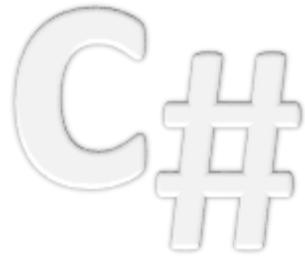


# Inleiding en doelstellingen: .NET Essentials: C#

Hoofdstuk 0



# Overzicht

- Lessen
- Blackboard
- Evaluatie

# Lessen: wat?

- Theoretische uitleg
- Denkoeferingen
- Pen-en-papier
- Demonstraties
- In praktijk brengen van theorie
  - programmeren met C#
  - leren werken met IDE
- H1-23 in het handboek (alles!)



*Niet kennen: ... blz ... (als bepaalde topics niet gekend moeten zijn, worden die weergeven in de slides m.b.v. deze kader)*

# Lessen: hoe?

- Tempo = hoog
- Breng mee naar de les!
  - laptop
  - handboek
- Neem notities!
- Lees na elke les de behandelde leerstof in je handboek na! Zorg dat je elke zin, elk woord (terminologie is ook belangrijk!), elk stuk code, ... volledig begrijpt. Vragen noteer/markeer je. Deze vragen worden aan het begin van de eerstvolgende les door je junior-collega's of lector beantwoord.

# Lessen: hoe?

- Sommige hoofdstukken zijn zelfstudie → tijdig voorbereiden!
- Werk actief mee!
- Tracht zoveel mogelijk tijdens de sessies te doen
- Stel liever je vragen tijdens de les (i.p.v. per mail); je collega-studenten hebben vaak dezelfde vragen!

# Blackboard

- Iedereen is reeds geregistreerd
- Inhoud:
  - Lessenverloop
  - Slides
  - Mededelingen
  - Opgave van taken
  - Links Pluralsight
  - Voorbeeldoplossingen
  - ...

# Evaluatie (1<sup>e</sup> en 2<sup>e</sup> zit)

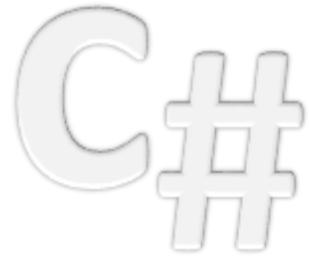
- Examen (100 %)
  - Theoretisch deel (15 %) = gesloten boek zonder laptop!
  - Praktisch examen (85%) = open boek met laptop
    - Een of meerdere programmeeroefeningen, in dezelfde stijl als de moeilijkere oefeningen gemaakt tijdens het jaar
    - je mag meebrengen naar het examen: handboek en afdruk van de slides

# De achtergronden van C#

Hoofdstuk 1

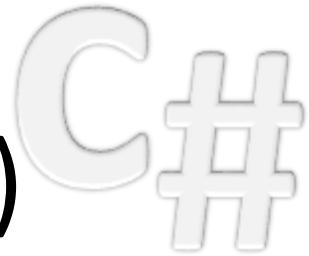
# In dit hoofdstuk ...

- Hoe en waarom C#?
- Wat is nieuw?
- Wat is .NET?
- Beginselen van programmeren



# De geschiedenis van C#

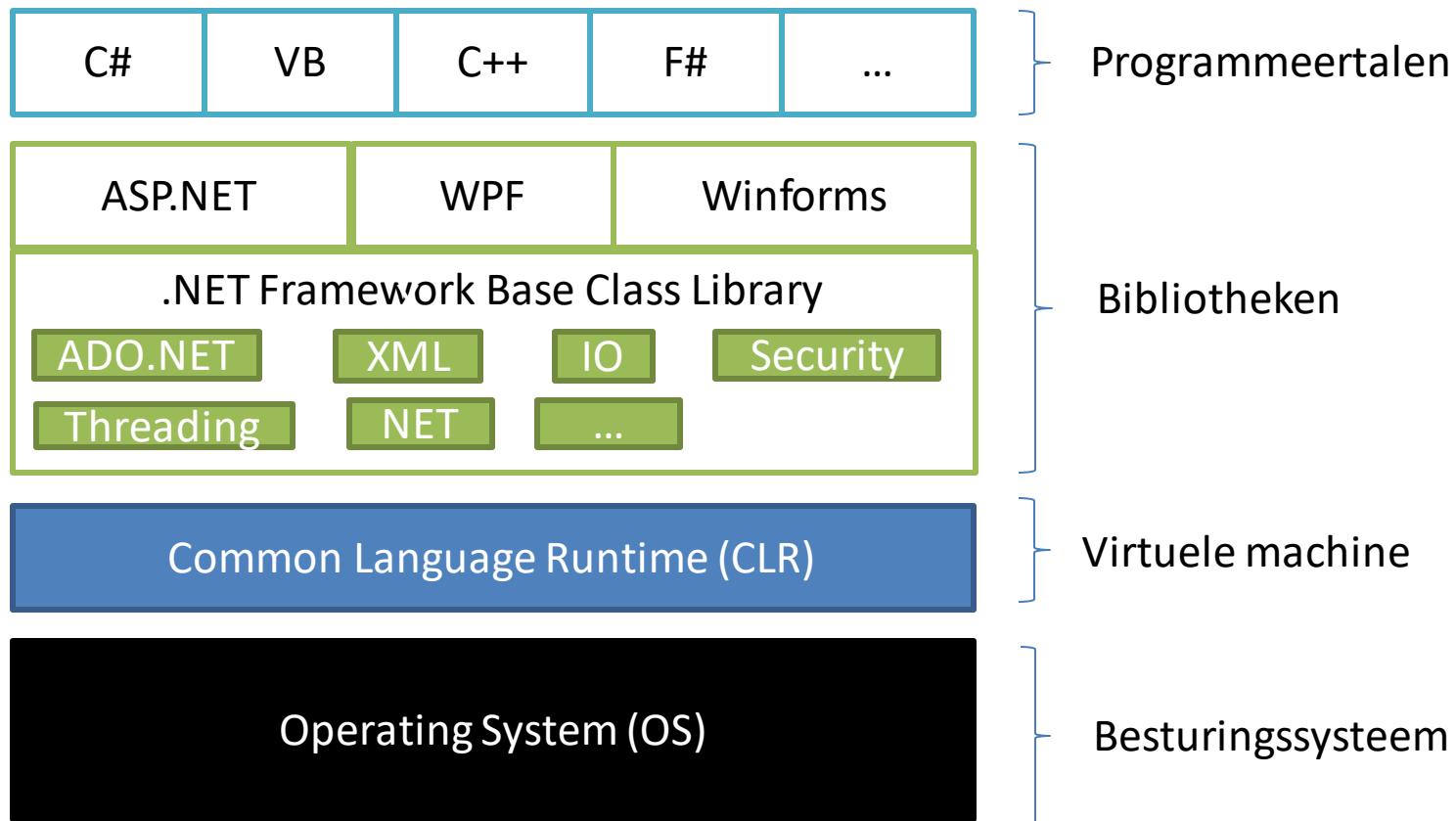
- Uitspraak: See Sharp
- 1960: Algol (Algorithmic Language)
- 1970: C
- 1980: C++
- 1990: Java
- 2002: C#



# .NET omgeving (framework)

- Runtime omgeving geïnstalleerd op (Windows) OS.
- Programmeertalen (C#, C++, VB.NET, F#)
- Groot aantal componenten (klassen) voor het schrijven van programma's
  - Web applicaties
  - Desktop applicaties
  - ...
- Geïnspireerd door Java:
  - VM (~CLR), OOP, Garbage collection, ...
- Niet meer enkel Windows gebonden
  - Met Xamarin compileren naar iOS, Android, MacOS X
  - Laatste versie (.NET Core) is platformonafhankelijk

# Overzicht .NET omgeving



# Wat is een programma?

- Logisch stappenplan (algoritme)
- Bestaat uit:
  - Opeenvolgingen (statements)
  - Herhalingen (lussen)
  - Selecties (If)
  - Methode-oproepen op objecten
    - Kant-en-klaar
    - Zelf geschreven

Maak je haar nat  
Doe er shampoo op  
Wrijf de shampoo door je haar  
Spoel het uit

# Samenvatting

- C# is afgeleid uit Java en C++
- Maakt deel uit van .NET (zeer krachtig)
- Een programma is een lijst van instructies
- OOP is zeer belangrijk

## Oefening 1.2

10000 blaadjes papier met een getal erop. Hoe ga je het grootste getal vinden?

*Dit soort redeneringen en voorstellingen helpen vaak bij het opstellen van programma's.*

# De C#-ontwikkelloomgeving

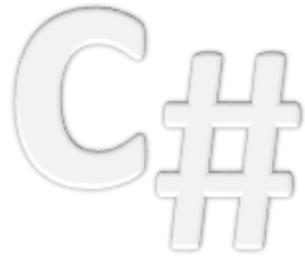
Hoofdstuk 2

# In dit hoofdstuk ...

- Kennismaking met de IDE
- Projecten
- Besturingselementen
- Een programma runnen
- Compilatiefouten opsporen

# Integrated Development Environment

- Een front-end voor de verschillende tools
  - Code editor
  - Compiler, linker
  - Debugger
  - Help
  - UI tekenen
  - Vaak voor verschillende talen en besturingssystemen



# Visual Studio

About Microsoft Visual Studio

Visual Studio

Microsoft Visual Studio Community 2017  
Version 15.9.4

© 2017 Microsoft Corporation.  
All rights reserved.

Installed products:

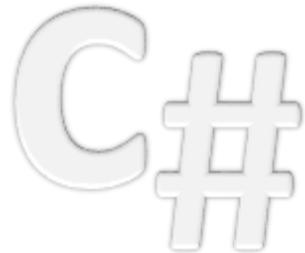
- C# Tools – 2.10.0-beta2-63501-03+b9fb1610c87cccc8ceb74a770dba261a58e39c4a
- Common Azure Tools – 1.10
- NuGet Package Manager – 4.6.0
- ProjectServicesPackage Extension – 1.0
- ResourcePackage Extension – 1.0
- ResourcePackage Extension – 1.0
- Visual Basic Tools – 2.10.0-beta2-63501-03+b9fb1610c87cccc8ceb74a770dba261a58e39c4a
- Visual Studio Code Debug Adapter Host Package – 1.0

Product details:

C# components used in the IDE. Depending on your project type and settings, a different version of the compiler may be used.

Warning: This computer program is protected by copyright law and international treaties. Unauthorized reproduction or distribution of this program, or any portion of it, may result in severe civil and criminal penalties, and will be prosecuted to the maximum extent possible under the law.

OK



# Visual Studio Community

The screenshot shows the Microsoft Visual Studio Community Start Page. At the top left is the 'Start Page - Microsoft Visual Studio' title bar with a 'Quick Launch (Ctrl+Q)' search bar. The main area has a 'Get Started' section with tips like 'Build your first app in 5 minutes!', 'Maximize your productivity with these tips and tricks for Visual Studio', and 'Develop modern, fully-native, Android and iOS apps'. Below this is a 'Recent' section under 'This week' and 'Last week', listing recent projects such as 'Galgje.sln', 'Some Shapes.sln', 'Oef3\_5\_Roos.sln', '3.6\_gezicht.sln', 'WpfApp5.sln', and 'WpfApp4.sln'. To the right are sections for 'Open' (with options for Visual Studio Team Services, Open Project / Solution, and Open Folder) and 'New project' (with a search bar and recent templates like 'WPF App (.NET Framework)'). On the far right is a 'Developer News' sidebar with articles like 'Database DevOps with Redgate ReadyRoll [video]', 'Delivering Rich Media Experiences with the Xamarin.Forms Video Player', 'Xamarin Developer Events This January', 'CoreML Programming with Xamarin.Mac and F#', and '.NET Framework January 2018 Security and Quality Rollup'. The bottom navigation bar includes 'Solution Explorer' and 'Team Explorer' tabs.

