A gentle intoduction to programming

But secretly an elaborate crash course

Stijn Dejongh

Digital Release

Contents

I Part One

1 I am 12 and what is this? 7
1.1 Introductory stuff 7
1.1.1 About the author(s) 7
1.2 What is coding about? 8
1.2.1 Why would you care about coding? 8
1.2.2 The big secrets 8

2 Setting up your tools 9
2.1 Getting your coding gear 9

II Actually doing stuff

3 The absolute basics 13
3.1 Outsourcing some basic practise 13
3.2 Your first Java program 13
3.2.1 Remarks 15
3.3 Diving in 16
3.4 Down the rabbit hole we go 18

4 Learning from examples 21
4.1 Why examples? 21

4.2 Writing a quizing application **21**
4.2.1 Tackling the problem 22
4.2.2 Writing the code 22
4.3 Writing a very simple shelf stacking application **35**
4.3.1 The goal of this 35
4.3.2 Writing the program 35

Part One

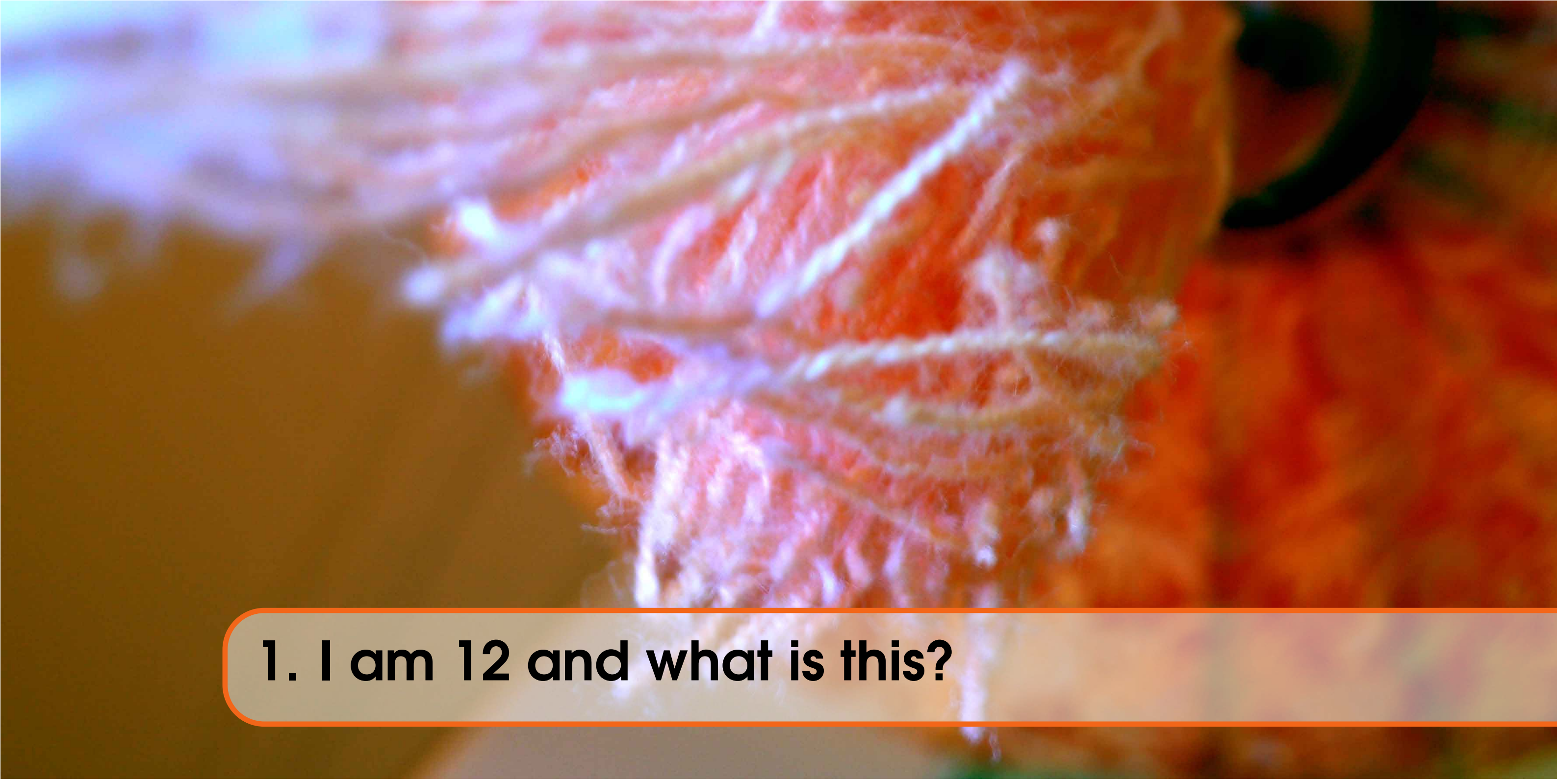
1 I am 12 and what is this? 7

1.1 Introductory stuff

1.2 What is coding about?

2 Setting up your tools 9

2.1 Getting your coding gear



1. I am 12 and what is this?

1.1 Introductory stuff

1.1.1 About the author(s)

Why would you write a course on coding?

So what is possessing me to write this “Introduction to coding”-book? Mostly because I care about programming, haven’t writen a longer text since my master thesis. Also, sometimes I am bored. These being all very good, and valid, reasons, the main reason is obviously: because coding is fun and more people should give it a go. I’ve tried to get friends, family, and aquitances excited about programming. Most of these attempts failed. The main reason for this is that most of the resources out there are either very simple and brain numbing, or very technical and overly complicated. The idea of this course is to get you coding fast, and explain things as they become relevant. Most text books begin with a very detailed overview of computer architecture, bytecode, and other technical information. My belief is that if you are interested in this stuff, there are books out there that explain these things profoundly, and much better than I can ever hope to do.

Ok cool, so who are you?

My name is Stijn. I am a Belgian twenty-seven year old guy. I like music, coffee, and being entertained. Apart from this, I am secretly very lazy. I have had a college education in informatics. This is basically computer science with a different name. I work as a professional developer. This is a fancy name for ‘’code (and related stuff) monkey”. I work with Java on a daily basis in a professional environment. That is to say, I know a thing or two about coding. But I also have a lot to learn.

Disclamer

I am writing this as I go along, thinking stuff up as I go. If you find this course to be unstructured, please tell me what to do to improve it.

R I will probably add pictures at some point, bear with me.

R I am going to use the word “entity” a lot. This can just as well be replaced with “thing”, but I like to sound intelligent.

R Code listings in the book contain red “return”-signs. Those would not actually appear in your code, they are put there if the line of code was split in order to fit on the page.

1.2 What is coding about?

1.2.1 Why would you care about coding?

There are two reason you are reading this text. Either you know me personally, and want to me a favour. Or you want to learn about programming. Or both. Either way, there a a few reasons why I think writing code is cool. First of all, the general idea of “making something do work for you so you don’t have to do it” is appealing to me. Apart from that, I like solving puzzles and figuring out how to make machines do the things I want them to do. And after a while, you learn to appreciate how ingeniouly designed the platform you are working on is, and how much there is to learn. If you like a challenge, and learning new skills, coding might just be your cup of tea.

