TODO

Chapter 1

Exercises

1.1 Functions

- Write the `sumTwo` function that returns the sum between two numbers
- Write the `smallestTwo` function that returns the smallest between two numbers
- Write the `smallestThree` function that returns the smallest between three numbers
- Write the `ageDescription` function that takes as input an age, and returns the description of what can be legally done at that age:
  - If the age is smaller than 3, return "nothing";
  - If the age is between 3 and 12, return "you can shoot semi-automated guns in the USA";
  - If the age is between 12 and 16, return "you can go to high school and think your life is really tough";
  - If the age is between 16 and 25, return "you can go to an HBO and think high school was really easy in comparison";
  - If the age is between 25 and 65, return "you can go to work";
  - If the age is more than 65, return "you can retire now"
- Write the `interval` function that takes as input two integers, and prints all values between them with a loop;
- Write the `intervalRec` function that takes as input two integers, and prints all values between them with recursion;

- Write the `evens` function that takes as input two integers, and prints all even values between them with a loop;
- Write the `evensRec` function that takes as input two integers, and prints all odd values between them with recursion;
- Write the `intervalSum` function that takes as input two integers, and returns the sum of all values between them with a loop;
- Write the `intervalSumRec` function that takes as input two integers, and returns the sum of all values between them with recursion;

1.2 Data structures

- Write the `Point` class with only an `x` and a `y`; make two sample instances of it;
- Write the `Person` class with a `name`, a `surname`, and an `age`; make two sample instances of it;
- Write the `Car` class with a `brand`, `model`, an `engineSize`, and a `productionYear`; make two sample instances of it;

1.2.1 Lists

- Write the `Empty` class with only the `IsEmpty` attribute;
- Write the `Node` class with attributes `IsEmpty`, `Head`, and `Tail`;
- Write the `printAll` function that loops through a list and prints all its values, iteratively;
- Write the `printAllRec` function that loops through a list and prints all its values, recursively;
- Write the `interval` function that creates a list with all values within a given interval `l,u`, iteratively;
- Write the `intervalRec` function that creates a list with all values within a given interval `l,u`, recursively;
- Write the `evens` function that creates a list with all even values within a given interval `l,u`, iteratively;

- Write the `evensRec` function that creates a list with all even values within a given interval `l,u`, recursively;
- Write the `odds` function that creates a list with all odd values within a given interval `l,u`, iteratively;
- Write the `oddsRec` function that creates a list with all odd values within a given interval `l,u`, recursively;
- Write the `multiplicationTable` function that creates a list of lists with the multiplication table; make a `printAllLists` function that prints all elements of a list of lists and use it to print the multiplication table list;

1.3 Understanding stack and heap

No reference solutions yet: TODO

1.4 Methods

No reference solutions yet: TODO

1.5 Higher order functions

No reference solutions yet: TODO

Chapter 2

Assignments

2.1 Exercises for the assignment

2.1.1 Make a car move

As said in the introduction. you will first focus on a relevant part. Assignment 1 is about cars in a city. We already gave you the city, so let's focus on the car.

The relevant concepts are classes and methods (those are links to the slides). In (very, very) short: A class is a blueprint which defines what data can be stored. An object can be instantiated, and can then store the data you want in attributes. Methods can be used to modify that data.

Console version

- Build a `Car` class;
- Add the attribute `Position`, which will be a simple integer;
- Add the `Move` method that increments the position by one;
- Make a test program that initialises the car and moves it ten times; print the position of the car on the console at every step.

Pygame version

- Get a template to kick-start your pygame application. For example from here. Delete code you don't need, so you have a relative clean application to start with.

- Draw – Now add a shape or picture that will represent the car. For now you can put it in the game loop. Play around a bit with its parameters and make shure you understand how you can move the 'car'.
- Now add the `Car` class from part 1.
- Instantiate a car-object.
- From within the gameloop, call the move-method on the car
- Now use the car-position to change where the car is drawn on the screen.

2.1.2 Make a list of cars move

In the assignment you will have more than one car. These multiple cars need to be stored in a way that makes it easy to use all these cars programatically. For this, we've introduced Lists in the lessons. In a list you typically store a lot of objects of the same type (but, it is possible to store any type). In this exercise you will combine the Car-objects with lists.

Console version

- Build the `Node` and `Empty` classes;
- Add the usual attributes `IsEmpty`, `Head`, and `Tail` to the classes;
- Make a test program that
  - initialises a list of cars,
  - moves each of them ten times,
  - print the position of each car on the console at every step.

Pygame version Expand on the pygame application from exercise 1.

- Add a `VerticalPosition` attribute to the car, so that each car has a different vertical position to distinguish it on the screen;
- Use the list of cars you just implemented to draw a pygame screen where various cars move from the left to the right of the screen.