# Een eerste programma schrijven

- Demo
- Kies voor WPF Application
- Kies een projectnaam (“Name”)
- Het project wordt opgeslagen in de map die je kiest bij “Location”
- Uitvinken: “Create directory for solution”

# Besturingselementen

- (eng: Control)
- Informatie presenteren en/of interactie met de gebruiker
- In “ontwerpfas” of “at design-time” kan je de lay-out en de properties instellen
- In de “uitvoeringsfas” of “at run-time” worden de controls zichtbaar in een Windows applicatie en kunnen de properties eventueel nog veranderen
- Property = eigenschap, zoals kleur, font, positie, ...  
in Java → “getters” en “setters”
- Name-property → de variabele naam van de control
- XAML <-> ontwerpvenster

# Events en de Button control

- Event = gebeurtenis
- Door de gebruiker, bv. Klik op knop
- Door het systeem, bv. Nieuw bericht
- Elke control kan reageren op events,  
dit noemt men een event-handler  
→ speciale methode die uitgevoerd wordt

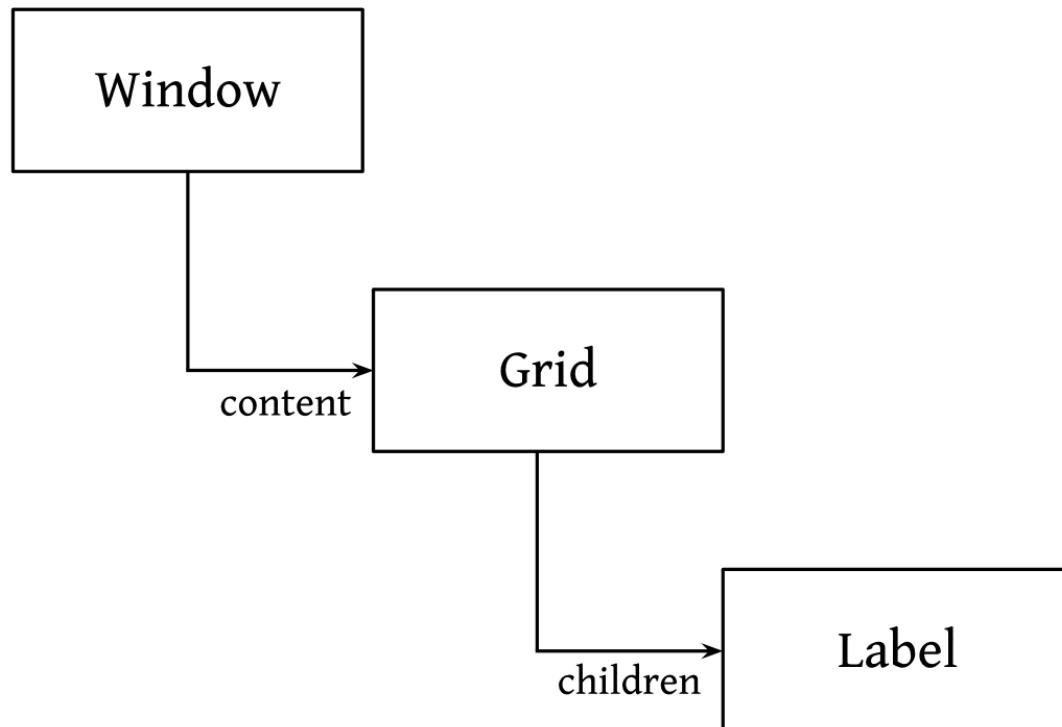


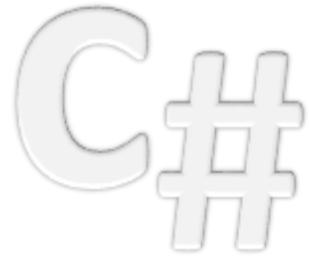
# Demo: Hello Button

## XAML-code

```
<Window x:Class="Hello_Button.MainWindow"
        xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        Title="MainWindow" Height="350" Width="525">
    <Grid>
        <Button Content="Click me" HorizontalAlignment="Left" Margin="60,60,0,0"
               VerticalAlignment="Top" Width="75"
               Name="testButton" Click="testButton_Click"/>
        <Label Content="" HorizontalAlignment="Left"
              Margin="60,100,0,0" VerticalAlignment="Top"
              Name="resultLabel"/>
    </Grid>
</Window>
```

# XAML hiërarchie





# Demo: Hello Button

```
public partial class MainWindow : Window
{
    public MainWindow()
    {
        InitializeComponent();
    }
    private void testButton_Click(object sender, RoutedEventArgs e)
    {
        resultLabel.Content = "Hello world";
        //MessageBox.Show("Hello world!");
    }
}
```

# Editorfaciliteiten - Programmeerfouten

- Autocompletion → ga nooit verder als de huidige lijn niet compileert
- Layout (code convention)
- Hoofdletter (code conventions)
- Selecteren objecten, properties en methodes

# Grammatica

## C#

- ; na elk statement
- Blokken: { ... }
- Opsplitsen van lange lijnen:
  - Na haakje
  - Na komma
  - Na operator

# Grammatica

## XAML

- Opentag (begintag) en sluittag (eindtag)  
vb: <Grid> ... </Grid>
- Lege elementen vb: <Label />
- Attributen: name = “value”
- Opsplitsen van lange lijnen:
  - je kan attribuut op een volgende regel beginnen

# MessageBox

- `MessageBox.Show("Hello");`
- Belangrijke boodschappen  
(niet overdrijven)
- Eventueel als debug hulpmiddel (als je nog niet weet hoe  
je de Visual Studio debugger kan gebruiken. Zie hoofdstuk 9)

# Geïntegreerde Help

- Zeer nuttig voor opzoekingswerk
  - Methodes
  - Properties
  - Objecten
- *niet* om de taal zelf aan te leren
  - Syntax
  - OO principes

# Compatibiliteit Visual Studio

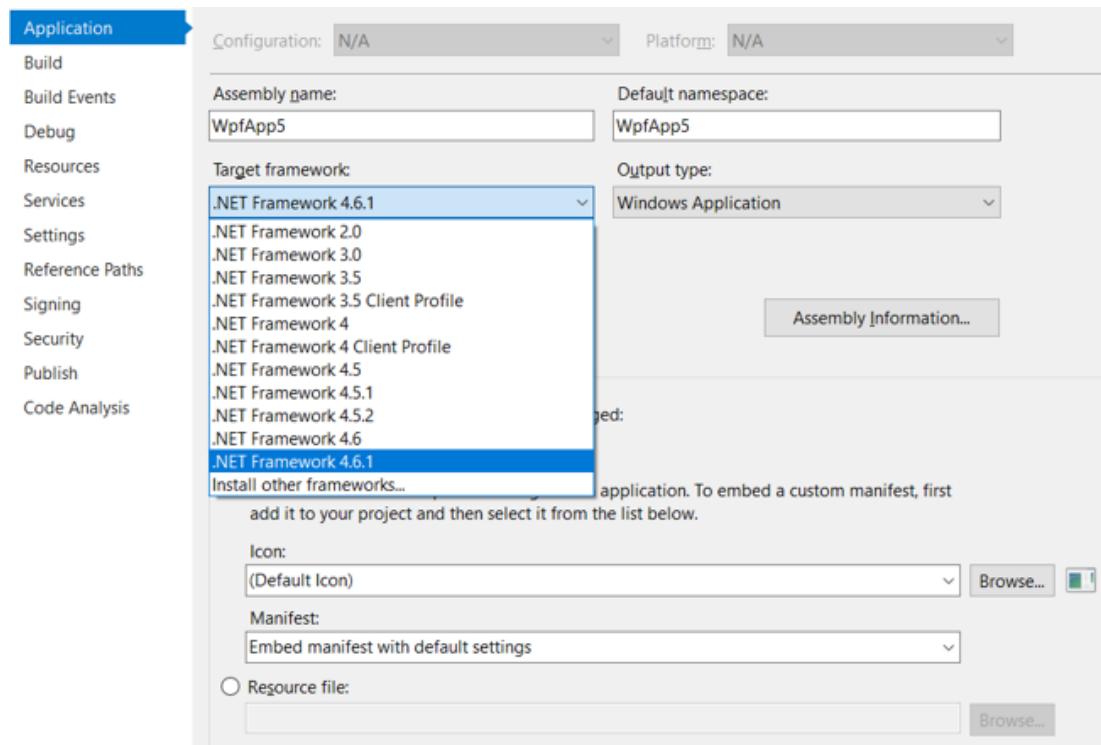
- VS 2017
  - Via [OnTheHub](#) te downloaden (gratis account)
  - Gratis Community editie:  
<http://www.visualstudio.com>

# Compatibiliteit .NET runtime

- De “meeste” applicaties gemaakt met oudere edities van .NET zullen draaien in .NET 4.7 (zonder hercompileren, dus rechtstreeks via het .exe bestand)
- De “meeste” applicaties gemaakt met een hogere versie van .NET zullen “waarschijnlijk” niet draaien in een lagere versie van de runtime
- Testen is de boodschap!
- Nog beter: versieproblemen vermijden door de correcte versie van .NET te selecteren en te hercompileren!

# Target framework selecteren

- Project properties
- Voor ons: “.NET Framework 4.6.1”



# Inleiding tot het gebruik van graphics

Hoofdstuk 3

# In dit hoofdstuk ...

- Tekenfaciliteiten voor eenvoudige vormen
- Methode aanroep
- Argumenten doorgeven
- Instructies schrijven die een programma vormen
- Commentaar toevoegen

# OO voorstelling systeem

Typisch aan OO programma's is dat zij de realiteit rondom ons trachten te modelleren met "objecten".



# OO voorstelling systeem

Een “Object” is een instantie van een bepaalde  
“Klasse” ...

```
Rectangle aRectangle = new Rectangle();
```

... methodes kunnen worden opgeroepen;

```
paperCanvas.Children.Add(aRectangle);
```

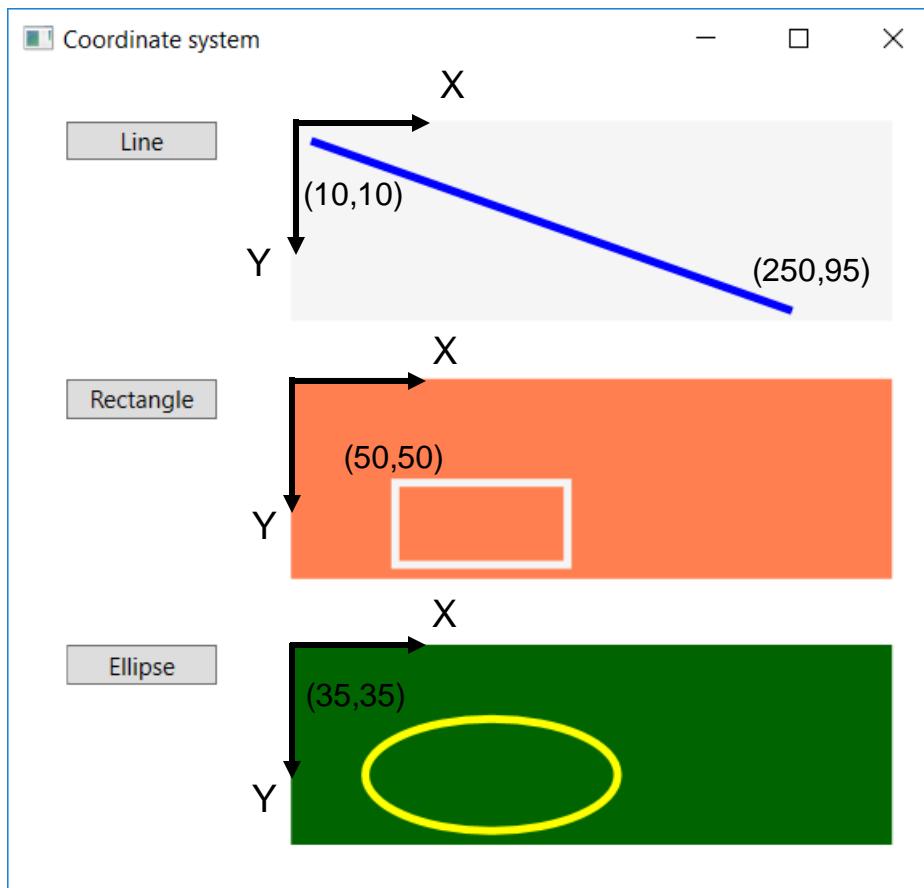
... en properties kunnen worden gebruikt.

```
aRectangle.Width = 100;
```

# Demo: een eerste tekening

- Let op de “IntelliSense”
- Geen syntaxfouten laten staan
  - Pijl ↑ of ↓: varianten van methodes
  - F1 toets: help venster

# Relatieve coördinaten



# Tekenmethoden

- Teken op een stuk papier enkele vormen, samen met de correcte C# methode oproep
- Wat is het verschil tussen de klasse Brush en de klasse SolidColorBrush?
- Van welke klasse zijn de kleuren Black, Indigo, enz?