1.2.2 The big secrets

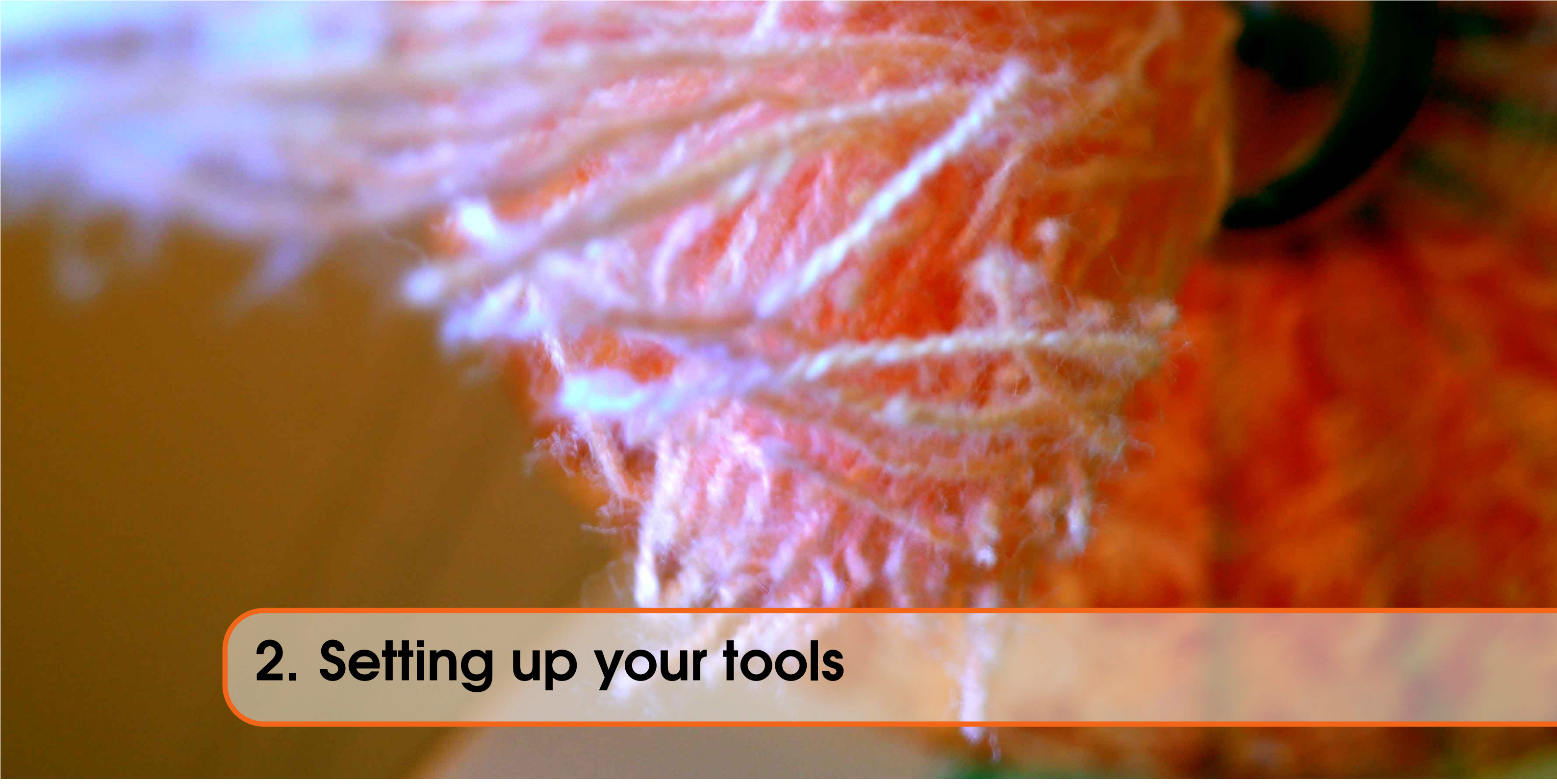
There are two important rules to coding. These *secrets* are why programming (and computers in general) are a mystery to most people. So without further ado, here they are:

Theorem 1.2.1 Coding is your way of telling the computer what to do. The computer will do exactly what you tell it to do, *and nothing more*.

Theorem 1.2.2 The best solution to any computer problem is googling it, and have a vague idea about what keywords to use.

The first rule is very important. Often in your coding life you will wonder why the computer is not doing what you want it to do. The simple reason is: because you didn’t explain your intentions clear enough. A computer will follow instructions. If those instructions are not well-written, it will behave in ways you did not anticipate. A lot of your development time will be spent browsing through your own code, trying to figure out where you made a mistake. Embrace this process.

Sometimes you have no idea what you are supposed to be doing to solve an issue. At first you try some things, and if that doesn’t work, you start looking for other people’s solution to your problem. The main advantage that “computer people” have over “non-computer people” is that they know some technical terms to describe their problem. Building this vocabulary usually comes with time. If you spend enough time tinkering with software and remembering some keywords, you’ll be able to describe your problem in a clearer way. And the more you google for answers, the more terms you learn. Those terms can be used later for new searches, and so you gradually become a “computer stuff expert”.



2. Setting up your tools

2.1 Getting your coding gear

Before we start the actual course, you need to know about the tools you can use to write code. The bare basics are

1. something to write text with
2. something to run your code
3. access to the internet or other ways too look for answers to your questions (if you are old-school: textbooks)

And that is about it.

That being said, there is a lot of specialized software that make the text writing easier for you. These things are called “development environments”, or *IDE*s. The advantage of using a more specialized tool is that it does a lot of your work for you. The best example is the auto-complete feature most environments have. You start writing a few letters, press the auto-complete, and pick the word you are trying to write. This approach will help you by not having your code fail because you made a typo. Since you are probably new to coding, start with a simple cookie-cutter IDE that is free to use. If you become more serious in your programming, consider paying for a more advanced tool. Or beter yet, have your employer buy it for you.

So for now, download and install the “Eclipse IDE”. You will also need to install something to run your code. We will use the Java environment. If you go to the Eclipse website (https://eclipse.org/), you should get the option to download Eclipse along with the Java JVM. If you are wondering what JVM stands for, just google it. It is interesting to know, but it makes no actual difference at this point. It is the name of the “something that runs your code”.

II

Actually doing stuff

3 The absolute basics 13
3.1 Outsourcing some basic practise
3.2 Your first Java program
3.3 Diving in
3.4 Down the rabbit hole we go

4 Learning from examples 21
4.1 Why examples?
4.2 Writing a quizing application
4.3 Writing a very simple shelf stacking application

3. The absolute basics

3.1 Outsourcing some basic practise

As the goal of this text is to teach you how to think like a programmer, I will skimp on the basic introductory lessons. If you are entirly new to programming, it is advisable to take the first two chapters of a basic syntax tutorial. The following sections will refresh this information, but will not give a detail-oriented approach to learning them for the first time. I have found the java language tutorial on Codecademy to be a good introduction to the basic types of value you have at your disposal in Java. To get a head start, take the first two tutorial chapters over at: *https://www.codecademy.com/learn/learn-java*. This should give you a running start for the rest of the course.

3.2 Your first Java program

At this point, you should have Eclipse and Java installed. If not, do that first. We'll start by doing something very simple and explaining the things that you are writing. One of the most basic programs is one that simply says "Hello!" to the person that runs it. This is often called a "Hello world!"-program. You will learn some basic commands and coding structure in order to achieve this. When picking up a new programming language, people usually start by writing this kind of program.

Fire up your Eclipse, and create a new java project. Give it any name you like. First of you need to know a thing or two about the Java language. Code is organised in so-called "classes". These classes are a collection of "functions" (also called "methods"). Functions are a set of instructions that will be executed one by one by the computer. There are various kinds of functions and classes, and we'll explain those as they become relevant. What is important to know is that classes are organized into "packages". These can be seen as folders containing the class-files. Create a new

Java class in your project and give it a quirky name. Your IDE will now generate something that looks like this:

```
public class YourName{

}
```

This is a bare-bones Java class. You will notice the keywords “public” and “class”. In coding you use keywords to tell the computer what the things you write are, this is called “syntax”. In this case we are saying we are creating a class called “YourName” with the “class YourName” expression. The “public” keyword means that the class can be accessed from anywhere in the program. The oposite to “public” is “private”. A private entity can only be accessed from within the class that defines it. There is a secritive third option, by using no keyword at all. This third option is “package private”. The name pretty much gives it away: you can access these entities from anywhere within the same package. These three keywords are known as “scope” keywords, as they define the scope or “reach” of the entities they describe. The curly brackets are used to state where something starts and ends.

