Programmering og Problemløsning
Datalogisk Institut, Københavns Universitet
Arbejdsseddel 10 - gruppeopgave

Jon Sporring and Christina Lioma

5. december – 2. januar.
Afleveringsfrist: onsdag d. 2. januar kl. 22:00

I denne periode skal I arbejde i grupper. Regler for individuelle afleveringsopgaver er beskrevet i "'Noter, links, software m.m."'→"'Generel information om opgaver"'. Formålet er at arbejde med:

- Classes
- Objects
- Methods
- Attributes
- Nedarvning

Opgaverne for denne uge er delt i øve- og afleveringsopgaver.

Øveopgave(r)

10ø.0 Implementer en klasse `Counter`. Objekter (variable) af typen `Counter` skal være tællere, og den skal have 3 metoder (funktioner): Konstruktoreren, som laver en tæller hvis start værdi er 0; `get`, som returnerer tællerens nuværende værdi; `incr`, som øger tællerens værdi med 1. Skriv et unit-test program, som afprøver klassen.

10ø.1 Implementér en klasse `Car` med følgende egenskaber. En bil har en specifik benzin effektivitet (målt i km/liter) og en bestemt mængde benzin i tanken. Benzin effektiviteten for en bil er specificeret med konstruktoren ved oprettelse af et `Car` objekt. Den indledende mængde benzin er 0 liter. Implementer følgende metoder til `Car` klassen:

- `addGas`: Tilføjer en specificeret mængde benzin til bilen.
- `gasLeft`: Returnerer den nuværende mængde benzin i bilen.
- `drive`: Bilen køres en specificeret distance, og bruger tilsvarende benzin. Hvis der ikke er nok benzin på tanken til at køre hele distancen kastes en undtagelse.

Lav også en klasse `CarTest` som tester alle metoder i `Car`.

10ø.2 Implementér en klasse `Moth` som repræsenterer et møl der flyver i en lige linje fra en bestemt position mod et lys således at møllets nye position er halvvejs mellem dets nuværende position og lysets position. En position er to float tal som angiver x og y koordinater. Møllets indledende position gives ved oprettelse af et `Moth` objekt vha. konstruktoren. Implementér metoderne:

- `moveToLight` som bevæger møllet i retning af et lys med specificeret position som beskrevet ovenfor.
- `getPosition` som returnerer møllets nuværende position.

Test alle metoder i `Moth` klassen.

10ø.3 I en ikke-så-fjern fremtid bliver droner massivt brugt til levering af varer købt på nettet. Dronetrafikken er blevet så voldsom i dit område, at du er blevet bedt om at skrive et program som kan afgøre om droner flyver ind i hinanden. Antag at alle droner flyver i samme højde og at 2 droner rammer hinanden hvis der på et givent tidspunkt (kun hele minutter) er mindre end 5 meter imellem dem. Droner flyver med forskellig hastighed (meter/minut) og i forskellige retninger. En drone flyver altid i en lige linje mod sin destination, og når destinationen er nået, lander dronen og kan ikke længere kollidere med andre droner. Ved oprettelse af et `Drone` objekt specificeres start positionen, destinationen og hastigheden. Implementér klassen `Drone` så den som minimum har attributterne og metoderne:

- `position` (attribut) : Angiver dronens position i (x, y) koordinater.
- `speed` (attribut) : Angiver distancen som dronen flyver for hvert minut.
- `destination` (attribut) : Angiver positionen for dronens destination i (x, y) koordinater.
- `fly` (metode) : Beregner dronens nye position efter et minuts flyvning.
- `isFinished` (metode) : Afgør om dronen har nået sin destination eller ej.

og klassen `Airspace` så den som minimum har attributterne og metoderne:

- `drones` (attribut) : En samling droner i luftrummet.
- `droneDist` (metode) : Beregner afstanden mellem to droner.
- `flyDrones` (metode) : Lader et minut passere og opdaterer dronernes positioner tilsvarende.
- `addDrone` (metode) : Tilføjer en ny drone til luftrummet.
- `willCollide` (metode) : Afgør om der sker en eller flere kollisioner indenfor et specificeret tidsinterval givet i hele minutter.

Test alle metoder i begge klasser. Opret en samling `Drone` objekter som du ved ikke vil medføre kollisioner, samt en anden samling som du ved vil medføre kollisioner og test om din `willCollide` metode virker korrekt.

10ø.4 Write a class `Car` that has the following data attributes:

- `yearOfModel` (attribute) : The car's year model.
- `make` (atttribute) : The make of the car.
- `speed` (atttribute) : 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` (method) : The `accelerate` method should add 5 to the `speed` attribute each time it is called.
- `brake` (method) : The `brake` method should subtract 5 from the `speed` attribute each time it is called.
- `getSpeed` (method) : 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 9ø.0, and modify methods `accelerate, break` so that the amount of gas left is reduced when the car accelerates or breaks. Call `accelerate, 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 `accelerate, brake` methods work properly.