# Tekenmethoden

- Lees de listing van het programma  
“Some Shapes”
- Statements worden sequentieel uitgevoerd
- Voorzie je problemen met dit programma als  
je dit thuis wil uitvoeren?
- Wat kan een oplossing zijn voor dit probleem?

# Commentaar

- Commentaar op één lijn

```
// draw a triangle
// base line
Line triangleBase = new Line();
triangleBase.X1 = 20; triangleBase.Y1 = 80;
triangleBase.X2 = 70; triangleBase.Y2 = 70;
triangleBase.Stroke = new SolidColorBrush(Colors.Black);
// left leg
Line leftLine = new Line();
...
```

# Commentaar

- Commentaar op één lijn

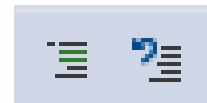
```
/* draw a triangle */  
/* base line */  
Line triangleBase = new Line();  
triangleBase.X1 = 20; triangleBase.Y1 = 80;  
triangleBase.X2 = 70; triangleBase.Y2 = 70;  
triangleBase.Stroke = new SolidColorBrush(Colors.Black);  
/* left leg */  
Line leftLeg = new Line();  
...
```

# Commentaar

- Commentaar over meerdere lijnen

```
/* example 1:  
 draw a triangle  
 base line*/  
Line triangleBase = new Line();  
...
```

- In het Engels! Wees kort, duidelijk, volledig
- Schrijf geen overbodige commentaar
- Dikwijls kan je commentaar vervangen door een methode(proef), dit verhoogt de leesbaarheid (en herbruikbaarheid) van je code
- VS tip: CTRL+K, CTRL+C en CTRL+K, CTRL+U of



# Methode i.p.v. commentaar

```
...
this.DrawTriangle(...);

private void DrawTriangle(Canvas drawingArea,
                        SolidColorBrush brushToUse,
                        int x1, int x2, int x3,
                        int y1, int y2, int y3)
{
    DrawLine(drawingArea, brushToUse, x1, y1, x2, y2);
    DrawLine(drawingArea, brushToUse, x2, y2, x3, y3);
    DrawLine(drawingArea, brushToUse, x3, y3, x1, y1);
}
```

# Methode i.p.v. commentaar

```
private void DrawLine(Canvas drawingArea,  
                      SolidColorBrush brushToUse,  
                      int startX, int startY,  
                      int endX, int endY)  
{  
    Line line = new Line();  
    line.X1 = startX; line.X2 = endX;  
    line.Y1 = startY; line.Y2 = endY;  
    line.Stroke = brushToUse;  
    drawingArea.Children.Add(line);  
}
```

# Methode oproepen

- Correcte oproep
  - Aantal argumenten
  - Juiste type
  - Goede volgorde
- Geen argumenten: toch ()
- Soms zijn methoden “overloaded”
  - Zelfde naam
  - Andere parameters

# Objecten tekenen met XAML

- Rectangle

```
<Canvas ...>
    <Rectangle Width="100" Height="50"
               Margin="10,10,0,0" Stroke="Black"
               Name="upperRectangle" Visibility="Hidden"/>
    ...
</Canvas>
```

- Line

```
<Line X1="10" Y1="10" X2="110" Y2="60" Stroke="Black"
      Name="lineInRectangle" Visibility="Hidden"/>
```

# Objecten tekenen met XAML

- Ellipse

```
<Ellipse Width="100" Height="50" Margin="10,80,0,0"  
        Stroke="Black" Name="ellipseInRectangle"  
        Visibility="Hidden"/>
```

- Image

```
<Image Source="imagedemo.jpg" Margin="120,10,0,0"  
      Width="150" Height="150"  
      Name= "rightImage" Visibility="Hidden"/>
```

# Objecten tekenen met XAML

```
private void drawButton_Click(...)  
{  
    upperRectangle.Visibility = Visibility.Visible;  
    lineInRectangle.Visibility = Visibility.Visible;  
    ellipseInRectangle.Visibility = Visibility.Visible;  
    rightImage.Visibility = Visibility.Visible;  
}
```

# Discussie: code conventies in C#

- *Welke code conventies ontdek je in de code van het handboek?*
- *Zijn er overeenkomsten / verschillen met de code conventies van Java?*

# Variabelen en berekeningen

Hoofdstuk 4

# In dit hoofdstuk ...

- Typen van numerieke variabelen
- Variabelen declareren
- Toekenning
- Rekenkundige operatoren
- Getallen in labels/texblocks en tekstvakken
- Eigenschappen van strings

# Wat is een variabele?

- *Leg uit in je eigen woorden ...*
- 2 belangrijke types:
  - int → gehele getallen
    - 4 bytes
  - double → kommagetallen
    - 8 bytes
- Andere types voor getallen:
  - long: zeer lange gehele getallen
  - float: kommagetallen, kleiner bereik dan double
  - decimal: grote precisie (financiële berekeningen)

# Declareren

- Zinvolle namen
- Regels
  - Beginnen met een letter of \_
  - Voorts letters of cijfers of \_
  - Geen spaties, symbolen of keywords
  - Max 255 tekens
  - camelCasing, bv: newStudentBachelor
- C# is hoofdlettergevoelig, maar de IDE helpt tijdens het typen

# Demo: Area of Rectangle

Type varNaam [ = startExpr];

```
double personHeight = 1.68;
int a = 3, b = 4;
int examMark = 65;
int betterMark = examMark + 10;
double salary; // unassigned
string name; // unassigned
```

# Toekenningsopdracht

```
length = 20;
```

- De variabele length krijgt de waarde 20
- length wordt 20
- Leesrichting: ←
- Puntkomma ;
- **Niet: length is gelijk aan 20!**



# Berekeningen en operatoren

```
length = length + 1;  
length += 1;  
length++;
```

- De nieuwe waarde van length wordt de oude waarde plus één
- Links: variabele naam, rechts: expressie
- Verkorte schrijfwijzen

# Operatoren

Operator	Betekenis	Prioriteit
*	Vermenigvuldigen	1
/	Delen	1
%	Modulo	1
+	Optellen	2
-	Aftrekken	2

# Operatoren

- Voorrangsregels: \*, / en % voor + en –
- Bij gelijke voorrang: van links naar rechts
- Gebruik van haakjes is aanbevolen
- Opsplitsen van expressies is aanbevolen  
(lange regels vermijden)
- Zie testvragen

# Operatoren

```
int i; int n = 3; double d;  
i = n + 3;                  // i krijgt de waarde 6  
i = n * 4;                  // i krijgt de waarde 12  
i = 7 + 2 * 4;              // i krijgt de waarde 15  
n = n * (n + 2) * 4;        // n krijgt de waarde 60  
d = 3.5 / 2;                // d krijgt de waarde 1.75  
n = 7 / 4;                  // n krijgt de waarde 1
```

# Strings

- Type: `string`
- Dit is een verkorte schrijfwijze voor de .NET klasse `System.String`
- Samenvoegen (concatenatie): `+`
- String waarden: dubbele quotes

```
string firstName = "Anders";
string lastName, wholeName;
string greeting; lastName = "Hejlsberg";
wholeName = firstName + " " + lastName;
greeting = wholeName + " invented C#";
```

# Strings en getallen

- Automatisch omzetten naar string

```
int streetNumber = 7;  
string streetName = "th Avenue";  
string fullName = streetNumber + streetName;
```

- Soms verwarrend:

```
int numberOfAppels = 2, numberOfPears = 3;  
string answer, note = "Answer is ";  
answer = note + numberOfAppels + numberOfPears;
```

Answer is 23  
Waarom?

# Typeconversie

- Als je variabelen gebruikt op plaatsen waar andere types verwacht worden, krijg je een fout. (Demo)

```
int age = 36;  
MessageBox.Show(age); // wrong!
```

- Oplossing: omzetting naar het juiste type

# Conversiefuncties

int waarde 1234

Convert.ToString

string waarde "1234"

double waarde 12.94

Convert.ToInt32

int waarde 13  
(afgerond)

double waarde 12.94

(int)

int waarde 12  
(afgekapt)

string waarde "12,94"

Convert.ToDouble

double waarde 12.94

# Getallen formatteren

- Omslachtig en foutgevoelig:

```
resultLabel.Content = euros + " euros and " + cents + " cents";
```

**composite format string** (= vaste string met 'indexed placeholders')

- Formatteren:

```
resultLabel.Content = String.Format("{0} euros and {1} cents",  
                                   euros, cents);
```

OF

```
resultLabel.Content = $"{euros} euros and {cents} cents";
```

**interpolated string** (= template string met 'interpolated expressions')

# Placeholders

Placeholder	Voorbeeld
{expr:c}	€ 32,54
{expr:d}	3254
{expr:0.000}	32,540
{expr:0.###}	32,54
{expr:t}	15:37

Online: [Standard Numeric Format Strings](#)

# Nieuwe Componenten

- Klasse: TextBox
  - Invoer van de gebruiker
  - Property: Text
- Klasse: Label
  - Tekstuele informatie
  - Niet wijzigbaar door gebruiker
  - Wel programmatorisch wijzigbaar
  - Property: Content en Property: Target
- Klasse: TextBlock
  - Tekstuele informatie
  - Niet wijzigbaar door gebruiker
  - Wel programmatorisch wijzigbaar
  - Property: Text
- Demo: Euros and Cents

# Methoden en argumenten

Hoofdstuk 5

# In dit hoofdstuk ...

- Methodes en functies schrijven
- Argumenten en parameters
- Doorgeven als waarde en als referentie
- Het gebruik van return bij functies

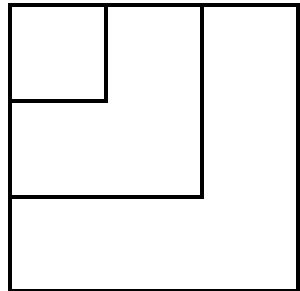
# Het nut van methodes

- Herhaalde statements groeperen
- “Op een hoger niveau denken”

*Je splitst een programma op in deeltaken die je op hun beurt weer in deeltaken opsplits, enz. De taken op het laagste niveau zijn eenvoudig te begrijpen en te programmeren*

