Programmering og Problemløsning
Datalogisk Institut, Københavns Universitet
Arbejdsseddel 10 - individuel opgave

Ken Friis Larsen, Christina Lioma og Jon Sporring

3. december - 11. december.
Afleveringsfrist: lørdag d. 11. december kl. 22.00.

I *object-oriented* programmering bruger vi objekter til at indkapsle små bidder af imperativ tilstand, samt til at hæfte operationer som ændrer ved data sammen med det data som ændres. Dette opgavesæt går ud på at træne jer i at bruge de forskellige sprogkonstruktioner som kendetegner object-oriented programmering. Vi skal se hvordan vi kan definere *klasser*, som vi kan *instansiere objekter* fra, samt hvordan vi kan definere *metoder*, *felter* og *properties* på vores klasser.

Emnerne for denne arbejdsseddel er:

- at kunne definere klasser
- at kunne instantiere objekter
- erklære metoder i klasser
- erklære felter og properties i klasser

Opgaverne er opdelt i øve- og afleveringsopgaver. I denne periode skal I arbejde individuelt med jeres afleveringsopgaver. Regler for gruppe- og individuelle afleveringsopgaver er beskrevet i "'Noter, links, software m.m."'→"'Generel information om opgaver"'.

Øveopgaver (in English)

10ø0 Implement a class `student`, which has 1 property `name` and an empty constructor. When objects of the `student` type are created (instantiated), then the individual name of that student must be given as an argument to the default constructor. Make a program that creates 2 `student` objects and prints the name stored in each object using the "."-notation.

10ø1 Change the class in Exercise 10ø0 such that the value given to the default constructor is stored in a mutable field called `name`. Make 2 methods `getValue` and `setValue`. `getValue` must return the present value of an object's mutable field, and `setValue` must take a name as an

argument and set the object's mutable field to this new value. Make a program that creates 2 `student` objects and prints the name stored in each object using `getValue`. Use `setValue` to change the value of one of the object's mutable fields, and print the object's new field value using `getValue`.

10ø2 Implement a class `Counter`. The class must have two methods:

- `get` which returns the present value of the counter field, and
- `incr` which increases the counter field by 1.

The constructor must make a counter field whose value is initially 0. Write a white-box test class that tests `Counter`.

10ø3 Implement a class `Car` with the following properties:

(a) a specific fuel economy measured in km/liter

(b) a variable amount of fuel in liters in its tank

The fuel economy for a particular `Car` object must be specified as an argument to the constructor, and the initial amount of fuel in the tank should be set to 0.

`Car` objects must have the following methods:

- `addGas`: Add a specific amount of fuel to the car.
- `gasLeft`: Return the present amount of fuel in the car.
- `drive`: Let the car drive a specific length in km, reducing the amount of fuel in the car. If there is too little fuel then cast an exception.

Make a white-box test class `CarTest` to test `Car` and run it.

10ø4 Implement a class `Moth`, which represents a moth that is attracted to light. The moth and the light live in a 2-dimensional coordinate system with axes (x, y), and the light is placed at $(0, 0)$. The moth must have a field for its position in a 2-dimensional coordinate system of floats. Objects of the `Moth` class must have the following methods:

- `moveToLight` which moves the moth in a straight line from its position halfway to the position of the light.
- `getPosition` which returns the moth's current position.

The constructor must accept the initial coordinates of the moth. Make a white-box test class and test the `Moth` class.

10ø5 Write a class `Car` that has the following properties:

- `yearOfModel`: The car's year model.
- `make`: The make of the car.
- `speed`: The car's current speed.

The `Car` class should have a constructor that accepts the car's year model and make as arguments. Set the car's initial speed to 0. The `Car` class should have the following methods:

- `accelerate`: The `accelerate` method should add 5 to the `speed` attribute each time it is called.
- `brake`: The `brake` method should subtract 5 from the `speed` attribute each time it is called.
- `getSpeed`: The `getSpeed` method should return the current speed.