10ø.5 Write a `Person` class with data attributes for a person's name, address, and telephone number. Next, write a class named `Customer` that is a subclass of the `Person` class. The `Customer` class should have a data attribute for a customer number and a Boolean data attribute indicating whether the customer wishes to be on a mailing list. Demonstrate an instance of the `Customer` class in a simple program.

10ø.6 (a) Write an `Employee` class that keeps data attributes for the following pieces of information:

- Employee name
- Employee number

Next, write a class named `ProductionWorker` that is a subclass of the `Employee` class. The `ProductionWorker` class should keep data attributes for the following information:

- Shift number (an integer, such as 1 or 2)
- Hourly pay rate

The workday is divided into two shifts: day and night. The shift attribute will hold an integer value representing the shift that the employee works. The day shift is shift 1 and the night shift is shift 2. Write the appropriate methods for each class.

Once you have written the classes, write a program that creates an object of the `ProductionWorker` class and prompts the user to enter data for each of the object's data attributes. Store the data in the object and then use the object's methods to retrieve it and display it on the screen.

(b) Extend the previous exercise as follows: Let a shift supervisor be a salaried employee who supervises a shift. In addition to salary, the shift supervisor earns a yearly bonus when his or her shift meets production goals. Write a `ShiftSupervisor` class that is a subclass of the `Employee` class you created in the previous exercise. The `ShiftSupervisor` class should keep a data attribute for the annual salary and a data attribute for the annual production bonus that a shift supervisor has earned. Demonstrate the class by writing a program that uses a `ShiftSupervisor` object.

(c) **(Extra difficult).** Considering that production during night shifts is reduced by 5% compared to production during day shifts, and that the hourly pay rate during night shifts is double the hourly pay rate during day shifts, compute the best possible worker & shift allocation over the period of 12 months. You need to think how to measure productivity and salary cost, and then find their best tradeoff in the period of 12 months.

10ø.7 Cheetahs, antelopes and wildebeests are among the world's fastest mammals. This exercise asks you to simulate a race between them. You are not asked to simulate their movement on some plane, but only some of the conditions that affect their speed when running a certain distance.

Your base class is called `Animal` and has these attributes:

- The amount of food needed daily (measured in kilograms)
- The weight of the animal (measured in kilograms)
- The maximum speed of the animal (measured in kilometres per hour)
- The current speed of the animal (measured in kilometres per hour)

The `Animal` class should have a primary constructor that takes two arguments: the animal's weight and the animal's maximum speed. The `Animal` class should also have an additional constructor that takes as input only the animal's maximum speed and generates the animal's weight randomly within the range of 70 - 300 kg. The `Animal` class should have two methods:

- The first method should set the current speed of the animal proportionately to its food intake and maximum speed as follows: if the animal eats 100% of the amount of food it needs daily, the animal's current speed should be its maximum speed; if the animal eats 50% of the amount of food it needs daily, the animal's current speed should be 50% of its maximum speed, and so on.
- The second method should set the amount of food needed daily proportionately to the animal's weight as follows: the animal should eat half its own weight in food every day (if the animal weighs 50 kg, it should eat 25kg of food daily).

Create a subclass `Carnivore` that inherits everything from class `Animal`, and modifies the second method as follows: the animal should eat 8% of its own weight in food every day.

Create a subclass `Herbivore` that inherits everything from class `Animal`, and modifies the second method as follows: the animal should eat 40% of its own weight in food every day.

Create an instance of `Carnivore` called `cheetah` and two instances of `Herbivore` called `antelope, wildebeest`. Set their weight and maximum speed to:

- cheetah: 50kg, 114km/hour
- antelope: 50kg, 95km/hour
- wildebeest: 200kg, 80km/hour

Generate a random percentage between 1 - 100% (inclusive) separately for each instance. This random percentage represents the amount of food the animal eats with respect to the amount of food it needs daily. E.g., if you generate the random percentage 50% for the antelope, this means that the antelope will eat 50% of the amount it should have eaten (as decided by the second method).

For each instance, display the random percentage you generated, how much food each animal consumed, how much food it should have consumed, and how long it took for the animal to

cover 10km. Repeat this 3 times (generating different random percentages each time), and declare winner the animal that was fastest on average all three times. If there is a draw, repeat and recompute until there is a clear winner.

Test all methods.

Optional extra: repeat the race without passing as input argument the weight of each animal (i.e. letting the additional constructor generate a different random weight for each instance).

Afleveringsopgave(r)

I Lake Superior på grænsen mellem USA og Canada ligger en øde ø kaldet Isle Royale. Her har man over en lang årrække fulgt populationen af ulve og elge (http://www.isleroyalewolf.org/). Bestanden af de 2 dyrarter er tæt knyttet til hinanden som rov- og byttedyr.