2.1.3 Moving along checkpoints

In the assignments the cars will not move along positions based on coordinates, but positions based on *tiles* are used. In this exercise we will focus on that concept, however we will use a slightly different example: metro's and metro stations. Our metro is always at a metro station. It can travel between 2 neighbouring stations, but we will not store any positions in between 2 stations. With this exercise you will gain a deeper understanding of Classes, Attributes and Lists. The theory and slides from the previous 2 exercises are applied here.

Console version

- Make a `Station` class, which contains a `Position` attribute and a `Name`; for example: Station(Position(10,40), "Kralinse zoom")
- Make a list of stations;
- Make a `Metro` class, which has an attribute `CurrentStation` and a method `Move`.
- The `CurrentStation` will store a reference to a node in the list of stations;
- In the `Metro` class, the `Move` method changes position to the `Tail`, which is the next checkpoint;
- Make a test program that initialises a list of metro's, and moves them until they all reach the final checkpoint; print the position of each metro (which is now a checkpoint) on the console at every step.

Let's reflect on what you did in the part of this exercise. The metro (or car) still stores its position, however that position is now abstracted away into a station object. This has the advantage that you can reason about station "kralinse zoom" for example, instead of (10, 40). Station still stores its position in terms of coordinates (10,40), because we will need that to draw the stations in pygame. Stations are connected to each other by the linked list. a Node's head (containing station "kralingse zoom") is connected to the tail (containing it's neighbour "Capelse brug").

Pygame version

- Draw a pygame screen with the Stations and the Metro's;
- The various metro move from one Station to the other.

2.1.4 Exercise 4 - crossings

Let's continue our voyage by car again. This gives more freedom to travel around the city.

Console version

- Make a `Node2D` class, which contains attributes `Head`, `TailLeft`, `TailRight`, `TailUp`, `TailDown`, and `Final`; this is effectively the same as a list, but with four possible choices for the `Tail` (we call this a **matrix**);
- Make a class Crossing, with attributes: Position and Name.
- Make a series of crossings and put them into `Node2D`'s; For example: Rotterdam CS, Hofplein, Eendrachtsplein, Beurs and Blaak.
- You have to define which two crossings are connected using the tails. For example: Rotterdam CS's tailRight would be Hofplein.
- In the `Car`, the `Position` will now be a reference to a `Node2D` in the matrix of checkpoints;
- In the `Car`, the `Move` method changes position to one of the `Tail`s, which is the next chosen checkpoint; the choice can be random;
- Make a test program that initialises a list of cars, and moves them until they all reach a specific checkpoint with `Final == True`; print the position of each car (which is now a checkpoint) on the console at every step.

Pygame version

- Draw a pygame screen with the checkpoints and the cars;
- The various cars move from one checkpoint to the other (like the cars in the city assignment).

2.1.5 Bikes

Console version

- Make a `Bike` class that has the `Move` method just like the car;
- Dutch `Bike`'s are fast, so the bike moves by two tiles at a time;

- Add a `PrintPosition` method to the `Car` and the `Bike`, which prints where the vehicle is;
- Make a test program that initialises a list contains a mixture of cars and bikes, and moves them until they all reach a specific checkpoint with `Final == True`; print the position of each car or bike (which is now a checkpoint) on the console at every step.

Pygame version

- Add a `Draw` method to the `Car` and the `Bike`, which draws where the vehicle is with the proper texture; the texture is also added as an attribute of both `Car` and `Bike`;
- Draw a pygame screen with the checkpoints, the bikes and the cars;
- The various cars and bikes move from one checkpoint to the other (like the cars and boats in the city assignment).

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

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

6. 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`

7. 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
8. 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.
9. Write a Java/C# program featuring
   - Write a class UserStory, which has:
     - 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, which has:
     - 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

Chapter 2

Reuse through polymorphism

2.1 Exercises

2.1.1 Exercise 1

- 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`

2.1.2 Exercise 2

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

Reuse through generics

3.1 Exercise 1

- (**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>`

3.2 Exercise 2 - based on Chapter 2.1.2

- Write an `IAction<T>` parametric interface with a method `Invoke` that takes no arguments number and returns an object belonging to the type of the `T` (the parameter of the interface)
- Write a `When` class that implements `IStateMachine`; A `When` takes an `IAction` of type `bool` (`IAction<bool>`) when instantiated and at every update it tries to invoke the given `IAction` and only if it returns `true` then set `Done` to `true`
- Test your program with the following code `new Sequence(new When( myRandom), new Run(new Print("Hello World")))`. Make sure that it prints "Hello World" after a random time. For this homework use MonoGame so to get the elapsed time for each update call. `myRandom` is an instance of the following class:

```
1 pulic class MyRandom : IAction<bool>{
2   Random seed = new System.Random();
3   public bool Invoke(){
4     return seed.Random().Next(10) > 7;
5   }
6 }
```

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.