            || !Character.isDigit(licensePlate.charAt(3))
            || !Character.isDigit(licensePlate.charAt(4)))
            return false;

        if(!Character.isUpperCase(0)
            || !Character.isUpperCase(licensePlate.charAt(1))
            || !Character.isUpperCase(licensePlate.charAt(5))
            || !Character.isUpperCase(licensePlate.charAt(6)))
            return false;

        if(!licensePlate.contains("I")
            && !licensePlate.contains("O")
            && !licensePlate.contains("EE"))
            return false;

        return true;
    }
}
```

```
package it.unive.dais.pol.id;

public class BikeID extends VehicleID {

    private final String chassisNumber;

    public BikeID(String make, String model, String chassisNumber){
        super(make, model);
        this.chassisNumber = chassisNumber;
    }

    public String getChassisNumber() { return chassisNumber; }
}
```

```
package it.unive.dais.pol.id;

public class EBikeID extends BikeID {

    private String batteryNumber;

    public EBikeID (String make, String model, String chassisNumber,
                                                             String batteryNumber){
        super(make, model, chassisNumber);
        this.batteryNumber = batteryNumber;
    }

    public String getBatteryNumber () { return batteryNumber; }
}
```

```
package it.unive.dais.pol.id;

public class MotorcycleID extends RegisteredVehicleID {

    public MotorcycleID(String make, String model, String licensePlate, String VIN){
        super(make,model, licensePlate, VIN);
    }

    public boolean isValidLicensePlate() {
        String licensePlate = getLicensePlate();

        if(licensePlate != null && licensePlate.length() != 7)
            return false;

        if(!Character.isDigit(licensePlate.charAt(2))
            || !Character.isDigit(licensePlate.charAt(3))
            || !Character.isDigit(licensePlate.charAt(4))
            || !Character.isUpperCase(licensePlate.charAt(5))
            || !Character.isUpperCase(licensePlate.charAt(6)))
            return false;

        if(!Character.isUpperCase(0)
            || !Character.isUpperCase(licensePlate.charAt(1)))
            return false;

        if(!licensePlate.contains("I")
            && !licensePlate.contains("O")
            && !licensePlate.contains("EE"))
            return false;

        return true;
    }
}
```

10.7 Exercise Solution

```
package it.unive.dais.pol.id;

public final class MotorcycleID extends RegisteredVehicleID {

    public MotorcycleID(String make, String model, String licensePlate, String VIN){
        super(make, model, licensePlate, VIN);
    }

    public final boolean isValidLicensePlate(){
        String licensePlate = getLicensePlate();

        if(licensePlate != null && licensePlate.length() != 7)
            return false;

        if(!Character.isDigit(licensePlate.charAt(2))
            || !Character.isDigit(licensePlate.charAt(3))
            || !Character.isDigit(licensePlate.charAt(4))
            || !Character.isUpperCase(licensePlate.charAt(5))
            || !Character.isUpperCase(licensePlate.charAt(6)))
            return false;

        if(!Character.isUpperCase(0)
            || !Character.isUpperCase(licensePlate.charAt(1)))
            return false;

        if(!licensePlate.contains("I")
            && !licensePlate.contains("O")
            && !licensePlate.contains("EE"))
            return false;

        return true;
    }
}
```

```
package it.unive.dais.pol.vehicle;

public class VehicleFactory {

    public static Vehicle createDefaultVehicle(String type){

        switch(type){
            case "Bicycle":
                return new Bicycle();
            case "Truck":
                return new Truck();
            case "Car":
                return new Car();
            default:
                return null;
        }
    }
}
```

The implementations just return the constructor for each type, although the return type of `createDefaultVehicle` is `Vehicle`, no casts are needed because all of the extends `Vehicle`.