- “Verdeel en Heers” principe

# Een bedrijfslogo

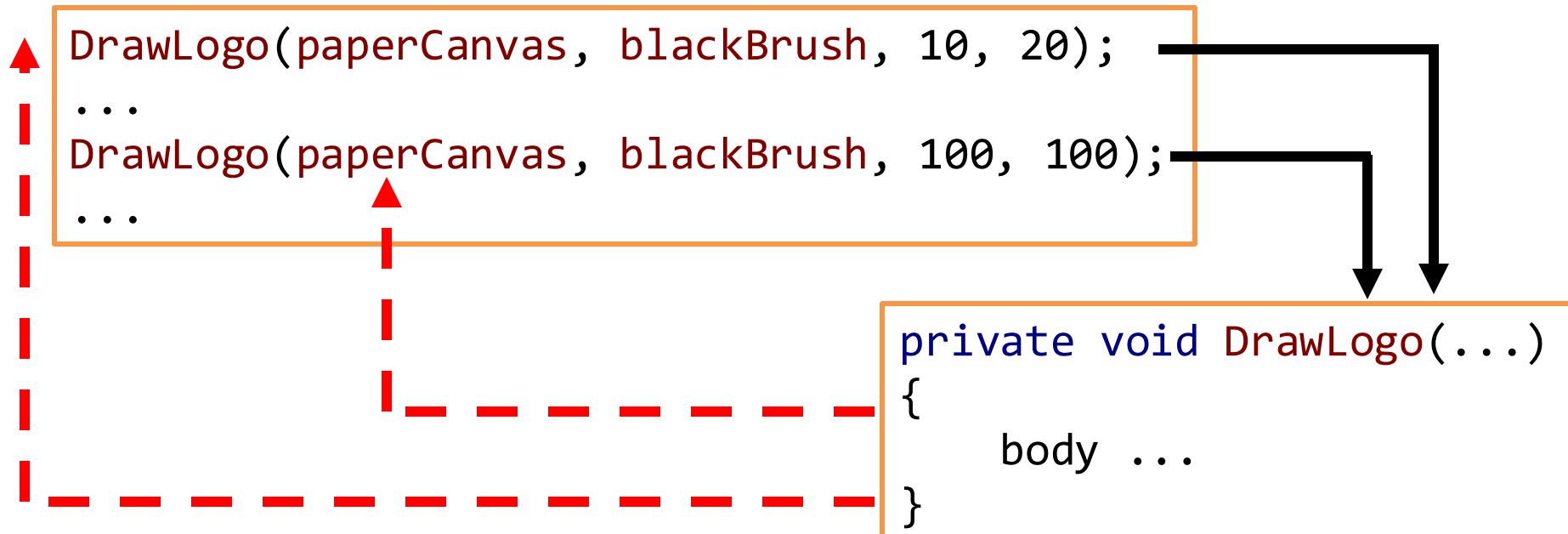


```
// Draw logo at top left
DrawRectangle(paperCanvas, blackBrush, 10, 20, 60);
DrawRectangle(paperCanvas, blackBrush, 10, 20, 40);
DrawRectangle(paperCanvas, blackBrush, 10, 20, 20);

// Draw logo at bottom right
DrawRectangle(paperCanvas, blackBrush, 100, 100, 60);
DrawRectangle(paperCanvas, blackBrush, 100, 100, 40);
DrawRectangle(paperCanvas, blackBrush, 100, 100, 20);
```

# Een methode schrijven

- Demo Logo Method
- Gebruik debugger om stap voor stap door het programma te lopen



# Parameterbinding

```
private void DrawLogo(Canvas drawingArea,  
                      SolidColorBrush brushToUse,  
                      double xPosition,  
                      double yPosition)
```

parameters

```
DrawLogo(paperCanvas, blackBrush, 10, 20);
```

drawingArea

brushToUse

xPosition

yPosition

paperCanvas

blackbrush

10

20

De argumenten worden  
doorgegeven via de  
parameters =  
parameterbinding

# Spelregels voor methodes

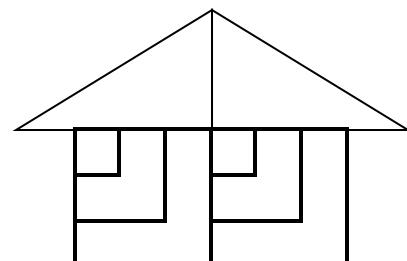
- De *schrijver* van de methode
  - Kiest de parameters
  - Kiest de parametertypes
  - Kiest een (betekenisvolle) naam
  - Kiest het type parameterbinding (zie verder)
- De *gebruiker* van de methode
  - Geeft de argumenten mee
  - In de juiste volgorde
  - Van het correcte type

# Een driehoekmethode

- Bestudeer zelf de code
- Een programma kan uit meerdere methodes bestaan
- Een methode zal dikwijls op zijn beurt andere methodes oproepen

# Een driehoekmethode

- *Bestudeer de code in het boek*
- *Kan je een andere set van parameters bedenken om deze methode te schrijven?*
- *Werk de implementatie uit (op papier)*
- *Welke oproepen zijn nodig voor volgende afbeelding ...*



# Wat zijn de lokale variabelen?

```
private void DrawTriangle2(Canvas drawingArea,
                           SolidColorBrush brushToUse,
                           double topX,
                           double topY,
                           double width,
                           double height)
{
    double rightCornerX, rightCornerY;
    rightCornerX = topX + width;
    rightCornerY = topY + height;

    DrawLine(drawingArea, brushToUse, topX, topY
    DrawLine(drawingArea, brushToUse, topX, rightCornerY,
             rightCornerX, rightCornerY);
    DrawLine(drawingArea, brushToUse, topX, topY,
             rightCornerX, rightCornerY);
}
```

# Lokale variabelen

- Hulpmiddel voor berekeningen
- Zijn gedeclareerd binnen een methode body
- Ze bestaan enkel tijdens de methode oproep  
→ *lokaal bereik* of *scope*
- Lokale variabelen zijn enkel zichtbaar binnen een methode body
- 2 methodes met lokale variabelen met dezelfde naam zijn dus toegestaan

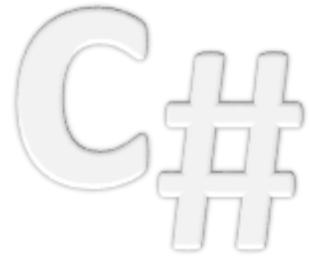
# Optionele parameters

- Normale aanroep: juiste type, correcte volgorde van de parameters
- De schrijver kan echter opteren voor standaardwaarden van argumenten
- De aanroeper hoeft enkel die parameters mee te geven die afwijken van deze standaardwaarden

# Optionele parameters

```
private void DrawTriangle3(Canvas drawingArea,  
                           SolidColorBrush brushToUse,  
                           double topX = 0,  
                           double topY = 0,  
                           double width = 30,  
                           double height = 30)
```

```
// other parameters get default values  
DrawTriangle3(paperCanvas, blackBrush);  
  
// values for topX and topY  
DrawTriangle3(paperCanvas, blackBrush, 50, 50);
```



# Optionele parameters

```
private void DrawTriangle3(Canvas drawingArea,  
                           SolidColorBrush brushToUse,  
                           double topX = 0,  
                           double topY = 0,  
                           double width = 30,  
                           double height = 30)
```

Hoe **enkel** de standaardwaarden voor `width` en `height` veranderen?

```
// named parameters  
DrawTriangle3(paperCanvas, blackBrush, width: 25, height: 40);  
  
// volgorde wisselen mogelijk  
DrawTriangle3(height: 150, width: 150,  
              drawingArea: paperCanvas, brushToUse: blackBrush);
```

# Naamconflicten

```
private void MethodOne(int x, int y)
{
    int z = 0;
    // code...
}
```

```
private void MethodTwo(int z, int x)
{
    int w = 1;
    // code...
}
```

```
int w = 10;
...
MethodTwo(w, 5);
...
```

# Het begrip Klasse

- Een klasse kan een willekeurig aantal methoden in een willekeurige volgorde bevatten
- Tot hiertoe, slechts 1 klasse: `MainWindow`
- Methodes
  - Zelfgeschreven (methodes en functies)
  - Event-handling (`private void button1_Click`)

# Klasse: Voorbeeld

MainWindow
-button1: Button
-components: IContainer
-button1_Click: void
-SomeMethod: void
-SomeOtherMethod: void
-InitializeComponent: void

- UML Notatie
- Instantievariabelen  
(zie hoofdstuk 6)
- Zelf geschreven methodes in  
MainWindow.xaml.cs
  - button1\_Click
- Gegenereerde en  
zelfgeschreven Xaml-code in  
MainWindow.xaml
- methode InitializeComponent  
in MainWindow.g.i.cs

# Functiemethoden

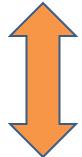
```
private int CalculateAreaOfRectangle(int length, int width)
{
    int area;
    area = length * width;
    return area;
}
```

- Een functie retourneert (precies) één waarde
- Return type declareren
- Gebruik: altijd via een toekenning

# Bouwen op methoden

- Bestudeer de code
- Waarom schrijf je geen objectnaam voor de methode-aanroep DrawTriangle(...)?

```
paperCanvas.Children.Add(...);
```



```
DrawTriangle(...);
```

# Argumenten als referentie

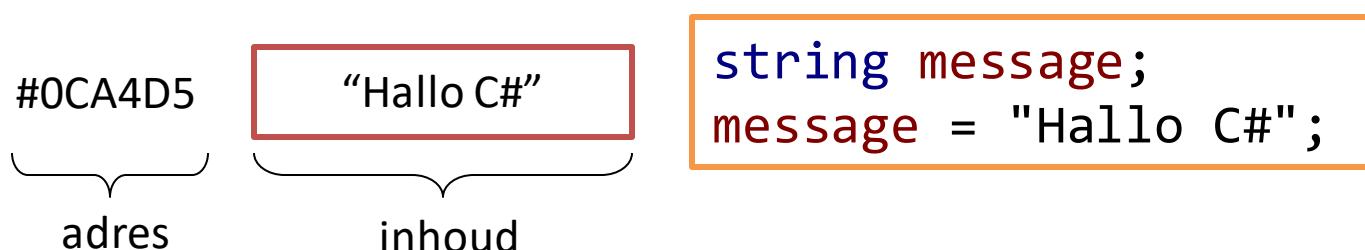
- Analogie met doorgeven van documenten
- We geven hier een iets “exactere” beschrijving van de werkelijkheid aan de hand van een geheugenmodel
- Het is van zeer groot belang om het verschil tussen doorgeven van parameters “als waarde” en “als referentie” te kunnen beschrijven, verklaren en gebruiken!

# Geheugengebruik

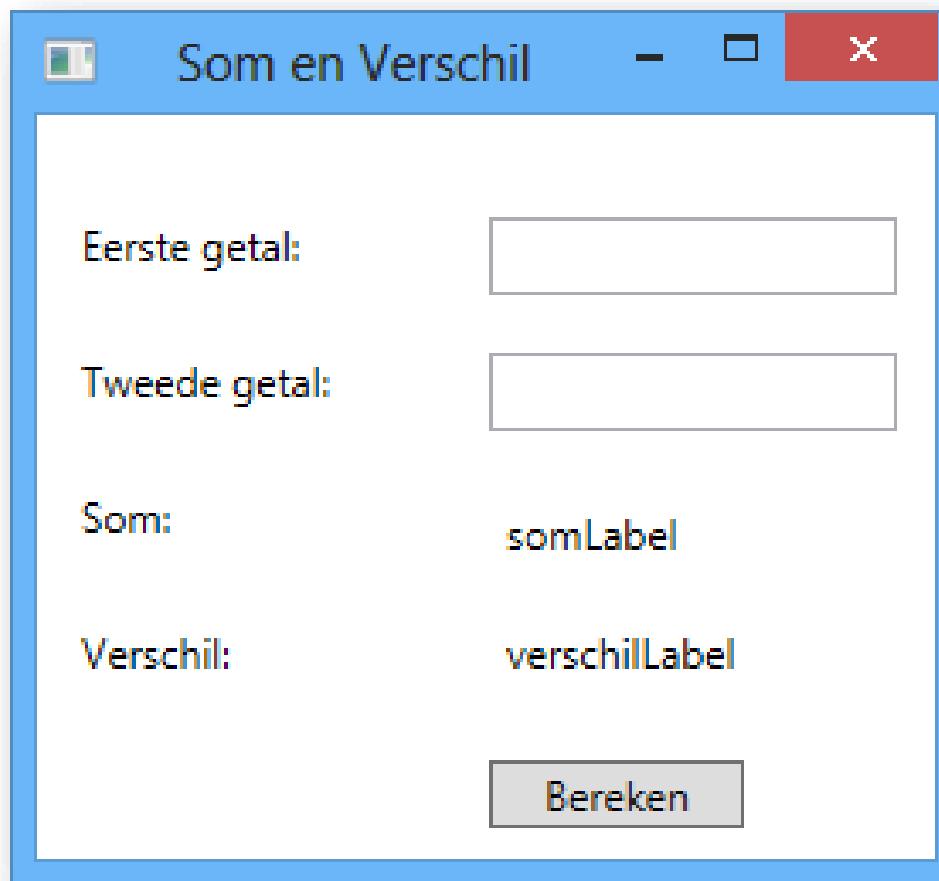
- Een computerprogramma dat wordt uitgevoerd, heeft geheugen nodig
  - Het programma zelf
  - Variabelen, objecten, ...
- De benodigde ruimte wordt vrijgemaakt in het RAM geheugen
- Indien het RAM geheugen zou vol zijn, wordt geheugen op schijf aangesproken  
(Swap space, virtueel geheugen)