R That wasn’t too bad, was it. We learned some technical terms, and got about three words written down. Let’s keep it up!

As we explained before, classes are a collection of functions, which are a collection of instructions for the computer. Now in order to make our prgram do anything, we need to write a function to contain our commands. This looks as follows:

```
public class YourName{

    public static void main(String[] parameters) {

    }
}
```

No worries, as before I’ll explain what those keywords mean. Function declarations always have the following format:

```
<scope keyword> <optional keywords> <return type> <function name>(<parameters>)
```

. You already know what “public” means, and now you know these scope keywords can also be used on functions. The “static” keyword is an optional modifier used to declare that something is a part of the class itself, and not of an instance of it. I know this explanation makes little sence right now, but it will be clear after we discuss “Objects”. For now that’s a bit too much of information, so just remember the “static’ keyword’ binds an entity to a class. We’ll elaborate on this later on, don’t worry. Next up is “void”. As you can guess from the function format given earlier, this is the “return type”. A function is a set of instruction that are executed when called from another place in the code. Sometimes you want your function to give some information back to the piece of code that called it. This is called “returning” something. The easiest example of this is a function that adds a few numbers. This function would return the result of the addition. In this case, we just want our function to write something to the screen, so it should not return anything. The “void” keywords tells the program that our function does exactly that: *return nothing*. The next piece of our function declaration is the function’s name, in this case “name”. In general you want to give your functions a meaningful name that describes what it does. If you see a piece of code calling a function named “doSomething”, you are none the wiser of what is happening. So be nice to future you, and give your functions name that actually make sense. Last but not least, we have the parameters. If the “return type” is the output of a function, the paramaters are it’s input. You hand the function the things that it needs to do it’s job. In our example of the addition function, you would give your functions the numbers it needs to add together. In the code snippet above, our parameters are “String[] parameters”. The square brackets indicate an “array”. This is a fancy word for a collection of entities. “String” in programming means “text”. So in this case we made a function, called main, that can take a few words and will give nothing back.

R In case why you are wondering why we give our main function text to begin with: Computer programs used to be executed from the command line, meaning people would type the program name and hit enter to start running it. In most cases you want to give your program an input, such as where to find a certain file. You would do this by typing the location of the file after the name of the program. This is the text parameter that we pass to our main function. For most functions, you have freedom of choise over the parameters you want it to take in. A main function is a bit special as it is the access point of your program, so we need to follow the conventions of such a function.

So we got our class, and we got our main function. Now we just need to make it actually do something. In our first program, we are just going to make it say “Hello”. This makes the program look like this:

Listing 3.1: YourName.java

```
    public class YourName{

    public static void main(String[] parameters) {
        System.out.println(‘‘Well, hello there beautiful!’’);
    }
}
```

There a few things going on here. This is what is happening: we are providing the String “Well, hello there beautiful!” as a parameter to the “println()” function of the class “System”. The “out” is an entity contained in the “System” class. The ‘’System” class is special, because it is a predefined class. This means you don’t have to write it yourself, and it is usable in any Java program you write. Hooray for free stuff! *System* is a way for your program to talk to the machine you run your code on. In this case we want it to display a piece of text on the screen. The “out” is an entity within the system, that lets you write things on the command prompt. This is the most basic way you can let your program talk to the user. “println()” stands for “print a line”.

Let’s go! Now to run our program. Simply right-click inside your class file, select “Run as”, and choose “Java application”. Alternativly, click on the green play button at the top of your screen. You will see a line of text apear in the Console on the botom of your screen. You just made yourself your first (and friendly) program.

3.2.1 Remarks

R As you were typing your code, you probably noticed your IDE trying to help you by giving you suggestions. You can let the IDE help you complete your words by pressing “Ctrl” and “Space” on your keyboard.

3.3 Diving in

R Now that you've chosen the red pill, I'll take you deeper into the rabbit hole. Most programming introductions now gently easy you into more and more of the language syntax, and then start a fairly complicated section about application structuring, and "The Object Oriented Way". Since understanding Objects are crucial to writing Java code, I'm just going to skip to that point and start explaining it. The actual syntax will be explained as we go along.

Without further ado, this is the difference between Classes and Objects:

Theorem 3.3.1 Classes are definitions. Objects are the concrete entities following those definitions.

If this statement confuses you, allow me to explain it with an example which should clarify the statement. A class is the definition of an idea. Let's say we have a class called "Car". It contains the definition of what a car is composed of, such as: "it has wheels, a number of seats, and a maximum speed." And object would be an actual car. For example we would have an actual car, called fancyNewMercedes, which has four wheels, two seats, and a maximum speed of 280 km/h. We have another actual car, called crappyOldVolvo, which has three wheels, one seat, and a maximum speed of 55 km/h. Both the "fancyNewMercedes" and "crappyOldVolvo" are said to be objects of the class Car. Objects of a class are commonly refered to as "instances of a class". To create such an object is called "to instantiate a class". Now that we have an understanding of what objects and classes are, let's take a look at the basic code that we would write if we were to program our example.

Listing 3.2: Car.java

```java
public class Car{
    int numberOfWheels;
    int numberOfSeats;
    double maximumSpeedInKilometersPerHour;

    public Car(int wheels, int seats, double
      ↪ maximumSpeedInKilometersPerHour) {
        this.numberOfWheels = wheels;
        this.numberOfSeats = seats;
        this.maximumSpeedInKilometersPerHour =
          ↪ maximumSpeedInKilometersPerHour;
    }
}
```

Listing 3.3: YourName.java

```java
public class YourName{

public static void main(String[] parameters) {
    System.out.println("Well, hello there beautiful!");

    Car fancyNewMercedes = new Car(4, 2, 280.0);
    Car crappyOldVolvo = new Car(3, 1, 55.0);
}
}
```

Now to take a closer look at the code, and explain the different things that are going on. let's start with the "Car" class. As I described in the example, a car is made of several properties (wheels, seats and a maximum speed). When writing this in code, we give our Car class so called "attributes". Each object of this class will have the same attributes, but with different values. Thinking back to the example: A car has wheels. Our mercedes has four, and our volvo has three. But they both have wheels. The format for defining an attribute is:

```
<scope keyword> <optional keywords> <attribute type> <attribute name>
```

Attributes too have a scope. In the example, all of our attributes are package-private. We see three attributes in our Car class: "numberOfWheels", "numberOfSeats", and "maximumSpeedInKilometersPerHour". The first two attributes are of type "int". This is the class our attributes are an object of. The "int" class is a predefined class to represent integers (both positive and negative whole numbers) . Our attribute "maximumSpeedInKilometersPerHour" is of type "double". This is another predefined class. The "double" class is used to represent decimal numbers.

Our Car class also contains that wierd "Car" function that has no return type, and accepts parameters that resemble our attributes. This is the function that is used to make an actual object of this class. Such a function is called a "constructor". You can see it being used in the main function of the "YourName" class. Constructors are called using the keyword "new" followed by the class of which we want to make an object. When calling the constructor, you feed it the parameters it needs to fill in it's attributes. Going back to the constructor of our Car class definition, we see these parameters being put inside the attributes. Also the keyword "this" shows up. The "this" keyword is used to reference the attributes of a class, but it is not required. Simply writing the following code would work just as well.

```
...
numberOfWheels = wheels;
...
```

I included the "this" keyword to show you it's primary use. As you can see in the code, both the attribute and parameter "maximumSpeedInKilometersPerHour" have the same name. Using "this" here makes it clear to the computer which one we mean.

R I tried my best to explain this as simple as possible, but there is no shame in not getting it the first time you read this. Go over it again if you feel it is still unclear, and take a good look at the example code. If you still don't understand after, give me a shout, and I'll change the course so it is more clear.