```
public static void main(String[] args){

    CarID c = (Car) VehicleFactory.createDefaultVehicleID("Car");
    VehicleID v = VehicleFactory.createDefaultVehicleID("Car");
    BikeID b = (BikeID) VehicleFactory.createDefaultVehicleID("Bike");
    TruckID t = (Truck) VehicleFactory.createDefaultVehicleID("Truck");

    haveTheSameIdentity(c, v);
}

public static boolean IDVerifier.haveTheSameIdentity(RegisteredVehicleID v1,
                                                    RegisteredVehicleID v2) {

    ...
}
```

`VehicleFactory.createDefaultVehicle` returns only a `Vehicle` type, for this reason it is necessary to cast the correct types during the variable assignments. Note, the cast will be performed dynamically during the execution. If you mistakenly wrote something like `BikeID b = (BikeID) VehicleFactory.createDefaultVehicleID("Car")`, this will lead to a class

cast exception during the execution.

To avoid possible class cast exceptions the result of `VehicleFactory.createDefaultVehicle` can be checked using `instanceof`.

For instance:

```
Car c = null;
Vehicle tmp = VehicleFactory.createDefaultVehicle("Car");
if( tmp instanceof Car){
    c = (Car) tmp;
    ...
}
```

No class casts are needed in the calls of `haveTheSameIdentity`, using `c` and `v` because all instances are `RegisteredVehicleID`, because their classes extend it. Instead, it is not possible to call `haveTheSameIdentity`, using `b` and `t` because `b` extends `VehicleID` but not `RegisteredVehicleID`. In this last case, a cast to `RegisteredVehicleID` on `b` will lead to a class cast exception.

11.8 Exercise Solution

```
import java.util.ArrayList;

public class MyStack<E> {

    private ArrayList<E> elements = new ArrayList();

    public void push(E element) {
        elements.add(element);
    }

    public E pop() {
        return elements.remove(elements.size()-1);
    }

    public int size() {
        return elements.size();
    }

    public String ToString(){
        return elements.stream()
            .map(Object::toString)
            .collect(Collectors.joining(","));
    }
}
```

```
public class Main {
    public static void main (String[] args) {
        MyStack<String> stack = new MyStack<>();
        stack.push("Moto Guzzi V8");
        stack.push("Fiat 124 Abarth Rally");
        stack.push("Aprilia RS 125");

        String veichicle = stack.pop();

        System.out.println("veichicle =" + veichicle);
        System.out.println("stack =" + stack);
    }
}
```

12.6 Exercise Solution

```
public abstract class VehicleID {

    protected final String make;
    protected final String model;
    ...
    public boolean equals(Object obj){
        if (this == obj)
            return true;
        if (obj == null)
            return false;
        if (!(obj.getClass() == VehicleID.class))
            return false;

        VehicleID other = (VehicleID) obj;

        if(this.make != null && !this.make.equals(other.make))
            return false;
        if(this.model != null && !this.model.equals(other.model))
            return false;
        if( (this.make == null && other.make != null) ||
            (this.model == null && other.model != null))
            return false;

        return true;
    }

    public int hashCode(){
        return 31 * (make != null ? make.hashCode() : 1)
                   + (model != null ? model.hashCode() : 1);
    }

    public String toString(){
        return "VehicleID [Make: " + make + ", Model:" +model+"]";
    }
}
```



```
public class BikeID extends VehicleID {

    protected final String chassisNumber;
    ...
    public boolean equals(Object obj){
        if(!super.equals(obj))
            return false;

        if (!(obj.getClass() == BikeID.class))
            return false;

        BikeID other = (BikeID) obj;

        if(this.chassisNumber != null &&
            !this.chassisNumber.equals(other.chassisNumber))
            return false;

        if( this.chassisNumber == null && other.chassisNumber!= null)
            return false;

        return true;
    }

    public int hashCode(){
        return super.hashCode() + chassisNumber != null ?
                                   chassisNumber.hashCode() : 1;
    }

    public String toString(){
        return "BikeID [Make: " + make + ", Model:" + model
               + ", ChassisNumber"+ chassisNumber +"]";
    }
}
```

The hashCode and equals functions call the super.hashCode in order to consider also the hashCode of inherited fields. Instead the toString method explicitly uses the inherited fields.

