Chapter 7

Exercise 1

Look at the unfinished *Building* class below.

1) Encapsulate the variables so that they can only be accessed from other classes via accessor methods. Make sure the *availableSpace* is always a number larger than 0 (square meters).
2) Fill in the constructor so that the provided variables will be assigned to the corresponding instance variables. Use the *this* keyword where appropriate.
3) Overload the constructor to allow for the creation of *Building* instances for buildings of which the address is unknown. The user should then only provide availableSpace. In this case set the *address* to “Unknown”.
4) Override the *toString()* method that gives a textual representation of a *Building* instance, for example: “This is a building with 82 square meters of office space located at Naamsestraat 69, 3000 Leuven.”

```
public class Building {

	int availableSpace;
	String address;

	public Building(int availableSpace, String address){

	}

}
```

Exercise 2

Look at the 2 unfinished classes below. The *Vehicle* class describes vehicles by a *topSpeed* and *mass*. It provides a method to calculate the value of a Vehicle. The *Truck* class describes a specific type of *Vehicle* that - next to a top speed and mass - also has a maximum payload.

1) Change the code so that *Truck* inherits from *Vehicle*.
2) Fill in the unfinished constructor in the *Truck* class.
3) Make sure that no instances from the *Vehicle* class can be created.
4) Make sure that direct access to the *topSpeed* and *mass* variables is limited to classes in the same package and to subclasses of *Vehicle*.
5) In *Truck*, create a *calculateValue* method that overrides the *calculateValue* method from *Vehicle*. The value of a truck is calculated the same way as the value of vehicle, but multiplied by (*maxPayload/1000)*. Ideally, whenever you will change the way you calculate the value of a Vehicle, the calculateValue of Truck does not need to be changed.

```
public class Vehicle {
```

```
	double topSpeed;
	double mass;

	public Vehicle(double topSpeed, double mass){
		this.setTopSpeed(topSpeed);
		this.setMass(mass);
	}

	public double calculateValue(){
		return 1000*this.topSpeed/(this.mass*0.1);
	}

	public double getTopSpeed() {
		return topSpeed;
	}

	public void setTopSpeed(double topSpeed) {
		this.topSpeed = topSpeed;
	}

	public double getMass() {
		return mass;
	}

	public void setMass(double mass) {
		this.mass = mass;
	}
}

public class Truck{

	private double maxPayload;

	public Truck(double topSpeed, double mass, double maxPayload) {

	}

	public double getMaxPayload(){
		return this.maxPayload;
	}

	public void setMaxPayload(double newMaxPayload){
		this.maxPayload = newMaxPayload;
	}
}
```

Exercise 3

1) Create an interface called *PersonalizedPrint* that forces classes to implement a function called *prettyPrint*.
2) Create an interface called *CanSing* that forces classes to implement a function called *sing*.
3) Implement the interfaces *PersonalizedPrint* and *CanSing* in a new class called *Person* that has variables *name* (String) and *gender* (String). Implement the *prettyPrint* method so that it returns a ‘fancy’ String

representation of the *Person* object. Implement the *sing* method to print some lyrics of your favorite song to the standard output.

Exercise 4

In this exercise, we will expand on the *CanSing* interface from Exercise 3.

1) Create a new interface called *CanMakeNoises* with functions *makeNoiseOne* (void) and *makeNoiseTwo* (void).
2) Since you have to be able to make noises in order to sing, make sure that all classes that implement *CanSing* have to implement *CanMakeNoises.*
3) Change the *Person* class from Exercise 3 to implement the updated *CanSing* interface.

Exercise 5

In this exercise, you will create a more elaborate hierarchy yourself.

1) Create 4 classes with the following attributes. Make sure to encapsulate these attributes and to provide at least 1 constructor per class that asks for all the attributes:
    a. Person
        i. name (String)
        ii. gender (String)
    b. Employee, inherits from Person
        i. employeeID (int)
    c. SalesPerson, inherits from Employee
        i. itemsSold (int)
    d. Programmer, inherits from Employee
        i. favoriteProgrammingLanguage (String)
        ii. linesOfCodeWritten (int)
2) Make sure that no instances of the *Employee* class can be created.
3) Force all subclasses of *Employee* to have a *calculateSalary* method. If we would create a new subclass of *Employee* (e.g. Secretary, CEO, …), we would be forced to also create a *calculateSalary* method for these classes.
    a. The salary of a SalesPerson is calculated as: *itemsSold*10*
    b. The salary of a Programmer is calculated as: *10.000 + linesOfCodeWritten* and is doubled if the programmer's favorite programming language is *"Java"*.