3.4 Down the rabbit hole we go

You made it this far, congratulations. I will continue this course in small sections, which can be seen as individual “lessons”. Each one will go a bit further on what we have already learned. And bit by bit, we will explore more of the Java programming language.

In this section we will build upon the cars example and flesh it out a bit further. What we have now is a very basic representation of a car. We are also able to create mutltiple car objects, each different from the others. Now our model doesn’t really do anything. We are able to keep making more and more cars, but they don’t actually do anything. Let’s change that. We will combine the last two lessons by making a bunch of cars, and having them tell us who they are and what is so special about them.

Let’s just jump into the code that does this for us.

Listing 3.4: Car.java

```java
public class Car{
    int numberOfWheels;
    int numberOfSeats;
    double maximumSpeedInKilometersPerHour;

    public Car(int wheels, int seats, double maximumSpeedInKilometersPerHour) {
        this.numberOfWheels = wheels;
        this.numberOfSeats = seats;
        this.maximumSpeedInKilometersPerHour = maximumSpeedInKilometersPerHour;
    }

    public String toString() {
    return "I am a car, and I have " + numberOfWheels  + " wheels, " +  numberOfSeats + "  seats, and I can drive up to " + maximumSpeedInKilometersPerHour + "kilometers per hour.";
    }
}
```

Listing 3.5: YourName.java

```java
import java.util.ArrayList;
import java.util.List;

public class YourName{

public static void main(String[] parameters) {
    System.out.println("Well, hello there beautiful!");

    Car fancyNewMercedes = new Car(4, 2, 280.0);
    Car crappyOldVolvo = new Car(3, 1, 55.0);
```

```
        Car toyCar = new Car(4, 4, 2.5);
        ArrayList<Car> cars = new ArrayList<Car>();
        cars.add(fancyNewMercedes);
        cars.add(crappyOldVolvo);
        cars.add(toyCar);

        System.out.println("Let me brag about my " + cars.size
           ↪ () + "cars!");
        for(int i = 0; i < cars.size(); i++) {
            Car carToShow = cars.get(i);
            System.out.println(carToShow.toString());
        }

        System.out.println("See you later, aligator.");
    }
}
```

First off, we added the “toString()” method to the class Car. This function returns a String, containing some information about the car. Again this function is public, as I believe it should be able to be called from anywhere in our program. The “toString()” function writes some standard text, and adds the values of our attributes to it. It mashes these text fragments together using the “+” operator. An operator is a special type of function, that takes two parameters. The plus sign is shorthand for:

```
    public String concat(String first, String second) {
    ...
    }
```

Shorthands are often called “Syntactic sugar”. Which means it has no effect on how the code is executed, it is just more pleasant to read. Java has a ton of these shorthands, and you will get to know a fair number of them in this course.

Just adding the “toString()” method is fine and all, but it does not really do anything to our program. If you take a look at the “YourName” class, you will see the main method got quite a few lines added to it. What is does now is:

- Say hello
- tell you it has a bunch of cars
- go over all the cars and print their “toString()” output to the screen
- say goodbye

R You probably noticed the “import” lines at the top. These expressions tell the program where to find classes that are not in the same package as the current class. In this case we are using “List” and “ArrayList” from the “java.util” package. As discussed before, packages are just folders containing files of code. The “java.util” package comes with the Java JVM. Other people wrote the code, and you can just use it; how cool! Don’t worry about writing these import lines yourself. Let your IDE do the work for you. Just start typing the name of a class you want to use inside your code, and press the auto- complete key combination. After you select the class you want to use, your IDE will add the import lines if they are needed.

We discussed the creation of the cars in the previous lesson. What is new here is that we put all those cars together into a List, and then go over them one at a time. A List, just like an Array, is a collection of objects. When creating the List, you need to tell the program what kind of objects you are going to put in the List. Hence the type definition inside the "<" and ">" signs. At this point, we have created a new, empy, collection that will one day be used to store cars in. The next lines add cars to the List, by using the "add" function. You see we add all our cars to the list in the next couple of lines. A bit further down you see the "for" keyword. This keyword is used to indicate you want to start a loop. In general you want your loops to start at a certain point, repeat a few actions, and then stop. The format of a for-loop is as follows:

```
for(<starting expression>, <loop expression>, <increment expression>)
```

The starting expression is a command that gets executed at the start of our loop, and is done only once. In this case, we make a new int, called "i" and give it the value zero. This "i" is our counter. It will increase by one, every time the commands in the loop are executed, so it counts how many times we have looped so far. The loop expression is a boolean expression. This is a fancy word for "true/false expression". Here we say: "i is smaller than the size of the list". While this loop expression is true, the loop will keep going. The last expression is the "increment expression", this is a command that will be executed every loop, after the computer execute the commands inside the loop. In our code, we use it to increment "i" by one. The "++" you see, is another shorthand. It means the same as:

```
    i = i + 1;
```

It is just easier to write.

Now to explain what is going on inside our loop. The first line takes an object from the list. This is done by using the "get(int i)" method. This function takes the ith item from our list and returns that. So in our code, we take the car from our list that is in the position equal to the value of our counter. It is important to know that most programming languages start counting items in collections from zero. This means that calling

```
    cars.get(0);
```

will return the *first* element of the list. It also means that the last item in the list is at position "list.size() - 1". This is why our loop stops when "i" is no longer smaller than the size of the list.

R Don't take my word for it, change the loop expression to make the loop run longer than our list has elements. You should see your program fall apart and give you error messages in your console window.

Exercise 3.1 This is a good place to start messing with your code a bit. Play around a bit with the loops, and the things that your car is able to do. Don't be afraid to add new functions. You could add functions and attributes to your cars so you can make them drive for a given amount of time, and tell you how they have driven. You should know enough keywords to do this, and if not, think about the "programmers usually google a lot of stuff" advise from the introduction. You know what to look for now. ■

4. Learning from examples

4.1 Why examples?

At this point, you are familiar with some of the basics of the language. In stead of slowly building more and more disjointed knowledge, we will continue our coding quest by learning from example. I was debating with myself what style to use for the next few lessons. The two options I was considering were "Show people the entire code, and then go over it line by line." and "Make case studies, and talk people through the programming process I used to come to my solution.". I decided the second option teaches you more about tackling a programing problem, and are still easy to follow. The "code dump approach" might scare some people off. In any case, I will make the full source code available together with this text, so that you can take a look at some of the examples and figure things out for yourself if that is what you prefer.

I will structure the following sections by first giving you the assignment. After that I will go through my thought process and start implementing things piece by piece, explaining new concepts as we go along. If you are feeling courageous, take a swing at solving the assignment yourself without looking at my solution. Afterwards, read my way of solving it, and see where our solutions differ and which one you like the most. Do note that I am not claiming my way of programming is the best way to do it. If you find a more understandable opr consise way of writing your code, do so. There are plenty of good books on what "good code" should look like, and I might though on those subjects as we advance through the examples.

4.2 Writing a quizing application

Exercise 4.1 Write a text-based quiz application that shows you a series of questions, and asks you to pick an answer for each one. Every correct answer adds to your score. At the end of the quiz, display the final score in a nicely formatted way. ■

4.2.1 Tackling the problem

This two line assignment is easy to formulate, but a lot harder to solve. The trick is to convert the problem into a "programming model". A "Model" is a term that means "the way you represent things in your program, and how they relate to each other". You can usually make a simple drawing of your model with some words and lines connecting the words. Making a model boils down to identifying the classes and methods you are going to need. More experienced programmers will have a basic model in mind after reading this assignment. If you don't immediatly see a way of making your model, take a look at the nouns in the assignment text. Usually they are good candidates for being written as a class in your applications. In this case, I identify "quiz", "question", and "answer" as the classes I want to use in my application.

