Chapter 1

Introduction to object-oriented programming

1.1 Classes

1.1.1 Exercise 0

Translate the Python-program below to Java or C#:

```
result = ""
for i in range(0,9):
    for j in range(0,i):
        result += "*"
    result += "\n"
print(result)
```

1.1.2 Exercise 1

Write a program that draws a smiley on the console (just like in INFDEV02-1).

1.1.3 Exercise 2

Write an example of Python code that would cause a type error in Java/C#

1.1.4 Exercise 3

Make a static function that sums all numbers between two inputs read from the console and prints the result

1.1.5 Exercise 4

Given all semantic and typing rules in the slides, write down in plain English or Dutch

1.1.6 Exercise 5

Make an `Interval` class that:

- takes two integers, `start` and `end`, as its constructor parameters
- has a `Sum` method that returns the sum of all numbers between `start` and `end`
- has a `Product` method that returns the product of all numbers between `start` and `end`

1.1.7 Exercise 6

Make a class `IntArrayOpperations` that:

- takes an array of integers, as its constructor parameter
- has a `Sum` method that returns the sum of all numbers in the array
- has a `Product` method that returns the product of all numbers in the array

1.1.8 Exercise 7

A `Counter` with the following body:

- With a `count` integer attribute;
- With an empty (parameterless) constructor;
- With a method `Reset`;
- With a method `Tick`;
- (**Advanced**) With a static method/overloaded operator `Plus` which adds two counters into one;
- (**Advanced**) With a method `OnTarget` that takes as input a lambda function which will be fired when the counter reaches a given count.

1.2 Arrays

1.2.1 Exercise 8

Make a class `UserStory` that contains:

- 2 variables:
  - hours
  - description
- getters and setters for those fields
- a toString method
- a main method that instantiates 3 UserStory-objects

Write a class `Sprint` that contains:

- 1 variable: an array of UserStories
- methods:
  - totalHours() which sums all the hours in the UserStories
  - a toString method
  - a main method that instantiates a Sprint-object and fills it with
  - addUserStory which adds a UserStory to the array of Userstories

1.3 Constructors and Collections

1.3.1 Exercise 9

We will revisit the `UserStory`- and `Sprint`-classes and extend them. In this exercise you will apply knowledge about constructors, collections

1. To both classes add a constructor which sets their instance variables.
2. In the `Sprint` class: Instead of an array, use an ArrayList to store UserStories.
3. `Sprints` usually have a startdate (17th of february) and a duration (for example: 1 week). Add these variables to the class. Try and google which datatypes (classes) are suitable for storing dates and durations.

4. Also, add getters for the previous variables and update the constructor.
5. `UserStories` have to store their status: Todo, In progress, To verify, Done. Add a variable that can store this.
6. Write methods in the `Sprint`-class that:
   - returns the amount of hours of work still to be done in a sprint.
   - returns the amount of hours already done in a sprint.
   - returns if the current sprint is done
7. (Optional) The datatype you chose for the status (Todo, Done, etc.) is probably a String, right? Readup[1] on Enums, a special type and use an enum to store the status.

[1] https://docs.oracle.com/javase/tutorial/java/javaOO/enum.html

Chapter 2

Reuse through polymorphism

2.1 Interfaces

2.1.1 Exercise 0

- Define an interface `Animal` with at least one method `SaySomething` that takes no arguments and returns `void`
- Define a `Cat` class that implements `Animal`. A cat prints on the console `Miao...` when `SaySomething` is called
- Define a `Dog` class that implements `Animal`. A dog prints on the console `Bao...` when `SaySomething` is called
- Define a `Cow` class that implements `Animal`. A cow prints on the console `Muuu...` when `SaySomething` is called

  Test your program with the following codes:

- The following code `Animal animal1 = new Cat(); animal1.SaySomething();` should print `Miao...`
- The following code `Animal animal2 = new Dog(); animal2.SaySomething();` should print `Bao...`
- The following code `Animal animal3 = new Cow(); animal3.SaySomething();` should print `Muuu...`

2.1.2 Exercise 1

- Make a `Person` interface with methods (or properties with only a getter):

  - Name
  - Surname
  - Age

- Make the `Customer`, `Student`, `Teacher` implementations of `Person`, ensuring that they all get at least three additional methods and attributes over those in `Person`