Exercise 6

Imagine you are working on a new game which lets players tend their own virtual farm, which has both Animals and Plants. Start by creating some interfaces which the different game elements will implement (some suggested methods are given):

- Growable
    - grow(), isFullyGrown()

- Feedable
    - feed(), isHungry()

Now you need some classes which implement your interfaces

- abstract Animal
    - Cow extends Animal
    - Bee extends Animal
    - Dog extends Animal
- abstract Plant
    - Carrot extends Plant
    - Rose extends Plant

Think about which classes implement which interfaces, this is your own choice. Fill in the class appropriately, based on the interfaces it implements.

Think about which variables and methods belong in the superclass, which in the subclass, and where Overriding is necessary.

Create a new class MiniFarm. Each MiniFarm should have a name and a farmer (this can be a String or a Farmer class instance). You should be able to keep track of the Animals and Plants on the farm. (You may want to look into ArrayList for this; it is similar to an array, but the size can change by adding and removing elements.)

Create a main method to play with your farm.

Extra: You may wish to buy and/or sell Animals and Plants. In which classes would you add price variables and sell methods?

Exercise 7: specific exercise on abstract classes versus interfaces

We have seen several concepts that are important to object-oriented programming, focusing heavily on inheritance. Two of these concepts are abstract classes and interfaces. The distinction between when to use these two concepts may be less easily understood. Java's Tutorials section explains the difference: https://docs.oracle.com/javase/tutorial/java/IandI/abstract.html

Abstract classes are similar to interfaces. You cannot instantiate them, and they may contain a mix of methods declared with or without an implementation. However, with abstract classes, you can declare fields that are not static and final, and define public, protected, and private concrete methods. With interfaces, all fields are automatically public, static, and final, and all methods that you declare or define (as default methods) are public. In addition, you can extend only one class, whether or not it is abstract, whereas you can implement any number of interfaces.

Interfaces are often considered to be the real drivers of polymorphism in Java. An interface allows multiple inheritance in Java, as a class can implement multiple interfaces (but can only extend one other class). As such, each interface can specify specific methods to implement, rather than one interface specifying all methods that a class should implement. Hence, you should use interfaces to define an application programming contract (i.e. a blueprint) to which other programmers – for example, in a project that requires you to collaborate with multiple people - have to fully adhere and implement. An interface defines what the name of each functions, how many arguments the function takes, and what the return type of each function is. Interfaces hence force each programmer to code against the API contract, i.e. the classes being implemented must respect the interface, while providing sufficient freedom when it comes to the actual implementation of each method.

A good exercise would be to thoroughly study, and implement for yourself as an exercise, a variation on the GraphicObject example in the Java Tutorials: https://docs.oracle.com/javase/tutorial/java/IandI/abstract.html

The main reason to modify the example in the Java Tutorials is so that we can actually draw something to screen, before having seen any of the graphical objects, using simple System.out.print statements for the purpose of this exercise.

Exercise: Assume that we want to implement multiple graphical objects that all extend an abstract GraphicalObject class (i.e. no objects of type GraphicalObject may be created), which in turn implements an interface Drawable, that contains the methods draw() and resize(int newX, int newY), i.e. two methods with a void return type. All graphical objects must be able to be moved to a new position on the canvas by implementing a method moveTo(int newX, int newY) compared to the origin of the canvas (which we will consider to be the console at the bottom of the Eclipse application). Note that the moveTo method is not required by (i.e. part of) the Drawable interface. You can implement as many subclasses of GraphicalObject as you want, but the easiest ones to draw to console will be Rectangle, Square and Line. Write a suitable main object to create objects of these different classes and to make use of the methods that are implemented for these objects.

Additional exercises?

There are many opportunities to practice object-oriented programming, as you can basically create a class for everything you see in everyday life. During the practical sessions, I've mentioned a couple of times that you can think of any type of application and try to implement it using object-oriented programming in Java. For example, you can implement a university classroom reservation

application. A university has many different classrooms, and they all serve a purpose, i.e. there should not be Room objects, but rather meeting rooms, computer classrooms, rooms that house administrative personnel or technical personnel. Each room also has a number of office chairs (of potentially different types), potentially different computers/laptops, …

There are a couple of exercises to be found in the official documentation of Oracle, and this on different topics of programming in Java. An exercise on inheritance can be found here: https://docs.oracle.com/javase/tutorial/java/IandI/QandE/inherit-questions.html

Many exercises can be found online through a simple Google search. Be careful though, as there are many poor solutions that will misguide you. Just because the code that you find seems to work (not a good criterion), doesn't mean the underlying code is properly designed. Your best bet is to find code from university courses from around the world. Many universities have ".edu" in their website address, which is one way of telling whether the material you find is used in an actual class/course.