Uitwerking Derde deeltentamen Imperatief programmeren
Vrijdag 8 november 2013, 8.30–10.30 uur

---

1. Voor elk punt (x, y) van het platte vlak, waarbij x en y reële getallen zijn, kan een bijbehorend getal worden bepaald – laten we dit het 'mandelgetal' noemen. Om het mandelgetal te kunnen uitrekenen, bekijken we eerst de volgende functie, die punten (a, b) van het vlak transformeert naar andere punten:

   $$f\ (a, b) = (a * a - b * b + x,\ \ 2 * a * b + y)$$

   Let op: deze functie transformeert het punt (a, b), maar in de berekening speelt ook de waarde van x en y, dat is het punt waarvan we het mandelgetal willen bepalen, een rol.

   Deze functie f nu, passen we toe op het punt $(a, b) = (0, 0)$. Op het punt dat daar uitkomt, passen we nog eens de functie f toe. Op het punt dat daar weer het resultaat van is, passen we opnieuw f toe, enzovoorts. We stoppen pas met toepassen van f als het resultaat-punt een afstand van meer dan 2 tot het punt $(0, 0)$ heeft. Het mandelgetal is nu gelijk aan het *aantal keren* dat f is toegepast.

   Voor sommige punten (x, y) is dat meteen al zo, en is het mandelgetal dus gelijk aan 1. Voor andere punten duurt het langer: die hebben een groter mandelgetal. Er zijn ook punten waarbij je f kan blijven toepassen, zonder dat de afstand tot de oorsprong ooit meer dan 2 wordt. Voor die punten stellen we het mandelgetal op 100.

   (a) Schrijf een methode `mandel` die het mandelgetal uitrekent van het punt waarvan de coordinaten als parameter worden meegegeven.

   (b) Schrijf het ontbrekende stuk van de methode `teken`, die de punten op het scherm zwart kleurt die een *oneven* mandelgetal hebben. De gedeclareerde `schaal` moet worden gebruikt zo dat het plaatje wordt getoond voor x en y tussen 0 en 4.

```
class Mandelbrot : Form
{
    double schaal = 0.01;

    // TODO opgave a: methode mandel

    public void teken(object obj, PaintEventArgs pea)
    {   Graphics gr = pea.Graphics;
        for (int x=0; x<400; x++)
        {   for (int y=0; y<400; y++)
            {    // TODO opgave b: body
            }
        }
    }
    public Mandelbrot
    {   this.Paint += this.teken;
    }
}
```

   Antwoord: (Zie de uitwerking van de eerste practicumopgave).

2. (a) Wat houdt het in als een programma een *MDI-interface* heeft?
   Wat gebeurt er met de menu-balk in dat soort programma's?
   Antwoord:
   Een programma met een *Multiple Document Interface* (MDI) toont aan de gebruiker een zogeheten *parent*-window, waarbinnen nul of meer *child*-windows ronddrijven, die elk een document tonen.
   De menubalk van het actieve child-window wordt gecombineerd met die van het parent-window getoond in het parent-window.

(b) Bij het schrijven van een tekstfile kun je kiezen uit een aantal verschillende *encodings*. Mogelijk zijn onder andere Ascii, Latin1 (ook bekend als iso-8859-1), Unicode, en UTF8.

- Noem een voordeel en een nadeel van Latin1 in vergelijking met Unicode.
- Noem een voordeel en een nadeel van UTF8 in vergelijking met Unicode.

Antwoord:

Latin1 gebruikt slechts 1 byte per character tegen Unicode 2 bytes, maar Latin1 kan dan ook slechts 256 verschillende tekens weergeven, tegen Unicode 65536.

UTF8 gebruikt voor de eerste 128 characters slechts 1 byte, en kan westerse teksten dus compacter opslaan. Nadeel is dat de (de)codering ingewikkelder is, en dat chinese teksten juist 3 bytes per karakter kosten tegen 2 in Unicode. (UTF8 kan precies dezelfde 65536 characters representeren als Unicode, dus dat is geen voordeel of nadeel).

(c) Wat is het verschil tussen een *abstracte methode* en een *virtuele methode*?

Wat is er het voordeel van om een methode *abstract* te maken in plaats van *virtual*?

Antwoord:

Bij de definitie van een abstracte methode wordt de body weggelaten. Een abstracte methode *moet* in de subklassen worden overridden, een virtuele methode *mag* in de subklassen worden overridden.

(NB: Zowel een abstracte als een virtuele methode kan in een subklasse worden overridden, dus dat is geen verschil. Zowel abstracte als virtuele methoden spelen een rol in de klasse-hiërarchie, dus dat is geen verschil. Een abstract methode kan wel worden aangeroepen –ook al heeft hij geen body– (dat is nou juist de lol ervan), net als een virtuele methode, dus dat is geen verschil. Een abstract methode heeft wel een object onderhanden (al zal dat object noodzakelijkerwijs een subklasse als type hebben); methoden zonder object onderhanden heten *statisch*, maar daar ging het hier niet over.)