```
public class EBikeID extends BikeID {
    private String batteryNumber;
    ...
    public boolean equals(Object obj){
        if(!super.equals(obj))
            return false;

        if (!(obj.getClass() == EBikeID.class))
            return false;

        EBikeID other = (EBikeID) obj;

        if(this.batteryNumber!= null &&
            !this.batteryNumber.equals(other.batteryNumber))
            return false;

        if( this.batteryNumber == null && other.batteryNumber!= null)
            return false;

        return true;
    }

    public int hashCode(){
        return super.hashCode() + batteryNumber!= null ?
                                   batteryNumber.hashCode() : 1;
    }

    public String toString(){
        return "EBikeID [Make: " + make + ", Model:" + model
              + ", ChassisNumber"+ chassisNumber
              + ", BatteryNumber"+ batteryNumber +"]";
    }
}
```

13.6 Exercise Solution

```
public abstract class RegisteredVehicleID implements
Comparable<RegisteredVehicleID>{

    protected final String make;
    protected final String model;
    protected String licensePlate;
    protected final String VIN;

    public RegisteredVehicleID (String make, String model,
                                String licensePlate, String VIN){

        this.make = make;
        this.model = model;
        this.licensePlate = licensePlate;
        this.VIN = VIN;
    }

    ...

    public int compareTo(RegisteredVehicleID other){
        if(!this.equals(other)){
            int res = compareStringField(this.licensePlate, other.licensePlate);
            if(res != 0)
                return res;
            res = compareStringField(this.make, other.make);
            if(res != 0)
                return res;
            res = compareStringField(this.model, other.model);
            if(res != 0)
                return res;
            res = compareStringField(this.VIN, other.VIN);
            if(res != 0)
                return res;
        }
        return 0;
    }

    private int compareStringField(String thisField, String otherField) {
        int res = checkNullness(thisField, otherField);
        if(res != 0)
            return res;
        if( thisField != null) {
            res = thisField.length() - otherField.length();
            if(res != 0)
```

```
        return res;
        res = thisField.compareTo(otherField);
        if( res != 0)
            return res;
    }
    return 0;
}

private int checkNullness(String thisField, String otherField) {
    if((thisField != null && otherField == null))
        return -1;
    if((thisField == null && otherField != null))
        return 1;
    return 0;
}
}
```

```
public void main(String[] args) {

    RegisteredVehicleID v1 = new CarID("Fiat", "500", "AA999ZZ", "0...1");
    RegisteredVehicleID v2 = new CarID("Ferrari", "F40", "BC111XX", "0...2");
    RegisteredVehicleID v3 = new MotorcycleID("Ducati", "Monster", "AA99999", "0...3");
    RegisteredVehicleID v4 = new CarID("Bugatti", "Veyron", "BA111XP", "0...4");
    RegisteredVehicleID v5 = new MopedID("Piaggio", "Vespa", "11222X", "0...5");

    SortedSet<RegisteredVehicleID> sortedSet= new TreeSet<>();

    sortedSet.add(v1);
    sortedSet.add(v2);
    sortedSet.add(v3);
    sortedSet.add(v4);
    sortedSet.add(v5);

    Iterator<RegisteredVehicleID> iterator = sortedSet.iterator();

    String toPrint = "";
    while(iterator.hasNext()){
        RegisteredVehicleID next = iterator.next();
        toPrint += next .getLicensePlate() + (iterator.hasNext() ? "," : "");
    }

    System.out.println(toPrint);
}
```

14.12 Exercise Solution

```
package it.unive.dais.pol.id;

public class Main {

    public static void main (String[] args) {

        OptionParser p = new OptionParser(args);

        ...

        String make = p.getMake();
        String model = p.getModel();
        String licensePlate = p.getLicensePlate();
        String VIN = p.getVIN();

        try {
            CarID carId = new CarID(make, model, licensePlate, VIN);
            CarInfo car = retrieveCarInfoFromDatabase(id);
            ...
        } catch ( InvalidLicensePlate e) {
            e.printStackTrace();
        } catch (SQLException e){
            ...
        }

    }

    ...
}
```

```
package it.unive.dais.pol.id;

public final class CarID extends RegisteredVehicleID {

    public CarID(String make, String model, String licensePlate, String VIN) throws
                                                    InvalidLicensePlate {
        super(make, model, licensePlate, VIN);
    }