# Geheugenadressering

- Elke locatie in het geheugen, heeft een adres
- Denk aan een “doos” met een bepaalde inhoud en een bepaald nummer (adres)
- Dit adres wordt in een programmeertaal voorgesteld door de naam van een variabele



# Een programma met 2 functies



# Een programma met 2 functies

```
private int ComputeSum(int n1, int n2)
{
    return n1 + n2;
}

private int ComputeDifference(int n1, int n2)
{
    return n1 - n2;
}
```

# Parameterbinding als waarde

*De inhoud van de geheugenlocaties worden gekopieerd en via de parameters doorgegeven*

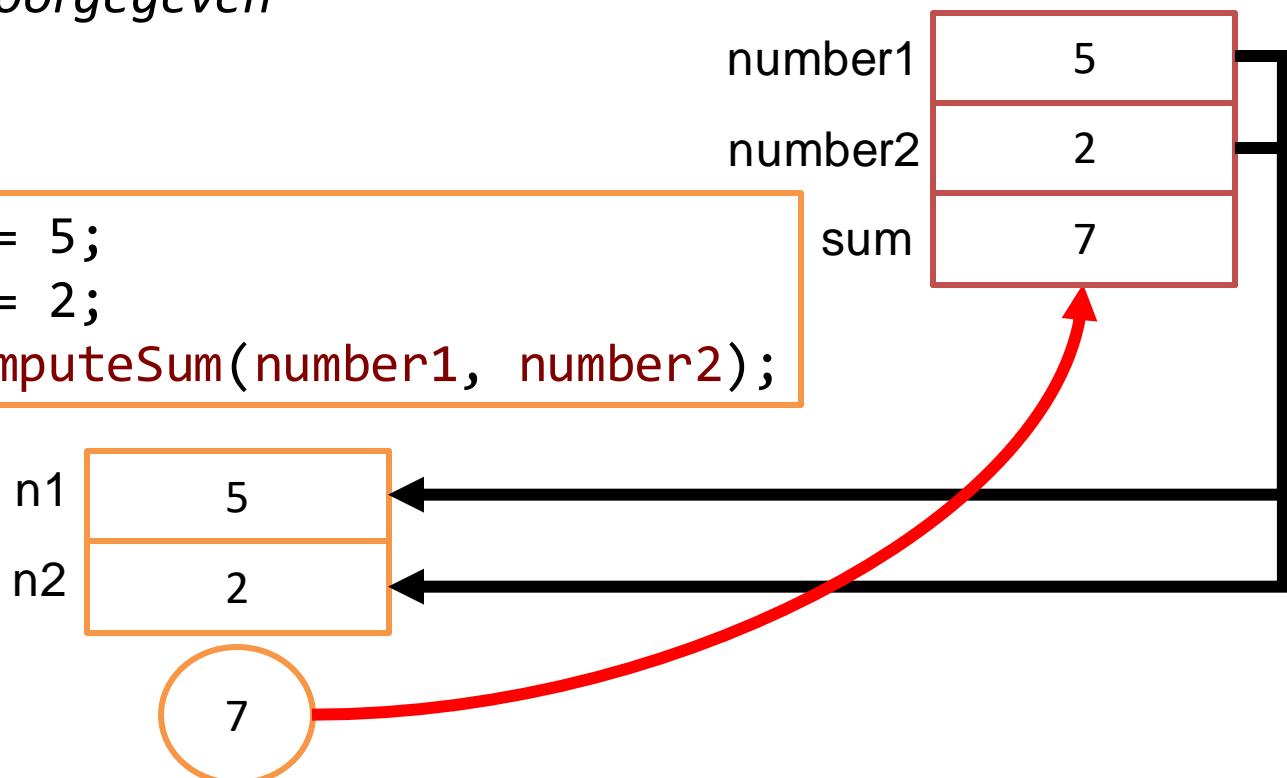
```
int number1 = 5;  
int number2 = 2;  
int sum = ComputeSum(number1, number2);
```

number1	5
number2	2
sum	?

# Parameterbinding als waarde

De inhoud van de geheugenlocaties worden gekopieerd en via de parameters doorgegeven

```
int number1 = 5;  
int number2 = 2;  
int sum = ComputeSum(number1, number2);
```



# Parameterbinding als waarde

*Wat denk je van volgende code?*

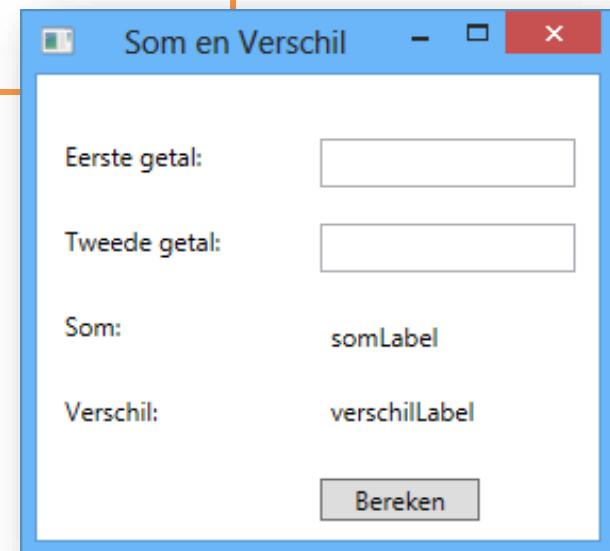
*Maak een tekening van de geheugenlocaties.*

```
private int ComputeSum(int n1, int n2)
{
    int s = n1 + n2;
    n1 = n1 + 5;
    n2 = n2 + 1;

    return s;
}
...
int number1 = 5;
int number2 = 2;
int sum = ComputeSum(number1, number2);
```

# Idem met 1 procedure

```
private void ComputeSumAndDifference(int n1,  
                                    int n2,  
                                    out int s,  
                                    out int d)  
{  
    s = n1 + n2;  
    d = n1 - n2;  
}
```

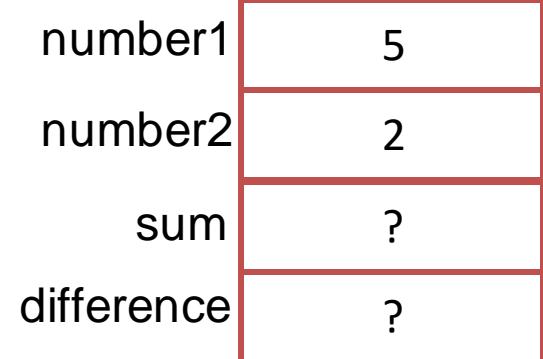


# Parameterbinding als referentie

*De adressen van de geheugenlocaties worden via de parameters doorgegeven.*

*De parameters wijzen dus naar de effectieve geheugenplaatsen!*

```
int number1 = 5;  
int number2 = 2;  
  
ComputeSumAndDifference(number1,  
    number2,  
    out int sum,  
    out int difference);
```



# Parameterbinding als referentie

De adressen van de geheugenlocaties worden via de parameters doorgegeven.

De parameters wijzen dus naar de effectieve geheugenplaatsen!

```
int number1 = 5;  
int number2 = 2;  
  
ComputeSumAndDifference(number1,  
                         number2,  
                         out int sum,  
                         out int difference);
```

number1

5

number2

2

sum

7

difference

3

n1 5

n2 2

s <adres sum>

d <adres difference>



# out vs ref

- **ref**
  - Argumenten hebben initiële waarde
  - De methode verandert deze waarde
  - = tweerichtingsverkeer
- **out**
  - Argumenten hebben nog geen waarde
  - De methode geeft deze een initiële waarde
  - = eenrichtingsverkeer

# Voorbeeld: out

```
private void calculateButton_Click(object sender, RoutedEventArgs e)
{
    int originalCents;
    originalCents = Convert.ToInt32(amountTextBox.Text);
    ExtractEurosAndCents(originalCents, out int wholeEuros,
                           out int centsLeft);
    eurosTextBlock.Text = Convert.ToString(wholeEuros);
    centsTextBlock.Text = Convert.ToString(centsLeft);
}

private void ExtractEurosAndCents(int totalCents,
                                  out int euros,
                                  out int centsLeft)
{
    euros = totalCents / 100;
    centsLeft = totalCents % 100;
}
```

Krijgen een nieuwe,  
berekende waarde

# Voorbeeld: out

```
private void calculateButton_Click(...)  
{  
    int originalCents;  
    originalCents = Convert.ToInt32(amountTextBox.Text);  
    ExtractEurosAndCents(originalCents, out int wholeEuros,  
                         out int centsLeft);  
    ...  
}
```

Beide codefragmenten zijn equivalent, maar  
we verkiezen de inline declaratie o.w.v.

- beknopt
- leesbaarder

= inline declaratie

```
private void calculateButton_Click(...)  
{  
    int originalCents, wholeEuros, centsLeft;  
    originalCents = Convert.ToInt32(amountTextBox.Text);  
    ExtractEurosAndCents(originalCents, out wholeEuros,  
                         out centsLeft);  
    ...  
}
```

# Voorbeeld: ref

```
private void button1_Click(object sender, RoutedEventArgs e)
{
    int pieWidth = 8, pieLength = 6;
    IncreaseSize(ref pieWidth, ref pieLength);
    IncreaseSize(ref pieWidth, ref pieLength);
    IncreaseSize(ref pieWidth, ref pieLength);
}

private void IncreaseSize(ref int width, ref int length)
{
    int area;
    width = width + 2;
    length = length + 2;
    area = width * length;
    MessageBox.Show($"Pie size: {width} by {length}. Area is {area}")
}
```

Krijgen telkens een update van hun waarde

# Parameterbinding: conclusie

Mechanisme	Uitleg	Gevolg	Voordeel
als waarde	Parameters zijn kopieën van de variabelen	Wijzigingen van de variabelen is onmogelijk	Beschermt variabelen tegen ongewenst overschrijven
ref	Parameters zijn referenties naar de variabelen zelf	Rechtstreeks wijzigen van de variabelen is mogelijk	Meerdere waarden kunnen teruggegeven worden
out	Parameters zijn referenties naar niet geïnitialiseerde variabelen	Rechtstreeks wijzigen van de variabelen is noodzakelijk	Meerdere waarden kunnen teruggegeven worden

# Opgelet voor misbruik!

- Schrijf enkel `ref` als het werkelijk noodzakelijk is, dus geen `ref` als de body geen wijzigingen doorgeeft
- Gebruik functies als je slechts 1 resultaat teruggeeft
- Als je `ref` gebruikt, geef dan samenhangende waarden terug. Misbruik dit mechanisme niet om methodes samen te gooien. Elke methode heeft in principe slechts één doel

# Het sleutelwoord `this`

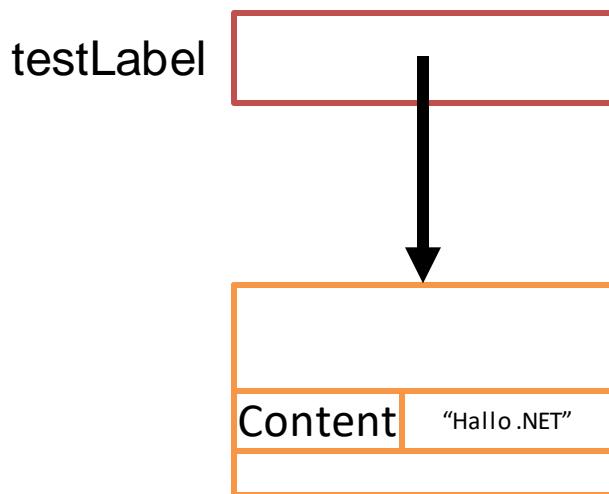
- Stelt het “huidige” object voor
- Heeft dus dezelfde betekenis als in Java

# Overloading

- Methodes/functies met
  - Dezelfde naam
  - Verschillende types/aantal parameters
- Bestudeer de methode Swap in het boek

# Geheugenadressering bij objecten

- Objecten worden bewaard op een speciale plaats in het geheugen: de heap
- Een object variabele is eigenlijk een referentie naar zo een geheugenlocatie



```
Label testLabel;  
' aanmaken met new of via XAML  
...  
testLabel.Content = "Hallo .NET";
```

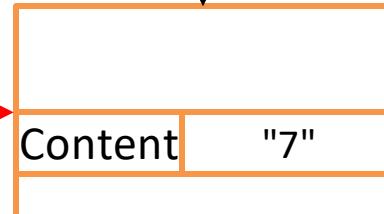
# Objecten doorgeven aan methoden

```
private void ShowSum(Label displayLabel, int a, int b)  
{  
    displayLabel.Content = Convert.ToString(a + b);  
}
```

ShowSum(resultLabel, 3, 4)

resultLabel

displayLabel



Je geeft een kopie van een referentie naar een object door. Je kan dus de properties van dit object wijzigen

# Discussie

- Bestudeer ExtractEurosAndCents / Swap
- Maak schetsen van de geheugenlocaties
- Is Swap ook te schrijven met doorgeven via waarden?
- Waarom is Swap(a,6) fout? Hoe zou de methode eruit zien als dit toegestaan zou zijn?
- Maak de tekening van ShowSum als het Label met ref zou doorgegeven worden. Zou dit een goede keuze zijn?