Als een methode abstract is (in plaats van virtueel met een lege body), dan kun je in het programma niet per ongeluk objecten aanmaken van de (nu abstracte) superklasse, en niet per ongeluk vergeten om de methode te overriden in een subklasse.

(d) Hoe ziet de body van een `interface`-declaratie eruit?

Stel dat er een `interface A` is gedeclareerd. Noem twee plaatsen in de syntax van een programma waar een interface-naam, zoals `A`, gebruikt kan worden.

Antwoord: In de body van een interface-declaratie staan methode-headers (en property-headers) van de methoden die in elke implementatie van deze interface gedefinieerd moeten worden. (In een interface-declaratie staan *geen* variabele-declaraties!).

De interface-naam kan worden gebruikt achter de dubbele punt in een klasse-header van een klasse die deze interface implementeert (`class B:A`), als type bij de declaratie van een variabele (die dan naar een object elke implementatie van de interface kan wijzen) (`A x=new B();`), en op allerlei andere plaatsen waar een type kan staan, zoals het elementtype van een collection (`List<A>`), het resultaat van een methode (`A m()`) enzovoorts. (Maar *niet* achter `new`, zoals in `A x=new A();`).

(e) Wat is het essentiële verschil tussen een `FileStream` en een `StreamReader` ?

De klasse `FileStream` is een voorbeeld van een *store*. Daarnaast kennen we ook zogeheten *decorators*. Wat is het verschil tussen een store en een decorator?

Antwoord:

Met een `StreamReader` kun je characters en strings, oftewel *teksten* lezen. Met een `FileStream` kun je bytes, oftewel *datafiles* lezen.

Niet het bedoelde antwoord, maar wel goed: met een `StreamReader` kun je naar keuze lezen of schrijven. Met een `FileStream` kun je alleen maar lezen.

(Dat een `StreamReader` een subklasse is van `TextReader`, en een `FileStream` een subklasse van `Stream` is wel waar, maar niet precies genoeg om een essentieel verschil te zijn, omdat het meteen de vraag oproept wat dan het verschil tussen een `TextReader` en een `Stream` is.)

Een *store* faciliteert de daadwerkelijke opslag op een extern medium (file, netwerk, geheugen), Een *decorator* voert tijdens het lezen/schrijven een vertaalslag uit (comprimeren, cryptograferen) maar vertrouwt voor de eigenlijke opslag op een onderliggende store.

(Het antwoord 'een decorator voegt iets toe aan de store' is te vaag, ook al omdat het ambigu is: iets toevoegen aan een file betekent meestal immers dat je informatie aan het eind van de file erbij schrijft.)

(f) Beschrijf de syntax en de semantiek van een `foreach`-opdracht.

In welke situatie is het noodzakelijk om een `foreach`-opdracht te gebruiken en is het niet mogelijk om een `for`-opdracht met een tellertje te gebruiken?

In welke situatie is het noodzakelijk om een `for`-opdracht met een tellertje te gebruiken en is het niet mogelijk om een `foreach`-opdracht te gebruiken?

Antwoord: Syntax van een `foreach`-opdracht:

`foreach (` *type naam* `in` *expressie* `)` *opdracht*

(Eventueel nog toevoegen: hierin moet de *expressie* een Collection (of eigenlijk een `IEnumerable`) als waarde hebben waarvan de elementen het genoemde *type* hebben.)

(Veel begane slordigheid: hinken op twee gedachten tussen een abstracte beschrijving zoals hierboven, en een voorbeeld, dus bijvoorbeeld `x` in plaats van *naam*, of `{ /* doe iets */ }` in plaats van *opdracht*.)

Semantiek: de *opdracht* wordt uitgevoerd voor elk element van de *expressie*, waarbij de *naam* steeds de waarde van dat element aanneemt.

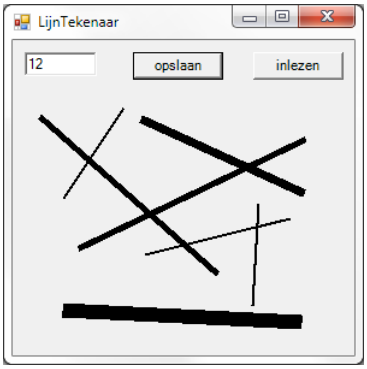
Foreach noodzakelijk: als de Collection niet ook een List is, en dus geen index-operatie kent.

(Niet helemaal goed: als de Collection niet *gesorteerd* is: sorteren is iets anders dan indexeren. Strikt genomen kan overigens elke `foreach`-opdracht toch gesimuleerd worden met een `for`-opdracht, die dan geen tellertje gebruikt maar een `IEnumerator`.)

For noodzakelijk: als er helemaal geen gegevensverzameling is, maar gewoon een opdracht meermalen wilt uitvoeren.