The next step is to figure out how my classes are linked together. I usually do this by explaining my model to myself. I would explain my quizing program structure like this: "There is a quiz. This quiz has a number of questions. Each question has a number of possible answers. An answer is either correct or incorrect. My quiz keeps track of the score, and is responsible for interacting with the user.". Next we doublecheck if our solution matches the assignment. In this case, it seems like it does.

4.2.2 Writing the code

So now I make a new project in my IDE, and give it some name. Next I make some packages to include my actual classes. For this quiz, I will divide my classes into "things that are part of the quiz", and "things that interact with the user". I will put the classes that represent the quiz into a package called "model", and the classes that interact with the user in "presentation". So now I have my two packages. Now to start implementing the solution to our problem. I personally think user interaction is a bit icky, so I tend to keep it for last. So I am going to start in my "model" package and create my classes. I add the class "Question", and the class "Answer". I will start with implementing the "Question" class for the simple reason that asking a good question is often more important than knowing all the answers.

Writing our Question class

As you remember from before, I usually explain my code to myself as I go along. And I will continue that practise for each class I write. Doing so helps you keep focussed and sometimes you discover smaller problems that you did not think about when you designed your general model. So we have a Question. It should have some text containing the actual question and some possible answers. This gives us the following code:

```java
package model;

import java.util.List;

public class Question {

    private List<Answer> possibleAnswers;
    private String question;
}
```

Notice I have made these attributes private. I don’t want anything to be able to change the text or mess with the answers after a Question is created. This also means no one is able to read these attributes. But we want other classes to be able to see what is inside the question. How else are we going to show this text to the user? The way to do this is by creating so called “getter” functions. These are functions that should return a copy of the class attribute. We return a copy, so that the original attribute object in our Question can not be changed. If we would just hand out the orginal attribute object, any other class can change it’s value. This is important mostly to prevent yourself from messing up your program without realising it, and spending tons of time afterwards to figure out what is going wrong.

R When I say the object itself gets passed along, I actually mean a reference to the memory location where the object is situated is passed. Your program will pass this reference. When you use the reference in another place, it will point to the same physical location of your computer memory. Hence, we can see this as passing “the object itself”. The alternative to this practise is to pass the value of the ojbect, in stead of a reference to the object itself. For a more in-depth and well-written explanation on how this works, look up “pass by reference” and “pass as value”.

```java
package model;

package model;

import java.util.ArrayList;
import java.util.List;

public class Question {

    private List<Answer> possibleAnswers;
    private String question;

    public String getQuestion() {
        return this.question;
    }

    public List<Answer> getPossibleAnswers() {
        return new ArrayList<>(this.possibleAnswers);
    }
}
```

If you look at this code, you will see the “passing a copy” in action in the “ getPossibleAnswers()” method. We create a new ArrayList, and pass the existing one along. The implementation of the ArrayList constructor that accepts another List, will copy all of the contained values into a new List. This is exactly what we want to happen. We call these kind of constructors “copy constructors”. Do note that not all classes have this type of constructor. For some classes, you need to use their “copy()” or “clone()” methods. The best way to figure out which one to use, is to read the documentation of the class you are using, and look for these methods.

You might have also noticed that I just return the String "question" without explicitly copying it to a new object. The reason behind this is that String is a special class. Every time you assign a String to a new variable, it gets copies out of the box. Some other classes have the same behaviour: "int" and "boolean" come to mind.

Now that we have a class containing some attributes, and methods to read them, we should probably give ourselves a way of adding answers to our List. For my implementation, I choose to only allow for one answer to be added at a time. In the same manner as with getters, you don't always want to just store an object you get from somewhere else.This would mean that the class that created the object can change it at any time, and the change will also happen inside our Question. So we will just allow other classes to pass along a text and an indication of whether the answer is correct or not. Our Question class will make a new Answer itself.

Listing 4.1: Question.java

```java
package model;

package model;

import java.util.ArrayList;
import java.util.List;

public class Question {

    private List<Answer> possibleAnswers;
    private String question;

    public String getQuestion() {
        return this.question;
    }

    public List<Answer> getPossibleAnswers() {
        return new ArrayList<>(this.possibleAnswers);
    }

    public void addAnswer(String response, boolean isCorrect)
       ↪ {
        this.possibleAnswers.add(new Answer(response,
           ↪ isCorrect));
    }
}
```

Now the last thing to do is to add a way to make a Question object, by writing a constructor.

```java
package model;

package model;

```

```
import java.util.ArrayList;
import java.util.List;

public class Question {

    private List<Answer> possibleAnswers;
    private String question;

    public Question(String questionToAsk) {
        this.question = questionToAsk;
        this.possibleAnswers = new ArrayList<>();
    }

    public Question(String questionToAsk, List<Answer>
      ↪ answerOptions) {
        this(questionToAsk);
        this.possibleAnswers = new ArrayList<>(answerOptions);
    }

    public String getQuestion() {
        return this.question;
    }

    public List<Answer> getPossibleAnswers() {
        return new ArrayList<>(this.possibleAnswers);
    }

    public void addAnswer(String response, boolean isCorrect)
      ↪ {
        this.possibleAnswers.add(new Answer(response,
          ↪ isCorrect));
    }
}
```

“What? There are two constructors?” Yes there are. In Java you can have multiple methods with the same name, as long as the program can know they are different functions. It knows this by looking at the parameters the different functions accept. If they accept a different set of parameters, they are seen as different functions. Writing multiple functions with the same name, but different parameters is called “overloading a function”.

In this case, we have a simple constructor that only takes a text with the question to ask as a parameter. Our second constructor also takes text, but enables us to also pass a list of answers along (again, we copy the list). The first line of the second constructor calls the first constructor, by using the “this()” function. This just means “call my own constructor”. You usually do this when your alternative constructor accepts some of the same parameters as an existing constructor. This is a way to shortcut writing the same lines of code over and over again, and is the prefered way of overloading.

R The main argument for overloaded functions to call the original one, is that your code stays in one place. This means that if you have to change anything, you should only have to change it in one place.

For our quiz, we'll need to know if a question was answered correctly. We'll write the "isAnswered-Correctly()" method, that returns a boolean, indicating whether we got the question right. This can be easily done by looping over the possible answers, and see if they were correct. This means we are already defining some properties of our next class, "Answer". It should be able to tell us whether or not it was answered correctly. This is absolutely fine, in most programs classes interact, and sometimes you notice that some of them are lacking functions. Feel free to add them in if you need them. Our "Question" will end up looking something like this:

```java
package model;

import java.util.ArrayList;
import java.util.List;

public class Question {

    private List<Answer> possibleAnswers;
    private String question;

    public Question(String questionToAsk) {
        this.question = questionToAsk;
        this.possibleAnswers = new ArrayList<>();
    }

    public Question(String questionToAsk, List<Answer>
      ↪ answerOptions) {
        this(questionToAsk);
        this.possibleAnswers = new ArrayList<>(answerOptions);
    }

    public String getQuestion() {
        return this.question;
    }

    public List<Answer> getPossibleAnswers() {
        return new ArrayList<>(this.possibleAnswers);
    }

    public boolean isAnsweredCorrectly() {
        boolean allAnsweredCorrectly = true;
        for(Answer answer : possibleAnswers) {
            allAnsweredCorrectly = allAnsweredCorrectly &&
              ↪ answer.isAnsweredCorrectly();
        }
        return allAnsweredCorrectly;
    }
    public void addAnswer(String response, boolean isCorrect)
        ↪ {
        this.possibleAnswers.add(new Answer(response,
            ↪ isCorrect));
    }
}
```

Let's disect the "isAnsweredCorrectly()" method. As you can see, it starts by holding a boolean that indicates if everything was answered correctly. This is the boolean we will eventually return to the caller of this method, and can be seen as a "tracker variable". We then start looping over the possible answers, changing the value of our "allAnsweredCorrectly" boolean on every loop. Note that the new value of the boolean is the logical operation "CurrentValue AND whether the answer was correctly filled in". This means that if we have one incorrect answer, our tracker will be "false".