Design a program that instantiates a `Car` object, and then calls the `accelerate` method five times. After each call to the `accelerate` method, get the current speed of the car and display it. Then call the `brake` method five times. After each call to the `brake` method, get the current speed of the car and display it.

Extend class `Car` with the attributes `addGas, gasLeft` from exercise 10ø3, and modify the methods `accelerate` and `brake` so that the amount of gas left is reduced when the car accelerates or brakes. Call `accelerate` and `brake` five times, as above, and after each call display both the current speed and the current amount of gas left.

Test all methods. Create an object instance that you know will not run out of gas, and another object instance that you know will run out of gas and test that your methods `accelerate` and `brake` work properly.

Afleveringsopgaver (in English)

10i0 In a not-so-distant future drones will be used for delivery of groceries. Imagine that the drone traffic has become intense in your area and that you have been asked to decide if drones collide.

We will make the simplifying assumptions that all drones fly at the same altitude, that drones fly with different speeds measured in centimetre/second and in different directions, and that drones fly with constant speed (no acceleration). If two drones are less than 5 meters from each other, then they collide. When a drone reaches its destination, then it lands and can no longer collide with any other drone. Create an implementation file `simulate.fs`, and add to it a `Drone` class with the following properties and methods:

- `Position` (property): returns the drone's current position in (x,y) coordinates.
- `Speed` (property): returns the drone's present speed in centimetre/second.
- `Destination` (property): returns the drone's present destination in (x,y) coordinates. If the drone is not flying, its present position and its destination are the same.
- `Fly` (method): Set the drone's new position after one second flight.
- `AtDestination` (method): Returns `true` or `false` depending on whether the drone has reached its destination or not.

The constructor must take the start position, the destination, and the speed as arguments. All positions and speeds are integers.

Extend your implementation file with a class `Airspace` that contains the drones and as a minimum has the following properties and methods:

- `Drones` (property): The collection of drones instances.

- `DroneDist` (method): The distance between two given drones.
- `FlyDrones` (method): Advance the position of all flying drones in the collection by one second.
- `AddDrone` (method): Add a new drone to the collection of drones.
- `WillCollide` (method): Given a time interval (number of minutes), determine which drones will collide. After two (or more) drones collide they are assumed to fall to the ground and are no longer considered. The method should return a list of pairs of drones that collided in the time interval.

  In the unfortunate event that three drones A, B and C are destined to collide at the same time, the list should contain the pairs (A,B), (A,C) and (B,C). In this case, you may choose between two interpretations: either the three drones collide simultaneously or one of them gets a lucky break and dodges the crash (in which case it shouldn't be in the list of collisions). Clearly document the choice you take.

Write a black-box test class `testSimulate.fsx` that tests both the `Drone` class and the `Airspace` class.

Notice that the required methods and properties are *minimum requirements*; feel free to add methods and properties if you need them.

Krav til afleveringen

Afleveringen skal bestå af:

- en zip-fil, der hedder `10i_<navn>.zip` (f.eks. `10i_jon.zip`)

Zip-filen `10i_<navn>.zip` skal indeholde en og kun en mappe `10i_<navn>`. I den mappe skal der ligge en `src` mappe og filen `README.txt`.

I `src` skal der ligge følgende og kun følgende filer:

- simulate.fs, testSimulate.fsx,

som beskrevet i opgaveteksten. Programmerne skal kunne oversættes med `fsharpc`, og de oversatte filer skal kunne køres med mono. Funktioner skal dokumenteres ifølge dokumentationsstandarden som minimum ved brug af `<summary>`, `<param>` og `<returns>` XML-tagsne. Filen `README.txt` skal ganske kort beskrive, hvordan koden oversættes og køres.

README.txt filen skal også inkludere et eller flere få eksempler på kørsler af hvert program, der illustrerer at og hvordan de virker.

God fornøjelse.