    ...
}
```

```
package it.unive.dais.pol.id;

public abstract class RegisteredVehicleID {

    protected final String make;
    protected final String model;
    protected String licensePlate;
    protected final String VIN;

    public RegisteredVehicleID (String make, String model,
                                String licensePlate, String VIN) throws InvalidLicensePlate {
        this.make = make;
        this.model = model;
        this.licensePlate = licensePlate;
        if(!isValidLicensePlate())
            throw new InvalidLicensePlate(licensePlate);
        this.VIN = VIN;
    }

    protected abstract boolean isValidLicensePlate();

    ...
}
```

```
package it.unive.dais.pol.id;

public class InvalidLicensePlate extends IllegalArgumentException {

    public InvalidLicensePlate(String invalidPlate){
        super("The following license plate is not valid: "+ "\"" +invalidPlate+ "\"");
    }
}
```

15.7 Exercise Solution

```
@Retention(RetentionPolicy.RUNTIME)
@Target(ElementType.METHOD)
public @interface CarID {
    String model() default "none";
    String make() default "none";
}
```

```
public class CarFactory {

    @CarID(model = "Giulietta", make="Alfa Romeo")
    public static Car createAlfaRomeoGiulietta(){
        return new Car(0, new FuelType("diesel", 1.4, 0.02));
    }

    @CarID()
    public static Car createCar(){
        return new Car(0, new FuelType("diesel", 1.0, 0.01));
    }
}
```

16.8 Exercise Solution


```
import java.lang.reflect.Constructor;
import java.lang.reflect.Field;
import java.lang.reflect.InvocationTargetException;
import java.lang.reflect.Method;
import java.lang.reflect.Parameter;

public class Main{

    public static void main(String[] args) throws InvocationTargetException,
                                                InstantiationException, IllegalAccessException,
                                                NoSuchMethodException, SecurityException,
                                                IllegalArgumentException,
                                                NoSuchFieldException {

        CarID id = createInstanceOfCarID(args[1], args[2], args[3]);

        String model = callGetModelByReflection(id);

        if (model.equals("Lamborghini"))
            System.out.println("Nice choice!");

        if (callIsValidLicensePlateByReflection(id))
            System.out.println("The license plate " +
                                getLicensePlateFieldByReflection(id) + "is valid!");
    }

    static CarID createInstanceOfCarID(String make, String model, String licensePlate)
        throws InvocationTargetException, InstantiationException,
        IllegalAccessException {
        external: for (Constructor constructor : CarID.class.getConstructors()) {
            constructor.setAccessible(true);
            Parameter[] parameters = constructor.getParameters();
            Object[] parametersValues = new Object[parameters.length];

            parametersValues[0] = make;
            parametersValues[1] = model;
            parametersValues[2] = licensePlate;

            return (CarID) constructor.newInstance(parametersValues);
        }
        return null;
    }
}
```

```
static String callGetModelByReflection(CarID id) throws NoSuchMethodException,  
SecurityException, IllegalAccessException, IllegalArgumentException,  
InvocationTargetException {  
  
    Class classCarID = id.getClass();  
    Class classRegVehicleID = classCarID.getSuperclass();  
    Method getModel = classRegVehicleID.getDeclaredMethod("getModel");  
    return (String) getModel.invoke(id, null);  
}  
  
static boolean callIsValidLicensePlateByReflection(CarID id) throws  
    NoSuchMethodException, SecurityException, IllegalAccessException,  
    IllegalArgumentException, InvocationTargetException {  
  
    Class classCarID = id.getClass();  
    Method isValidLicensePlate = classCarID  
        .getDeclaredMethod("isValidLicensePlate");  
    return ((Boolean) isValidLicensePlate.invoke(id, null)).booleanValue();  
}  
  
static String getLicensePlateFieldByReflection(CarID id) throws  
    NoSuchFieldException, SecurityException, IllegalArgumentException,  
    IllegalAccessException {  
  
    Class classCarID = id.getClass();  
    Class classRegVehicleID = classCarID.getSuperclass();  
    Field licensePlate = classRegVehicleID.getDeclaredField("licensePlate");  
    return (String) licensePlate.get(id);  
}  
}
```

Note in the method *callIsValidLicensePlateByReflection* is called from the CarID class, but can be also called from the RegisteredVehicleID class. In these case, the same result would be obtained because the implementation of the CarID class would have to be taken.

18.6 Exercise Solution