R Your IDE will complain that the method "isAnsweredCorrectly()" does not exist yet, by underlining it with a red dashed line. If you click on the red icon in the sideline, on the same line as the error, the IDE will give you some options to fix this problem. You can choose for the "create method" option, and your missing method will be created automatically. Do note that only the method name and modifiers will be generated, it won't actually do anything useful.

Writing the Answer class

Our answer class should have a few attributes that we know of already. These are:

- text to show to the user
- whether or not the answer is correct

So, our basic implementation will look like this:

```java
package model;

public class Answer {

    private String answerText;
    private boolean isCorrect;

    public Answer(String answerText) {
        this.answerText = answerText;
        this.isCorrect = false;
    }

    public Answer(String answerText, boolean isCorrect) {
        this.answerText = answerText;
        this.isCorrect = isCorrect;
    }

    public String getAnswerText() {
        return answerText;
    }

    public void setAnswerText(String answerText) {
        this.answerText = answerText;
    }

    public boolean isCorrect() {
        return isCorrect;
    }

    public void setCorrect(boolean isCorrect) {
        this.isCorrect = isCorrect;
    }

    public boolean isAnsweredCorrectly() {
        //TODO: auto generated method
        return false;
    }
}
```

You'll notice two constructors again, and a bunch of getters. Do note that getters for boolean values are usualy start with "is" in stead of "get". This makes the code that calls the function a lot easier to read. You'll also notice our " isAnsweredCorrectly()" method is not filled in, and will always return false. So let's do something about that, shall we? The main question to ask is: "when is an answer correctly answered?". This would be when the answer is true (isCorrect) and the user has selected this answer. This tell us we need to add a way of knowing if an answer was selected. The easiest way of doing this, is to add another attribute that holds just that information. Then we can use that attribute to implement the "isAnsweredCorrectly()" method. Our class will now look something like this:

```
package model;

public class Answer {

    private String answerText;
    private boolean isCorrect;
    private boolean isSelected;

    public Answer(String answerText) {
        this.answerText = answerText;
        this.isCorrect = false;
    }

    public Answer(String answerText, boolean isCorrect) {
        this.answerText = answerText;
        this.isCorrect = isCorrect;
    }

    public String getAnswerText() {
        return answerText;
    }
```

```java

    public void setAnswerText(String answerText) {
        this.answerText = answerText;
    }

    public boolean isCorrect() {
        return isCorrect;
    }

    public void setCorrect(boolean isCorrect) {
        this.isCorrect = isCorrect;
    }

    public boolean isSelected() {
        return this.isSelected;
    }

    public void select() {
        this.isSelected = true;
    }

    public void unselect() {
        this.isSelected = false;
    }

    public boolean isAnsweredCorrectly() {
        return (isCorrect() && isSelected()) || (!isCorrect()
            ↪ && !isSelected());
    }
}
```

That is pretty much all we need from the “Answer” class. Not that “ isAnsweredCorrectly()” is true if the answer was correct and selected, or if it was incorrect and not selected.

Bringing it all together with the Quiz class

Our Quiz class will be the class holding our main method, from which everything else is controlled. So let’s start by writing the outline of the class and main method:

```java
package presentation;

import java.util.ArrayList;
import java.util.List;
import java.util.Scanner;

import model.Answer;
import model.Question;

public class SimpleQuizGame {
    private static final int CORRECT_SCORE_REWARD = 5;
    private static final int INCORRECT_SCORE_REWARD = 0;

    public static void main(String[] args) {

    }
}
```

This will look very familiar to you, apart from the weird capitalized attributes. You see that they have a bunch of modifiers in front of them. “private” and “static”, you already now. The “final” keyword means the value of the attribute can not be changed after it has been created. In this case “ CORRECT$_S$*CORE*$_R$*EWARD″hasavalueof*“5″*,andwillalwayshaveavalueof*“5″*whenourprogramisrunning.Attibutes*

Now that we got the basic outline of our main method, let’s start implementing some things. First of all, we’ll need to make some questions to put in the quiz. To make eveything more readable and structured, we will make a separate method that creates a list of questions.

```java
package presentation;

import java.util.ArrayList;
import java.util.List;

import model.Answer;
import model.Question;

public class SimpleQuizGame {

    private static final int CORRECT_SCORE_REWARD = 5;
    private static final int INCORRECT_SCORE_REWARD = 0;

    public static void main(String[] args) {
        int score = 0;
        List<Question> questions = createQuestions();
    }

    private static List<Question> createQuestions() {
        List<Question> questions = new ArrayList<>();
        Question easyQuestion = new Question("What is the
           ↪ first leter of the roman alphabet?");
        easyQuestion.addAnswer("a", true);
        easyQuestion.addAnswer("b", false);
        easyQuestion.addAnswer("c", false);
        easyQuestion.addAnswer("d", false);
        questions.add(easyQuestion);
        return questions;
    }
}
```

The code pretty much speaks for itself here. We create a list of questions, then make a question. We then add answers to our question, indicating whether or not these answers are correct. And when all this is done, we add our question to the list, and finally return the list. Our list contains just one question for now, but it's easy enough to add some of your own if you feel like it.

The next thing we want to do, is ask the user to answer all of our questions. So we will create a loop, and in that loop ask the user for an answer. Again, for readability, we'll put the user interaction in a different method.

```java
package presentation;

import java.util.ArrayList;
import java.util.List;
import java.util.Scanner;

import model.Answer;
import model.Question;

public class SimpleQuizGame {

    private static final int CORRECT_SCORE_REWARD = 5;
    private static final int INCORRECT_SCORE_REWARD = 0;

    public static void main(String[] args) {
        int score = 0;
        List<Question> questions = createQuestions();
        for(Question question : questions) {
            pollForAnswer(question);
        }
        System.out.println("You finished the game.");
    }

    private static List<Question> createQuestions() {
        List<Question> questions = new ArrayList<>();
        Question easyQuestion = new Question("What is the
           ↪ first leter of the roman alphabet?");
        easyQuestion.addAnswer("a", true);
        easyQuestion.addAnswer("b", false);
        easyQuestion.addAnswer("c", false);
        easyQuestion.addAnswer("d", false);
        questions.add(easyQuestion);
        return questions;
    }

    private static void pollForAnswer(Question question) {
        System.out.println("Your question is:");
        System.out.println(question.getQuestion());
        System.out.println("--------------------");
        System.out.println("The possible answers are:");
        List<Answer> possibleAnswers = question.
           ↪ getPossibleAnswers();
        for(int i = 0; i < possibleAnswers.size(); i++) {
            System.out.println("Option " + (i +1) + ": " +
               ↪ possibleAnswers.get(i).getAnswerText());
        }
        System.out.println("Please type in your answer:");
        Scanner scanner = new Scanner(System.in);
        int selectedAnswer = scanner.nextInt() -1;
        possibleAnswers.get(selectedAnswer).select();
        scanner.close();
    }

}
```

Most of this code should be understandable with what you already know. We print out some lines, allerting the user that he is getting asked a question. Then we show the possible answers (again with a for loop). Then we tell the user to type something. Getting a user input is done easily with a "Scanner" object. This opens a feed into our program, passing the user's input to our application. When you open such a feed, always remember to close it aswell. Your IDE will complain when you don't. Leaving a feed (or "input stream") open, will cause computer resources to be occupied. If you do this a lot, the computer will eventually start running a lot slower. The next line is reading a "int" value from the Scanner feed, by using the "nextInt()" method. We then select the answer from our list of answers, and tell it it has been selected using the "select()" method. Then we close our Scanner feed.

R You might have noticed we are combining list positions (indexes) and the ordering humans use. Java starts counting from zero, humans start counting from one. So we add one to the list position when printing them to the screen, and we subtract one from the chosen answer when we fetch the answer from our list.