# Objecten

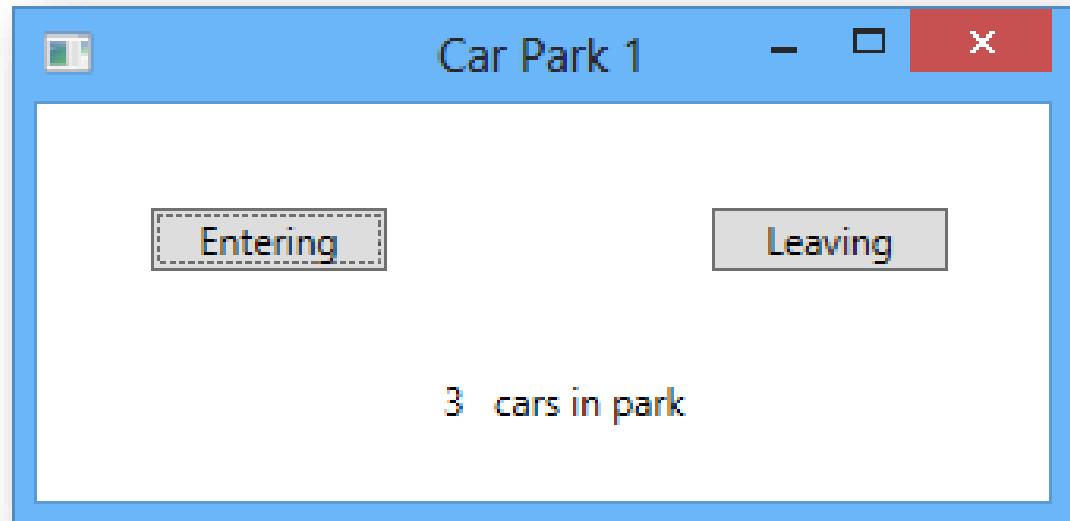
Hoofdstuk 6

# In dit hoofdstuk

- Verder uitdiepen van kennis van objecten
- Instantievariabelen
- Methoden en properties
- Bibliotheekklassen
- Het gebruik van var

# Car park programma

- Demo
- *Waarom mag carCount niet als lokale variabele per methode gedeclareerd worden?*



# Programmacode

- Hernoemen van namen van variabelen voor de duidelijkheid (bv. enterButton)
- `private int carCount = 0;`
  - Kleine letter beginnen (code convention)
  - carCount is een instantievariabele
- “class-scope”  $\leftrightarrow$  lokale scope
- `private`  $\rightarrow$  andere klassen buiten MainWindow geen toegang
- Gebruik geen instantievariabelen als “verdoken” lokale variabelen!



# Window constructor

```
public MainWindow()
{    // creates all controls based on the corresponding
    // XAML file
    InitializeComponent();

    // initialize your own variables and properties
    countLabel.Content = carCount;
}
```

# Window constructor

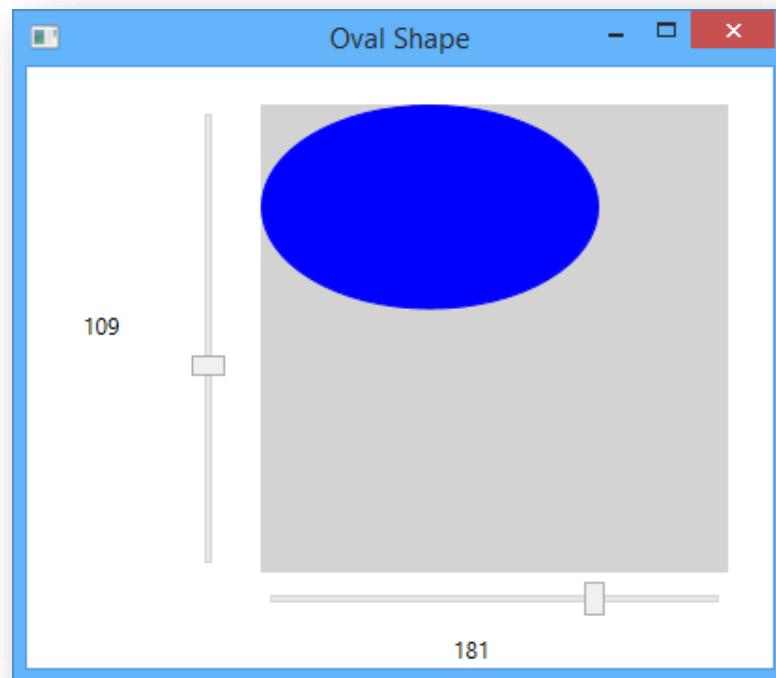
- Uitgevoerd bij het aanmaken van het MainWindow object
- Voor MainWindow objecten wordt de constructor gegenereerd door Visual Studio
- Kan door de programmeur uitgebreid worden met initialisatiecode (bv. instantievariabelen)
- Let op: toevoegen op de juiste positie!
- Demo: initialisatie carCount

```
public MainWindow()
{
    InitializeComponent();

    // here you can add code
}
```

# De klasse Slider

- Doel: een waarde instellen waarbij de exacte waarde op zich niet belangrijk is.
- Dikwijls voor auditieve of visuele instellingen (bv. geluidsniveau, kleuren, ...)
- Properties
  - Minimum
  - Maximum
  - Orientation
  - Value
- Event: ValueChanged
- Demo: Oval Shape



# De klasse Slider

- Opdracht: Oval Shape
  - Testvraag 6.3 (gebruik eventueel de debugger)
  - Zorg ervoor dat het programma correct werkt

# using en namespaces

- Alle bibliotheekklassen zijn ingedeeld in namespaces
- Zelf kunnen we ook namespaces gebruiken om de klassen van ons programma logisch te groeperen
- Om een klasse van een bepaalde namespace te gebruiken gebruik je het `using` statement
- Bepaalde namespaces worden altijd geïmporteerd; het is een goede gewoonte om ongebruikte (grijze) namespaces te verwijderen (Remove Unnecessary Usings)

# Leden, methoden en properties

- Een klasse heeft leden (eng: members)
  - Methoden om taken uit te voeren
  - Properties die de toestand weergeven
- Methoden en properties werken op object-niveau
  - Niet de klasse Label heeft een hoogte van ...
  - Wel: een Label instantie (label1) heeft een hoogte van ...
- Properties kan je instellen en/of ophalen
- Properties zijn een speciale variant van instantievariabelen (zie hoofdstuk 10)

# De klasse Random

- Een object van deze klasse kan worden aangemaakt met de constructor

```
private Random ageGuesser = new Random();
```

- Aan dit object kan je dan willekeurige getallen uit een bereik vragen

```
int value = ageGuesser.Next(5, 110);
```

5 inbegrepen, 110 niet

# De klasse Random

- De Random klasse is een component die geen visuele weergave at-run-time heeft <-> visueel: de components (bv. Canvas)
- Demo: Guesser
- *Waarom is er (g)een using statement nodig?*
- *Bestaan er nog andere constructors en/of methoden om getallen te kiezen? Hoe heet dit OO principe?*

# De klasse DispatcherTimer

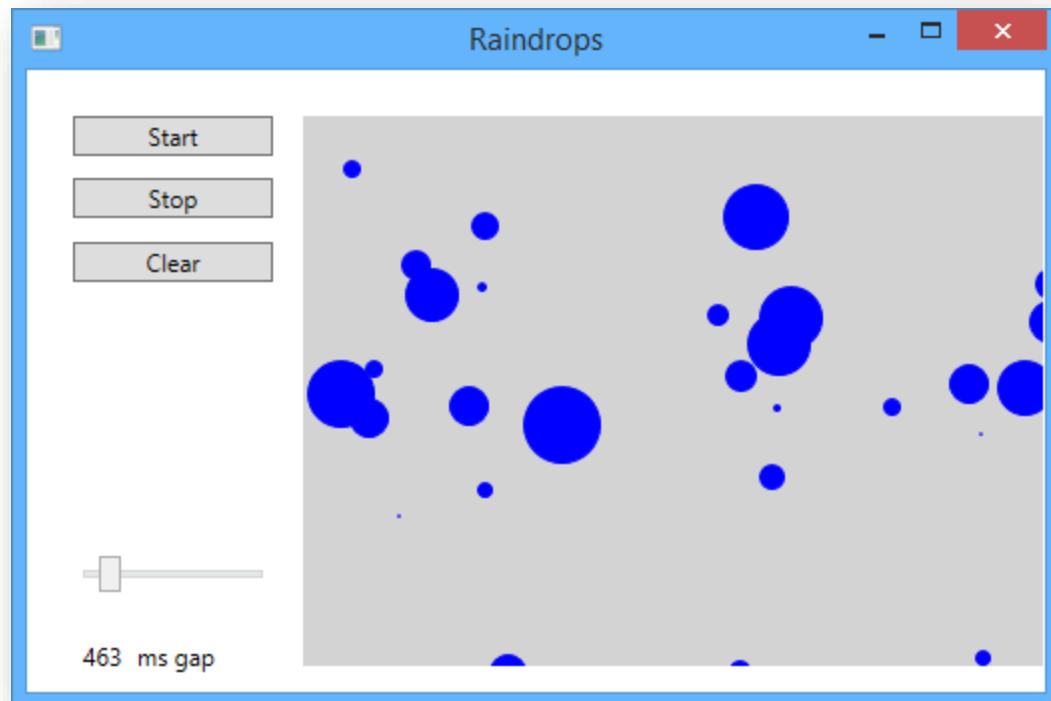
- Object aanmaken met de constructor

```
private DispatcherTimer timer = new DispatcherTimer();
```

- Namespace: System.Windows.Threading
- Demo: Raindrops

# De klasse DispatcherTimer

- Methoden
  - Start()
  - Stop()
- Property
  - Interval
- Event
  - Tick



# De klasse DispatcherTimer

- Event-handler met C#-code installeren  
(eventhandler met += met event verbinden)  
`timer.Tick += timer_Tick;`
- Als je “vergeet” de Minimum property van de Slider in te stellen, krijg je een fout. Verklaar!