(Ook goedgerekend: 'als je een array wilt doorlopen'. Dat gebeurt inderdaad vaak met een for-opdracht, maar in C# is elke array ook een implementatie van `IEnumerable`, dus kun je ook daarvoor een foreach-opdracht gebruiken.)

3. Bekijk het programma `LijnTekenaar`, waarvan hiernaast een screenshots staat. In het window zijn een tekstveld en twee buttons zichtbaar. De gebruiker kan met de muis steeds twee punten aanklikken, die dan verbonden worden door een lijn. Het aantal lijnen dat de gebruiker mag tekenen is niet aan een maximum gebonden.
De dikte van de lijn wordt gespecificeerd door het getal dat in het tekstveld staat ingevuld op het moment van de tweede klik. Je mag zonder controle aannemen dat in het tekstveld alleen maar cijfertekens zijn ingevuld.



Hieronder is al een deel van het programma gegeven. Vul hierop het volgende aan:

(a) Schrijf een hulpklasse `Lijn`, zo dat in een object van die klasse alle gegevens die nodig zijn om een lijn te kunnen tekenen beschikbaar zijn. Maak een constructormethode die zo'n object van beginwaardes voorziet, een methode `ToString` die de waarden 'inpakt' in een string, en een tweede constructormethode die zo'n string weer 'uitpakt'.

(b) Schrijf de methoden `muisklik` en `teken` in de klasse `LijnTekenaar`, en geef de declaraties van de daarvoor benodigde member-variabelen.

(c) Als de gebruiker op de knop 'opslaan' drukt, wordt de toestand van de tekening opgeslagen in een file waarvan de naam in de constante `naam` staat. Als de gebruiker op de

knop ‘inlezen’ drukt, wordt de huidige tekening vervangen door de opgeslagen tekening. Schrijf de methoden `opslaan` en `inlezen` die daarvoor nodig zijn.

```
using System;
using System.Windows.Forms;
using System.Drawing;
using System.Collections.Generic;
using System.IO;

namespace Opgave3
{   public class Program
    {   static void Main()
        {   Application.Run(new LijnTekenaar());
        }
    }
    public class Lijn
    {   // OPGAVE a
    }
    public class LijnTekenaar : Form
    {   const string naam = "tekening.txt";
        TextBox tb;

        public LijnTekenaar()
        {   this.Text = "LijnTekenaar";
            Button b1, b2;
            tb = new TextBox(); tb.Text="5"; tb.Location=new Point(10,10); tb.Size=new Size(60,30)
            b1 = new Button();  b1.Text="opslaan"; b1.Location=new Point(100,10);
            b2 = new Button();  b2.Text="inlezen"; b2.Location=new Point(200,10);
            this.Controls.Add(tb); this.Controls.Add(b1); this.Controls.Add(b2);
            this.MouseClick += this.muisklik;
            this.Paint      += this.teken;
            b1.Click        += this.opslaan;
            b2.Click        += this.inlezen;
        }
        // OPGAVE b en c
    }
}
```

Antwoord:

(a) De klasse `Lijn`:

```
public class Lijn
{
    public Point start, eind;
    public int dikte;

    public Lijn(Point p1, Point p2, int d)
    {   start = p1;
        eind  = p2;
        dikte = d;
    }
    public Lijn(string s)
    {   string[] v = s.Split();
        start = new Point( int.Parse(v[0]), int.Parse(v[1]));
        eind  = new Point( int.Parse(v[2]), int.Parse(v[3]));
        dikte = int.Parse(v[4]);
    }
    public override string ToString()
    {   return start.X + " " + start.Y + " " + eind.X + " " + eind.Y + " " + dikte;
    }
}
```

(b) De methoden `klik` en `teken` met membervariabelen:

```
List<Lijn> lijnen = new List<Lijn>();
Point punt;
int n = 0;

void klik(object o, MouseEventArgs mea)
{   if (n%2==0)
        punt = mea.Location;
    else
        lijnen.Add(new Lijn(punt, mea.Location, int.Parse(tb.Text)));
    n++;
    this.Invalidate();
}
void teken(object o, PaintEventArgs pea)
{   foreach (Lijn lijn in lijnen)
        pea.Graphics.DrawLine( new Pen(new SolidBrush(Color.Black), lijn.dikte)
                              , lijn.start, lijn.eind
                              );
}
```

(c) De methoden `opslaan` en `inlezen`:

```
void opslaan(object o, EventArgs ea)
{   StreamWriter w = new StreamWriter(naam);
    foreach (Lijn lijn in lijnen)
        w.WriteLine(lijn.ToString());
    w.Close();
}
void inlezen(object o, EventArgs ea)
{   lijnen.Clear();
    StreamReader r = new StreamReader(naam);
    string regel;
    while ((regel = r.ReadLine()) != null)
        lijnen.Add(new Lijn(regel));
    r.Close();
    this.Invalidate();
}
```

EINDE TENTAMEN