Chapter 1

Introduction to object-oriented programming

1.1 Exercises

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

```
1 result = ""
2 for i in range(0,9):
3     for j in range(0,i):
4         result += "*"
5     result += "\n"
6 print(result)
```

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

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

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

5. Write a Java/C# program featuring

   - A `Counter` class;
   - 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.

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

7. Make an `Interval` class that:

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

Chapter 2

Reuse through polymorphism

2.1 Exercises

- 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

No reference solution yet:

- 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`

Chapter 3

Reuse through generics

3.1 Exercises

- (**Advanced**) Make a `List<T>` interface with methods `Length`, `Iterate`, `Map`, and `Filter`
- (**Advanced**) Define the concrete classes `Node<T>` and `Empty<T>` both implementing `List<T>`
- (**Advanced**) Make a `List<Vehicle>`, fill it with a series of concrete vehicles, and make them all move ten times
- Make a generic `Number<N>` abstract class, with methods:
  - `Zero` that returns an `N`
  - `One` that returns an `N`
  - abstract methods `Negate`, that takes an `N` and returns an `N` (for example `Negate(1)` return `-1`) - Plus, Times, DividedBy that all take two N's and returns an N
  - The non-abstract method `Minus` that makes use of `Plus` and `Negate`
  - abstract methods `SmallerThan` and `Equal`, that take two `N`'s and return a `boolean`
  - The non-abstract methods `SmallerOrEqual`, `GreaterThan`, `GreaterOrEqual`, `NotEqual`
- Make a class `IntNumber` that implements `Number<int>`
- Make a class `FloatNumber` that implements `Number<float>`

- Try to make a class `StringNumber` that implements `Number<string`: how far can you come?
- Make the `Interval` class we have seen in the first homework of DEV3 generic with respect to the type of the parameters `l` and `u`; specifically, build a generic class `Interval<N>` which takes as input two `N`'s `l` and `u`, and also an instance of `Number<N>`

Chapter 4

Architectural and design considerations

4.1 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.