# Object initializer

```
//ellipse = new Ellipse();
//ellipse.Width = size;
//ellipse.Height = size;
//ellipse.Stroke = brush;
//ellipse.Fill = brush;
//ellipse.Margin = new Thickness(x, y, 0, 0);

ellipse = new Ellipse()
{
    Width = size,
    Height = size,
    Stroke = brush,
    Fill = brush,
    Margin = new Thickness(x, y, 0, 0)
};
```

= moderne C#-code  
!Let op de plaats  
van de ; en de ,

# Het gebruik van var

```
Ellipse ellipse = new Ellipse();
```

mag je ook schrijven als:

```
var ellipse = new Ellipse();
```

= moderne C#-notatie

- = impliciet getypeerde lokale variabele
- type van ellipse is Ellipse en kan niet gewijzigd worden ⇔ bv Javascript
- enkel voor lokale variabelen (niet voor instantievariabelen)
- verplicht het object te initialiseren

```
var ellipse; // illegal!
```

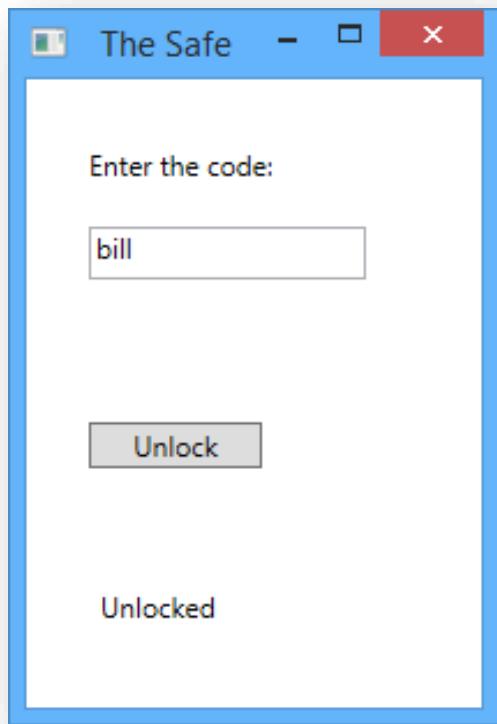
# Beslissingen

Hoofdstuk 7

# In dit hoofdstuk ...

- In elke programmeertaal bestaat de mogelijkheid om beslissingen te nemen
- C# : `if` en `switch`

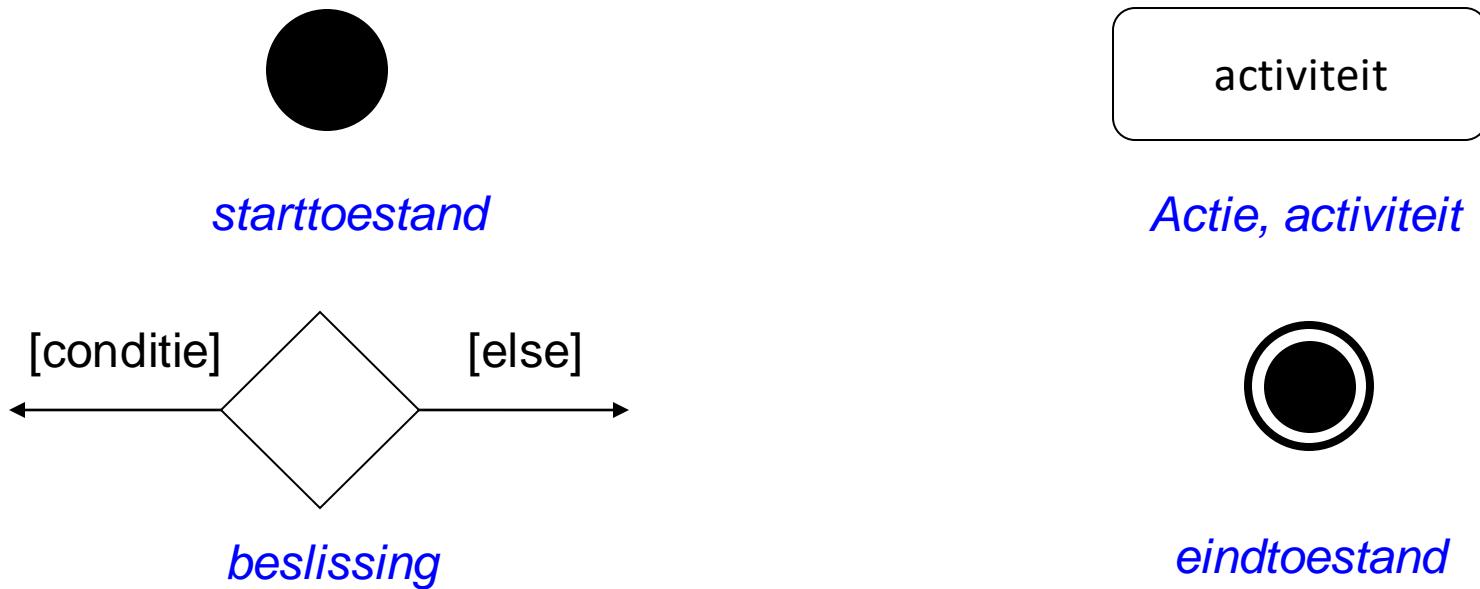
# if



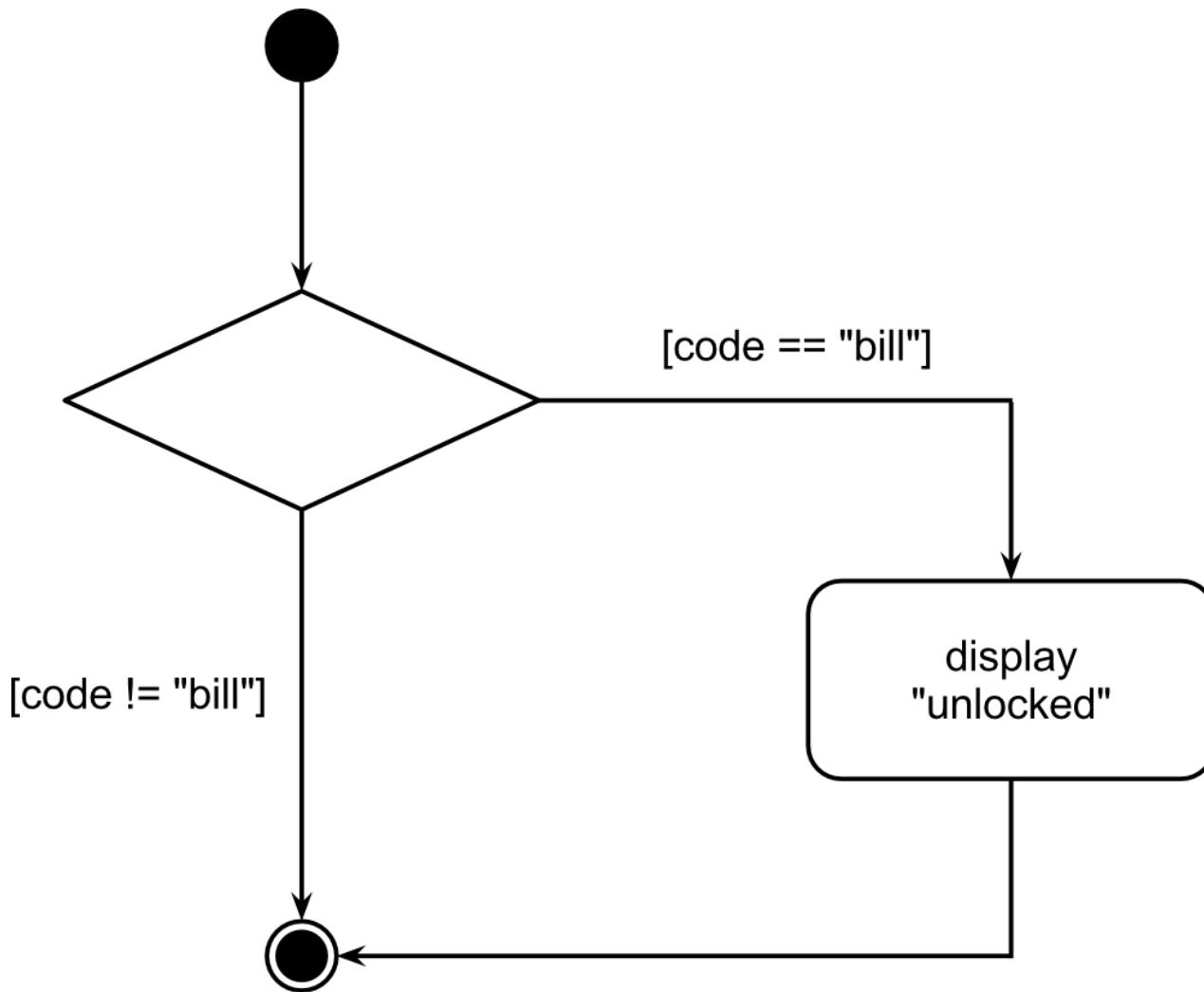
```
private void unlockButton_Click(object sender,  
                               RoutedEventArgs e)  
{  
    string code;  
  
    statusLabel.Content = "";  
    code = codeTextBox.Text;  
  
    if (code == "bill")  
    {  
        statusLabel.Content = "Unlocked";  
    }  
}
```

Inspringen → leesbaarheid

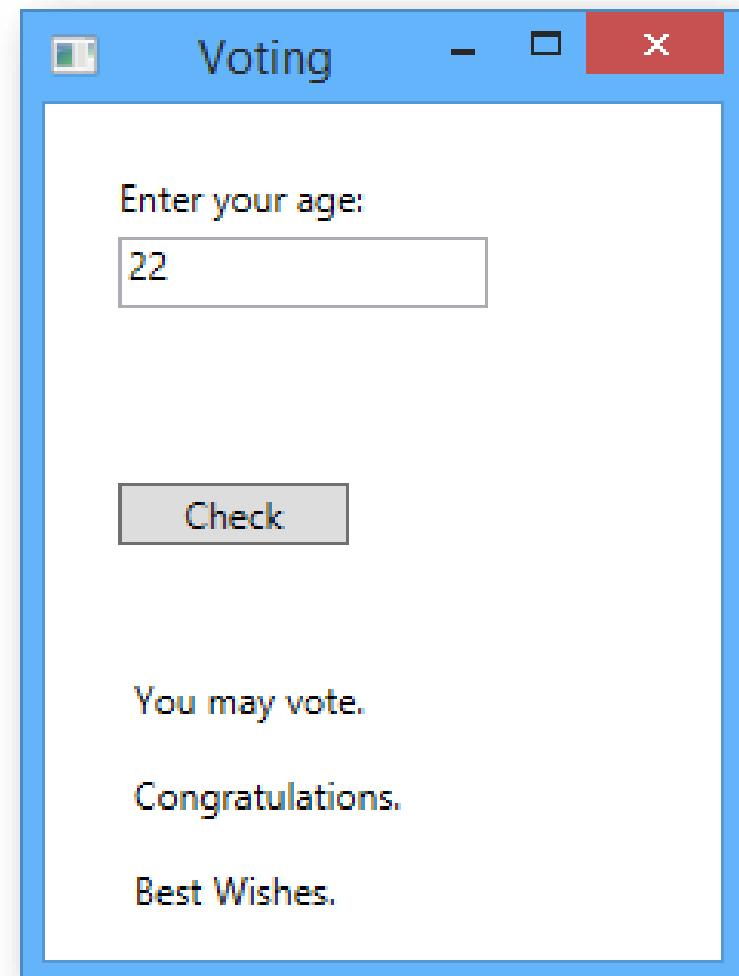
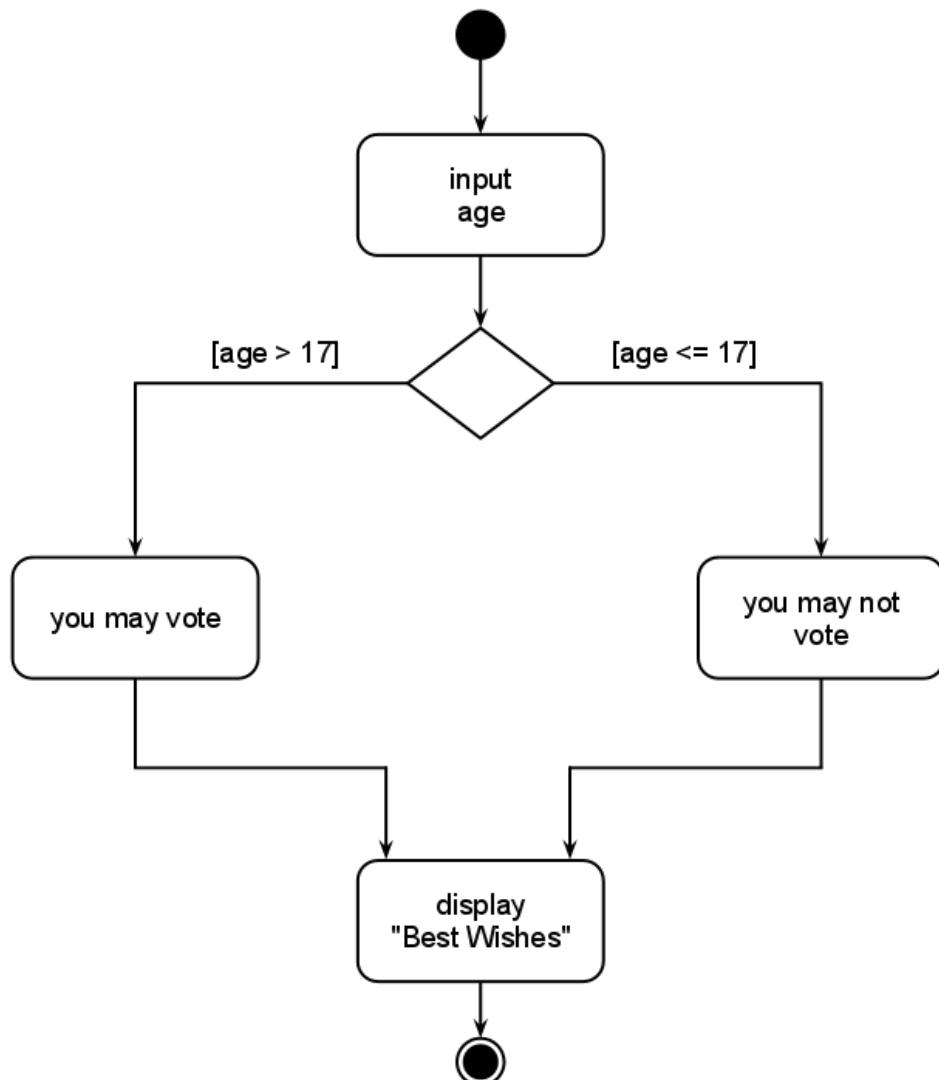
# UML Activity diagram