2.1.3 Exercise 2

- Write a `Vehicle` interface with a method `move` and a method `loadFuel`; `loadFuel` accepts a `Fuel` instance, where `Fuel` is an interface of your writing; `move` returns a boolean which is `true` if there is enough fuel, and `false` otherwise
- Write a concrete class `Car` and a concrete class `Gasoline` that implement, respectively, `Vehicle` and `Fuel`; the `Car` checks that the given fuel is indeed `Gasoline`
- Write a concrete class `Truck` and a concrete class `Diesel` that implement, respectively, `Vehicle` and `Fuel`; the `Truck` checks that the given fuel is indeed `Diesel`
- Write a concrete class `Enterprise` and a concrete class `Dilithium` that implement, respectively, `Vehicle` and `Fuel`; the `Enterprise` checks that the given fuel is indeed `Dilithium`
- Make a program that receives three vehicles, without knowing their concrete type, and moves them (without resorting to conversions) until their fuel is up

2.1.4 Exercise 3

- Make a `ListInt` interface with methods `Length`, `Iterate`, `Map`, `Filter`, and properties (read-only) IsEmpty
- Define the concrete classes `NodeInt` and `EmptyInt` both implementing `ListInt`
- (**Advanced**) Make a `ListInt`, fill it with a series of numbers, increment them all by one (hint: use `Map`), and print them all on the screen (hint: use `Iterate`)

2.1.5 Exercise 4

Basic:

- Write an `IStateMachine` interface with a method `Update` and attribute `Done`, where `Update` takes a `float` number and returns `void`, and `Done` is read-only and of type `bool`
- Write a concrete class `Wait` that implements `IStateMachine`; A `Wait` takes an initial time when instantiated and at every update it decreases such amount until it gets all consumed. When the time is totally consumed `Done` becomes `true`
- Write a concrete class `Print` that implements `IStateMachine`; A `Print` takes an initial message when instantiated and after the first update it prints the message and sets `Done` to `true`
- Write a concrete class `Sequence` that implements `IStateMachine`; A `Sequence` takes two `IStateMachine` objects when instantiated and it keeps updating the first state machine until done before start updating the second state machine. `Done` is set to true when both state machines are done
- Test your program with the following code `new Sequence(new Wait(10), new Print("Hello World"))`. Make sure that it prints ”Hello World” after 10 seconds. For this homework use MonoGame so to get the elapsed time for each update call.

Advanced:

- Extend the `IStateMachine` interface with a new method `Reset` that takes no arguments and returns `void`.
- Make a `Repeat` class that implements `IStateMachine`; A `Repeat` takes a state machine when instantiated and at every update it keeps updating the given state machine until it is done. When the given state machine is done it gets reset, so its behavior can start all over again. The Done attribute of `Repeat` is always `false`
- Test your program with the following code `new Repeat(new Sequence(new Wait(10), new Print("Hello World")))` . Make sure that it prints ”Hello World” every 10 seconds, *forever*. For this homework use MonoGame so to get the elapsed time for each update call.

Chapter 3

Architectural and design considerations

3.1 Exercises

- Write an `Event` abstract class or interface with a method `perform`;
- Write a `Timer` class with a method `tick` and a method `reset`; `reset` restarts the timer, while `tick` makes the timer move forward and returns whether or not the target time has been reached; when the timer reaches the target time, then fire the events in the list of timer responses
- Make a `TrafficLight` class which uses timers to implement red, green, and yellow lights;
- (**Advanced**) Rebuild timers, but this time with lambda's instead of our custom `Event`.
- (**Advanced**) Make a `Component` interface;
- (**Advanced**) Make an `Entity` abstract class which houses a list of components;
- (**Advanced**) Write a `Car` class that inherits from `Entity` and which implements all the functionality that you would expect from a car, but with the *Entity-Component* model; you will need to build components for the engine, the wheels, etc. and all that the `Car` class does is make correct use of these components.

No reference solution yet:

- Build an entity-component system where a `Person` is made up of multiple components such as shoes, clothes, make-up, personality, and intelligence (all implemented via appropriate interfaces); the `Person` then performs a few actions, such as doing sports, studying, and socializing through methods: the results of these actions depend on the components of the person so that, for example, doing sports with elegant shoes will have unpleasant results.