```
public class Main {  
  
    public static void main(String[] args) {  
  
        CarIDModel model = new CarIDModel();  
  
        CarIDView view = new CarIDView();  
  
        CarIDController controller = new CarIDController(model, view);  
  
        Scanner sc = new Scanner(System.in);  
  
        String action = null;  
  
        do {  
  
            controller.updateView(.ASK_ACTION);  
  
            action = sc.nextLine();  
  
            if(action.toUpperCase().equals("ADD")){  
                controller.updateView(ASK_MAKE);  
                String make = sc.nextLine();  
  
                controller.updateView(ASK_CAR_MODEL);  
                String carModel = sc.nextLine();  
  
                controller.updateView.ASK_LICENSE_PLATE);  
                String licensePlate = sc.nextLine();  
  
                controller.updateView(ASK_VIN);  
                String VIN = sc.nextLine();  
  
                controller.addCarID(make, carModel, licensePlate, VIN);  
  
            }else if(action.toUpperCase().equals("REMOVE")){  
                controller.updateView(View.ASK_VIN);  
                String VIN = sc.nextLine();  
  
                controller.removeCarID(VIN);  
            }  
  
            controller.updateView(View.CAR_IDS);  
  
        } while(action != null && !action.toUpperCase().equals("EXIT"));  
    }  
}
```

```
public class CarIDController {

    private CarIDModel model;
    private CarIDView view;

    public CarIDController(CarIDModel model, CarIDView view){
        this.model = model;
        this.view = view;
    }

    public void removeCarID(String VIN){
        model.remove(VIN);
    }

    public void addCarID(String make, String carModel,
                        String licensePlate, String VIN){
        model.add(VIN, new CarID(make, carModel, licensePlate, VIN));
    }

    public void updateView(CarIDView.PrintableView pv){
        switch(pv){
            case ASK_ACTION :
                view.printAskForAction();
                break;
            case ASK_MAKE :
                view.printAskForMake();
                break;
            case ASK_CAR_MODEL :
                view.printAskForModel();
                break;
            case ASK_LICENSE_PLATE :
                view.printAskForLicensePlate();
                break;
            case ASK_VIN :
                view.printAskForVIN();
                break;
            case CAR_IDS :
                view.printCarIDsDetails(model.getCarIDs());
        }
    }
}
```

```
public class CarIDModel {  
  
    private Map<String, CarID> map;  
  
    public CarIDModel(){  
        this.map = new HashMap<>();  
    }  
  
    public void remove(String VIN){  
        map.remove(VIN);  
    }  
  
    public void add(String VIN, CarID carId){  
        map.put(VIN, carId);  
    }  
  
    public Collection<CarID> getCarIDs(){  
        return map.values();  
    }  
}
```

```
public class CarID {  
  
    private final String make;  
    private final String model;  
    private final String licensePlate;  
    private final String VIN;  
  
    public CarID(String make, String model, String licensePlate, String VIN){  
        this.make = make;  
        this.model = model;  
        this.licensePlate = licensePlate;  
        this.VIN = VIN;  
    }  
  
    ...  
}
```

```
public class CarIDView {

    public void printAskForAction(){
        System.out.println("Please type one of the following"
                           + " actions ADD, REMOVE, EXIT:");
    }

    public void printAskForMake(){
        System.out.println("Please type the make of the car:");
    }

    public void printAskForModel(){
        System.out.println("Please type the model of the car:");
    }

    public void printAskForLicensePlate(){
        System.out.println("Please type the license plate of the car:");
    }

    public void printAskForVIN(){
        System.out.println("Please type the VIN of the car:");
    }

    public void printCarIDsDetails(Collection<CarID> carIDs){
        for(CarID id : carIDs)
            System.out.println(carIDs.getVIN() + ","
                               + id.getLicensePlate() + ","
                               + id.getMake() + "," + id.getModel() + ",");
    }

    public enum PrintableView {
        ASK_ACTION,
        ASK_MAKE,
        ASK_CAR_MODEL,
        ASK_LICENSE_PLATE,
        ASK_VIN,
        ASK_CAR_IDS
    }
}
```