We are almost there. We just need to keep track of the score, and show that to the user at the end of the game. Adding all of this in gives us our completed Quiz class:

```
package presentation;

import java.util.ArrayList;
import java.util.List;
import java.util.Scanner;

import model.Answer;
import model.Question;

public class SimpleQuizGame {

    private static final int CORRECT_SCORE_REWARD = 5;
    private static final int INCORRECT_SCORE_REWARD = 0;
```

```java
    public static void main(String[] args) {
        int score = 0;
        List<Question> questions = createQuestions();
        for(Question question : questions) {
            pollForAnswer(question);
            score +=  question.isAnsweredCorrectly()?
               ↪ CORRECT_SCORE_REWARD :INCORRECT_SCORE_REWARD;
        }
        System.out.println("You finished the game.");
        System.out.println("Your final score is:");
        printScore(score);

    }

    private static List<Question> createQuestions() {
        List<Question> questions = new ArrayList<>();
        // TODO Create some questions and answers here
        Question easyQuestion = new Question("What is the
           ↪ first leter of the roman alphabet?");
        easyQuestion.addAnswer("a", true);
        easyQuestion.addAnswer("b", false);
        easyQuestion.addAnswer("c", false);
        easyQuestion.addAnswer("d", false);
        questions.add(easyQuestion);
        return questions;
    }

    private static void pollForAnswer(Question question) {
        System.out.println("Your question is:");
        System.out.println(question.getQuestion());
        System.out.println("--------------------");
        System.out.println("The possible answers are:");
        List<Answer> possibleAnswers = question.
           ↪ getPossibleAnswers();
        for(int i = 0; i < possibleAnswers.size(); i++) {
            System.out.println("Option " + (i +1) + ": " +
               ↪ possibleAnswers.get(i).getAnswerText());
        }
        System.out.println("Please type in your answer:");
        Scanner scanner = new Scanner(System.in);
        int selectedAnswer = scanner.nextInt() -1;
        possibleAnswers.get(selectedAnswer).select();
        scanner.close();
    }

    private static void printScore(int score) {
        String scoreDisplayText = "| " + score + " |";
        String dashes = "";
        for(; dashes.length() < scoreDisplayText.length();
           ↪ dashes = dashes + "-");
        System.out.println(dashes);
        System.out.println(scoreDisplayText);
        System.out.println(dashes);
    }

}
```

I purposfully used some cool features of Java, just to show them to you. Let's start with this line:

```
score +=  question.isAnsweredCorrectly()? CORRECT_SCORE_REWARD
   ↪  :INCORRECT_SCORE_REWARD;
```

An expression like this is shorthand for the good old "IF-THEN-ELSE" construction. The format goes like this:

```
<boolean expression>? <value if true> :<value if false>
```

So what we do here is either add zero or five to the score, depending on if the question was answered correctly.

The next shorthand I used is in these lines:

```
        String dashes = "";
        for(; dashes.length() < scoreDisplayText.length();
           ↪ dashes = dashes + "-");
```

If you remember the format for the "for loop", you'll see that we have an empty initial value expression. The loop expression states that the String "dashes" must be shorter than the length of the "scoreDisplayText". And every loop, we add a "-" to the dashes String. So what this will do is make a String of "-" characters, that is exactly as long as the "scoreDisplayText" String. Perfect for our "fancy formatting" assignment.

And there we are! Our application is done. Go ahead and run it, to see what it does.

Exercise 4.2 As you might have noticed, that application crashes when you enter something that is not a number. Working with user input generally means you will have to check if the input matches what you expect. I thought adding this into this case study was a bit overkill. You can probably think of an easy way to check if the input your program receives matches what you expect: a number that is lower than or equal to the amount of possible answers. Go ahead and build in some sort of safe guard against bad user input. ■

4.3 Writing a very simple shelf stacking application

4.3.1 The goal of this

Honesty

I'll be honest here. The idea behind this example was not so much an application with a real world use, but an attempt at a concrete exampl eof why you would ever need *inheritance* in an actual Java program. I came up with a quite simple and clean program that illustrates this, and then made up a use case for the program. Forgive me.

Definition of the program

Now that we are on the same page, let me explain what our goal is for this *stacking application*. You might have a personal library or storage room with a lot of things in them. If you don't visit them often, or if the shelf space is extremely large, you might forget what you have in your possesion. The program we will be writing is a first step towards an inventory system, that allows you to keep track of your possessions. In this course we will limit ourselves to a simple program that creates a bunch of shelves, and put some items on them. In the end we will show the user what is stored on the shelves in plain text.

4.3.2 Writing the program

Code Structure

In previous sections I have discussed packages and how they can be used to organize your code. We will do just this for our inventory application. First of all, we will have classes representing the *problem domain*. This is a fancy way of saying that these are classes representing actual things that are described in our problem definition. For instance a *shelf* would be one of those problem domain classes. I usually put classes like these in a package called *model*. That is because these classes are a model of the problem we are trying to solve. Our other classes will have the task of working with these model classes and making them do stuff. So we need a package to contain the class (or classes) that make our program do things. I will call this package *runner*. This runner package will contain the class with our main method.

R Feel free to structure your own code as you see fit, and give the packages whatever name that seems to fit best for you. When you are writing code for yourself, you only need to worry about *future you*. So pick names that make sense to you now, and that you feel will make sense to you later. After a while you will develop your own structuring system that you can keep using. This will make it a lot easier for you to navigate your own code. When you are working on a project with multiple people, you generaly want to discuss how you will be naming packages and classes, and how they will be structured.

The classes we need

As before, first we need to figure out what our model should look like. From the problem description above, you might have identified the same base classes as I have. I chose to make the classes *Shelf* and *StorableItem*. The *Shelf* class will represent the physical shelf itself, and it will be able to contain items. The *StorableItem* will be the class representing things we can put on the shelf. I realize that *StorableItem* is a very generic name for this. We will make more concrete items to place on the shelves later on in the example, when we talk about the actual *inherritence structure* that we will use. Don't worry too much about what that means for now, it will be come clear in the next paragraphs. Next to these model classes, we will also make a class to contain our main method. Let's callthat class *ShelfStacker*.

R By now, I believe it is no longer needed to hold your hand every step of the way while we develop our code. In this lesson, I will dump the finished code on you all at once, and discuss some of it's features more in detail. If you have read and practised the previous lessons, you should be able to understand most of what is going on. In case you don't, be sure not to feel bad about it. See this as an exercise in figuring out code. You will often come across code someone else wrote, trying to figure out what it means. If you feel certain pieces of code should be explained more in depth, feel free to drop me a line, and I will update the course.

Onwards and upwards we go! Let's take a look at the Shelf class.

Listing 4.2: Shelf.java

```java
package be.doji.course.inheritance.model;

import java.util.ArrayList;
import java.util.List;

/**
 * Created by Doji on 24/06/2017.
 */
public class Shelf {
    private List<StorableItem> itemsOnShelf;
    private int maximumAmountOfItemsOnShelf;

    public Shelf(int shelfSize) {
        this.itemsOnShelf = new ArrayList<>();
        this.maximumAmountOfItemsOnShelf = shelfSize;
    }

    public void addItem(StorableItem itemToStore) {
        itemsOnShelf.add(itemToStore);
    }

    public List<StorableItem> getItemsOnShelf() {
        return new ArrayList<>(itemsOnShelf);
    }

    public String toString() {
        String description = "This is a shelf containing: \n";
        description += getShelvesDescription();
        return description;

    }

    private String getShelvesDescription() {
        if (this.getItemsOnShelf().isEmpty()) {
            return "nothing at all \n";
        } else {
            String description = "";
            for (StorableItem item : itemsOnShelf) {
                description += item.toString() + "\n";
            }
            return description;
        }
    }
}
```

Nothing new to see here, we made a simple class that holds a few *StorableItems* in a List. Our Shelf also has an integer defining how many things we can put on there. The *toString()* method just tells anyone calling it what is going on on this shelf. It will return a String with a descriptive text explaining what is on the shelf. This method calls the *getShelvesDescription()* method. The shelves description method in it's turn will either tell us the shelf is empty, or return the description of all the items on the shelf.