Denne opgave omhandler simulering af populationen af ulve og elge i et lukket miljø. Elge spiser planter og i denne opgave vil vi antage at ulve kun spiser elge. Både ulve og elge formerer sig, hvilket medfører at populationstørrelserne svinger. Typiske mønstre er, at hvis elgbestanden bliver stor, så vokser ulvebestanden efterfølgende, da der nu kan brødfødes flere ulve. Når ulvebestanden er stor, så falder elgbestanden efterfølgende, da ulvene nedlægger mange elge. Når der er få elge, så falder ulvebestanden pga. manglende føde, hvorefter elgbestanden igen vokser.

10g.0 I det følgende skal der simuleres et lukket miljø med ulve og elge. Simuleringen skal benytte følgende regler:

(a) Et miljø består af $n \times n$ felter.

(b) Alle levende dyr placeres i et felt, og der kan højst være et dyr per felt. Når et dyr dør, fjernes det fra miljøet. Hvis et dyr fødes, tilføjes det i et tomt felt. Ved simuleringens begyndelse skal der være u ulve og e elge som placeret tilfældigt i tomme felter.

(c) Miljøet opdateres i tidsenheder, som kaldes tiks, og simuleringen udføres T tiks. Indenfor et tik kan dyrene gøre et af følgende: Flytte sig, formere sig, og for ulvenes vedkommende angribe en elg. Kun et dyr handler ad gangen og rækkefølgen er tilfældig.

(d) Dyr kan flytte sig et felt per tik til de af de 8 nabofelter, som er tomme.

(e) Alle dyr har en formeringstid f angivet i antal tiks, og som tæller ned. Når formeringstiden når nul (for et levende dyr), og der er et tomt nabofelt, så fødes der et nyt dyr af samme type ved at det nye dyr tilføres i et tomt nabofelt. Moderdyrets formeringstid sættes til startværdien, hhv. f_{elg} og f_{ulv}.

(f) Ulve har en sulttid s angivet i antal tiks, og som tæller ned. Hvis sulttiden når nul, så dør ulven og fjernes fra miljøet.

(g) Ulve kan angribe og spise elge, hvis der er en elg i et nabofelt. Når en ulv angriber en elg i et nabofelt, så er der chance p for, at elgen dør og ulven spiser. Hvis elgen dør, så fjernes elgen fra miljøet, ulven flytter til elgens felt, og ulvens sulttid sættes til startværdien, s_{ulv}.

(h) Når alle handlinger er afsluttet reduceres alle formerings- og sulttællere for levende dyr med 1.

Lav et program, som kan simulere dyrene som beskrevet ovenfor og skrive en rapport. Kravene til programmeringsdelen er:

(a) Programmet skal implementere klasser for miljø, ulve og elge. Det er ikke et krav at der bruges nedarvning.

(b) Man skal kunne starte simuleringen ved angive parametrene T, n, u, e, f_{elg}, f_{ulv}, s_{ulv} og p, som argumenter til det oversatte program fra komandolinjen.

(c) De angivne parametre og tidserien over antallet af dyr per tik skal gemmes i en fil.

(d) Der skal laves et antal eksperimenter, hvor simuleringen køres med forskellige værdier af simuleringens parametre. For hvert eksperiment skal der laves en graf (ikke nødvendigvis i F#), der viser antallet af ulve og elge over tid.

(e) Programmet skal kommenteres ved brug af fsharp kommentarstandarden

(f) Programmet skal laves en white-box testing.

Kravene til rapporten er:

(g) Rapporten skal skrives i LaTeX.

(h) I skal bruge `rapport.tex` skabelonen

(i) Rapporten skal som minimum i hoveddelen indeholde afsnittene Introduktion, Problemanalyse og design, Programbeskrivelse, Afprøvning, Eksperiment og Diskussion og Konklusion. Som bilag skal I vedlægge afsnittene Brugervejledning og Programtekst.

(j) Eksperimentafsnittet skal vise tidsseriegraferne og kort diskutere hvert eksperiments udfald.

(k) Rapporten må maximalt være på 10 sider alt inklusivt.

Afleveringsopgaven skal afleveres som et antal fsx tekstfiler navngivet efter opgaven, som f.eks. `10g0.fsx`. Tekstfilerne skal kunne oversættes med fsharpc, og resultatet skal kunne køres med mono. Funktioner skal dokumenteres ifølge dokumentationsstandarden, og udover selve programteksten skal der vedlægges en kort rapport i pdf format kaldet `10g.pdf`, som indholder de dele af besvarelsen, som ikke naturligt vil være i en programtekst. Det hele skal samles i en zip fil og uploades på Absalon.

God fornøjelse.