Now that we have our Shelf class in place, we should make sure we have something to put on them. Enter the *StorableItem* class. The class will be a very simple representation of a thing we can put on a shelf. It will have a name, and a weight. It will also have a *toString()* method that returns a formatted version of the contents of these fields, so it can be used by the shelf. For this example, we don't need to go into more detail.

Listing 4.3: StorableItem.java

```java
package be.doji.course.inheritance.model;

/**
 * Created by Doji on 24/06/2017.
 */
public abstract class StorableItem {

    private String name;
    private int weightInGrams;
    private static final int DEFAULT_WEIGHT_IN_GRAMS = 250;

    public StorableItem(String name) {
        this(name, DEFAULT_WEIGHT_IN_GRAMS);
    }

    public StorableItem(String name, int weightInGrams) {
        this.name = name;
        this.weightInGrams = weightInGrams;
    }

    public String getName() {
        return name;
    }

    public void setName(String name) {
        this.name = name;
    }

    public int getWeight() {
        return this.weightInGrams;
    }

    public void setWeight(int weight) {
        this.weightInGrams = weight;
    }

    protected abstract String getPrefixName();

    public String toString() {
        return "item [" + getName() + "] { " + getPrefixName()
           ↪ + " } with weight: " + getWeight();
    }
}
```

There's a few cool things going on here. You might have noticed there are two constructors. One takes only a name, the other takes a name and a weight. Having two methods with the same name and a different set of arguments in a class is called *method overloading*. It enables you (or anyone else using your code) to call the version of the method that fits their intentions best. In this example we do not always want to explicitly specify how much something weighs. That's why there is a class variable called *DEFAULT_WEIGHT_IN_GRAMS*. We will use this value in the cases no weight was given to the constructor. You might have guessed that the first constructor is calling the second one. This is the prefered way of providing overloaded methods with the same basic functionality. You call the base version of the method from the other ones, so that you don't have to write the same code over an over again. The benefit of this is that if you have to change something in the base functionality, you don't have to change the code in multiple places. The default weight has a name in capitol letters. This is not required but, as a convention in the Java programming comunity, *constants* are usually written in all caps. This value is a constant because it will never be able to change while the program is running. The *final* keyword tells the computer that once a value has been assigned to this variable, it is never allowed to be changed.

The second thing that is out of the ordinary is the *abstract* keyword in the class definition. This is a way of saying: *you can not make object of this class*. You might wonder why you would ever want to write a class that you can not instantiate (this means: make an object of it). This is where the *inheritance* comes in. inheritance in Java is similar to the concept in the real world. A class inherrits all attributes and non-private functions of it's parent. This means we can make other classes that inherrit from the *StorableItem* class and have the same basic functionality. This concept is one of the corner stones of Object Oriented Programming. We will write some classes that inherrit from the *StorableItem* class soon.

The *abstract* keyword in the *getPrefixName()* method tells Java that all classes that inherrit from this class are required to have a function of this name in them. This means you can say "All classes that are inherriting from StorableItem, have to be able to give us a bit of text informing us who they are". Another example would be an abstract method called " makeNoise()" in an " Animal" abstract class. Each type of animal will make a different sound, but they will all be able to make noise.

Now let's make some concrete classes representing objects we can store on the shelves. As discussed previously, these classes will inherrit from the *StorableItem* class.

Listing 4.4: Book.java

```java
package be.doji.course.inheritance.model;

/**
 * Created by Doji on 24/06/2017.
 */
public class Book extends StorableItem {

    public Book(String name) {
        super(name);
    }

    public Book(String name, int weightInGrams) {
        super(name, weightInGrams);
    }

    @Override protected String getPrefixName() {
        return "BOOK";
    }
}
```

And another one:

Listing 4.5: Plushie.java

```java
package be.doji.course.inheritance.model;

/**
 * Created by Doji on 24/06/2017.
 */
public class Plushie extends StorableItem {

    public Plushie(String name) {
        super(name);
    }

    public Plushie(String name, int weightInGrams) {
        super(name, weightInGrams);
    }

    @Override protected String getPrefixName() {
        return "PLUSH";
    }
}
```

You can see the *extends* keyword in the class definition. This tells Java that our class inherrits from another one. In this case, both *Plushie* and *Book* inherrit from *StorableItem*. Another way of saying this is: “Book and Plushie are subclasses of StorableItem”. The classes themselves contain little else than constructors, as they get most of their functionality from their parent. You will also notice that they both have a *getPrefixName()* method that returns a code indicating what type of storable item they are. This is required, because we added the abstract method in our StorableItem class. The @*Override* indicates the method replaces a method with the same signature in the parent class. It is not required to add this as it doesn't change any functionality, but it makes your code a lot more readable.

All we need now is our main method to see what our model is able to do. This is the main class I wrote:

```java
package be.doji.course.inheritance.runner;

import be.doji.course.inheritance.model.Book;
import be.doji.course.inheritance.model.Plushie;
import be.doji.course.inheritance.model.Shelf;

import java.util.ArrayList;
import java.util.List;

/**
 * Created by Doji on 24/06/2017.
 */
public class ShelfStacker {

    private static List<Shelf> shelves = new ArrayList<>();
    private static final int DEFAULT_SHELF_SIZE = 4;

    public static void main(String[] args) {
        System.out.println("Building some shelves for you...")
            ↪ ;
        makeShelves(4);
        System.out.println("Shelves built!");
        System.out.println("Putting stuff on the shelves...");
        putStuffOnTheShelves();

        System.out.println("The shelves now look like this: ")
            ↪ ;
        for(Shelf shelf : shelves) {
            System.out.println(shelf.toString());
        }
    }

    private static void putStuffOnTheShelves() {
        Book javaTutorial = new Book("A Gentle introduction to
            ↪  Java");
        Plushie pikachu = new Plushie("Pikachu wearing a fancy
            ↪  hat");

        Shelf topShelf = shelves.get(0);
        topShelf.addItem(pikachu);
        topShelf.addItem(javaTutorial);
    }

    private static void makeShelves(int amountOfShelves) {
        for(int i = 0; i < amountOfShelves; i++) {
            shelves.add(new Shelf(DEFAULT_SHELF_SIZE));
        }
    }
}
```

The method makes some shelves, and puts some things on there. Let's take a look at some interesting details here. We create a new Book and a new Plushie just like we would do with any normal classes that are not part of an inheritance structure. The really cool thing becomes apparent when we look at the *Shelf* class, and it's *addItem()* method.

```
public void addItem(StorableItem itemToStore) {
        itemsOnShelf.add(itemToStore);
    }
```

This very simple method is able to add both books and plushies to the shelf, even if it does not explicitly accept these type of objects. This is the power of inheritance. We specified that any *StorableObject* can be put on a shelf. The shelf does not care whether it is actually a Book or a Plushie. It just knows it is something that can be stored on a shelf.

R This should give you a basic and concrete understanding of what inheritance is, and how you can use it in actual applications. Thinking back of our quizing application, we could inheritance to make a multitude of different types of questions that will just fit into the existing program. Using inheritance has two major advantages: Firstly, it greatly simplifies your code by allowing you to write things just once. Secondly, it makes your application extensible. In our stacking example, it is really easy to add more subclasses of the *StorableItem* class.

Exercise 4.3 Add some more subclasses of the *StorableItem* class, and add them to your shelves. You can also make the Shelf class check whether or not the shelf is already full before accepting a new thing to be added to it, and telling you there is no more room if it is full. ■