# UML Activity diagram



# if ... else





# if ... else

```
private void checkButton_Click(object sender, RoutedEventArgs e)
{
    int age;
    age = Convert.ToInt32(ageTextBox.Text);

    if (age > 17)
    {
        decisionLabel.Content = "You may vote.";
        commentaryLabel.Content = "Congratulations.";
    }
    else
    {
        decisionLabel.Content = "You may not vote.";
        commentaryLabel.Content = "Sorry.";
    }
    signOffLabel.Content = "Best Wishes";
}
```

# Vergelijgingsoperatoren

Symbool	Betekenis
>	groter dan
<	kleiner dan
==	is gelijk aan
!=	ongelijk aan
<=	kleiner of gelijk aan
>=	groter of gelijk aan

# Voorwaarden combineren

- **&& operator**

```
if ((age > 6) && (age < 16))  
{...}
```

- **|| operator**

```
if ((age > 16) || (age < 6))  
{...}
```

- **! operator**

```
if (!(age > 17))  
{...}
```



Gebruik haakjes

# Voorwaarden combineren

- *Herschrijf volgende voorwaarde*

```
if (!a > 6) && !(b >= 16))  
{...}
```

- *Herschrijf nogmaals vorige voorwaarde, maar gebruik enkel not en or. Hoe heet deze regel (Logica)?*

# Dice programma

- *Bestudeer het dice programma*
  - Attributen van Slider:
    - *TickPlacement="BottomRight"*
    - *IsSnapToTickEnabled="true"*
  - Waarom is het nodig om de evenHandlers via `+=` met de events te verbinden?
  - Tip: werk met 1x de Tab-toets om een handler-methode aan te maken

# Dice programma

- *Los de testvragen op*
- *Hoe kan je je programma robuuster maken, zodat CheckValues() niet meer afhankelijk is van de twee Slider controls?*  
*(Hint: hoofdstuk 5)*

# Geneste ifs

```
if (age > 6)
{
    if (age < 16)
    {
        decisionLabel.Content = "junior rate.";
    }
    else
    {
        decisionLabel.Content = "adult rate.";
    }
}
else
{
    decisionLabel.Content = "child rate.";
}
```

# Vereenvoudigen ifs

```
if ((age > 6) && (age < 16))
{
    decisionLabel.Content = "junior rate.";
}
else if (age >= 16)
{
    decisionLabel.Content = "adult rate.";
}
else
{
    decisionLabel.Content = "child rate.";
}
```

# Geneste ifs

- *Bestudeer “Tom en Jerry”, herschrijf de if structuur als er 3 schuifregelaars zouden zijn*
- *Testvraag 7.8, schrijf de if structuur*

# switch

```
if (dayNumber == 1)
{
    dayName = "Monday";
}
else if (dayNumber == 2)
{
    dayName = "Tuesday";
}
else if ...
...
else if (dayNumber == 7)
{
    dayName = "Sunday";
}
```

```
switch (dayNumber)
{
    case 1:
        dayName = "Monday";
        break;
    case 2:
        dayName = "Tuesday";
        break;
    ...
    case 7:
        dayName = "Sunday";
        break;
}
```

# switch

- Meerdere opdrachten binnen case
- Meerdere opties per case
- Default optie

```
case 6:  
    MessageBox.Show("Hoera");  
    dayName = "Saturday";  
    break;
```

```
case 6:  
case 7:  
    dayName = "Weekend";  
    break;
```

```
switch (dayNumber)  
{  
    ...  
    default:  
        dayName = "Illegal day";  
        break;  
}
```

# switch: aandachtspunten

- Het `break` statement is verplicht
- Een `default` gedeelte is altijd aan te raden
  - Detectie onverwachte waarden (bugs)
  - `MessageBox.Show`  
of  
`Debug.WriteLine`

# Het type bool

- Twee waarden: true en false
- Verkorte schrijfwijze voor System.Boolean
- Dit type kan je gebruiken net als alle andere types
  - Instantievariabelen
  - Lokale variabelen
  - Methode/functie argumenten
  - Functie return types
- Demo: Remember

# De ternaire operator ?:

*Resultaatvariabele = voorwaarde ? eerste\_expressie : tweede\_expressie;*

```
if (filled)
{
    rectangle.Fill = brush;
}
else
{
    rectangle.Fill = null;
}
```

```
rectangle.Fill = filled ? brush : null;
```

# Herhalingen

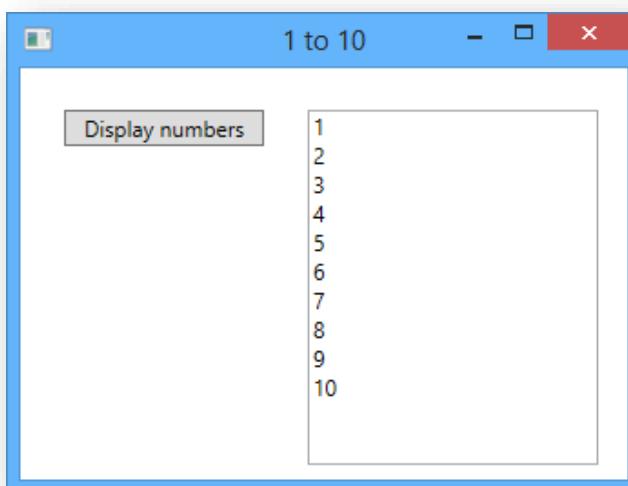
Hoofdstuk 8

# In dit hoofdstuk

- In elke programmeertaal bestaat de mogelijkheid om herhalingen of lussen uit te voeren
- C#: for, while en do

# for

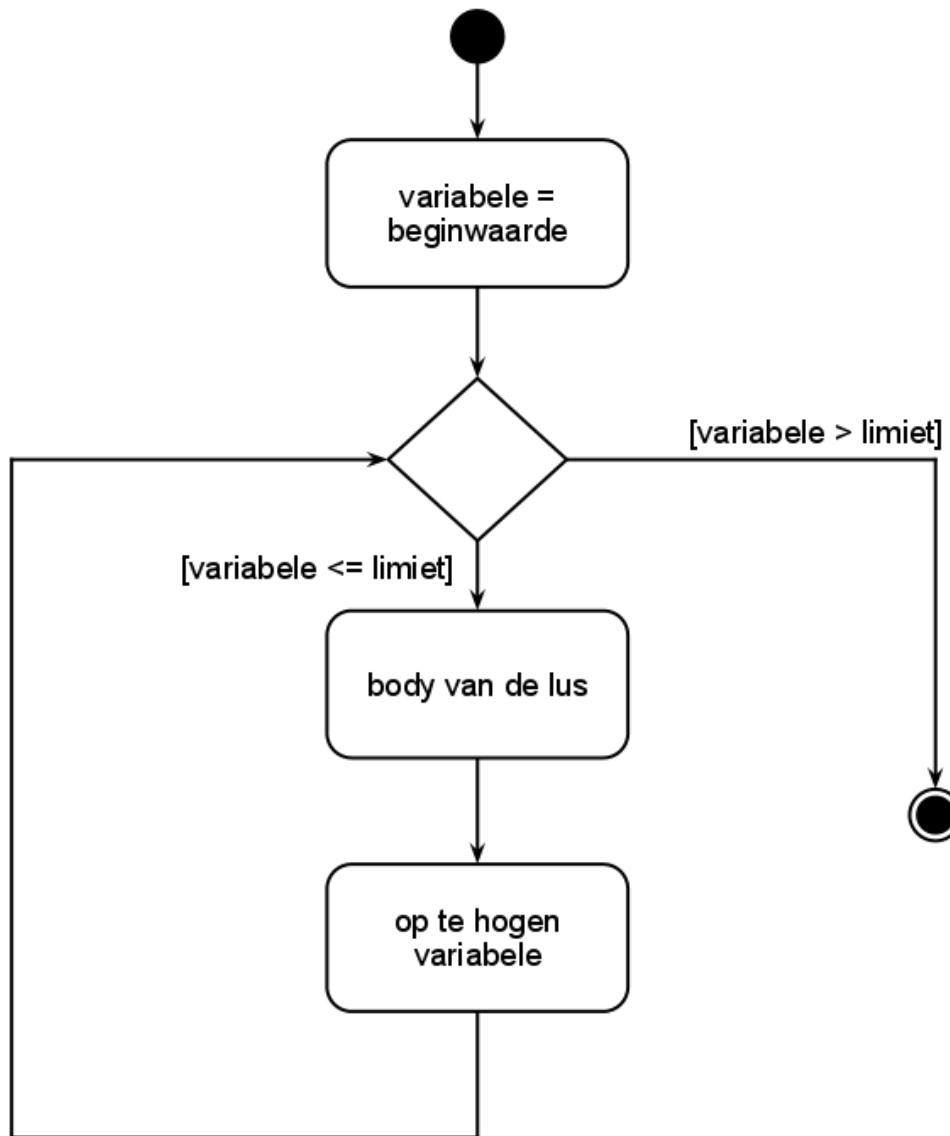
```
private void displayButton_Click(object sender,
                                RoutedEventArgs e)
{
    numbersTextBox.Clear();
    for (int number = 1; number <= 10 ; number++)
    {
        numbersTextBox.AppendText(number +
                                  Environment.NewLine);
    }
}
```



# for

- De *body* van de lus wordt herhaaldelijk uitgevoerd
- Declaratie van lusvariabele **number** meestal lokaal in de lus
- Nieuwe methoden
  - `numbersTextBox.Clear();`
  - `numbersTextBox.AppendText(...);`
- Nieuwe properties
  - `Environment.NewLine`
  - Is [Environment](#) een namespace of een class?
  - Welke andere properties zijn nog gedefinieerd?
  - Welk(e) karakter(s) wordt hiermee toegevoegd? Ken je nog zulke karakters?
- Demo:
  - Gebruik de debugger om de lus te volgen
  - Een andere (primitievere) truc  
`MessageBox.Show(Convert.ToString(number));`

# for: activity-diagram



# for

- *Beschouw volgend stukje code, hoeveel keer wordt de lus uitgevoerd?*

```
for (int number = 1; number <= 10 ; number++)
{
    numbersTextBox.AppendText(number +
                                Environment.NewLine);
    number = 10;
}
```

# for: conclusie

- Gebruik een **for-lus** als je op voorhand weet hoeveel keer de body moet uitgevoerd worden
- De lusvariabele (**number**) zou **nooit** mogen veranderd worden in de body
  - Hiervoor bestaat de **while** lus!
- De lusvariabele mag wel gelezen of geraadpleegd worden

# for: varianten

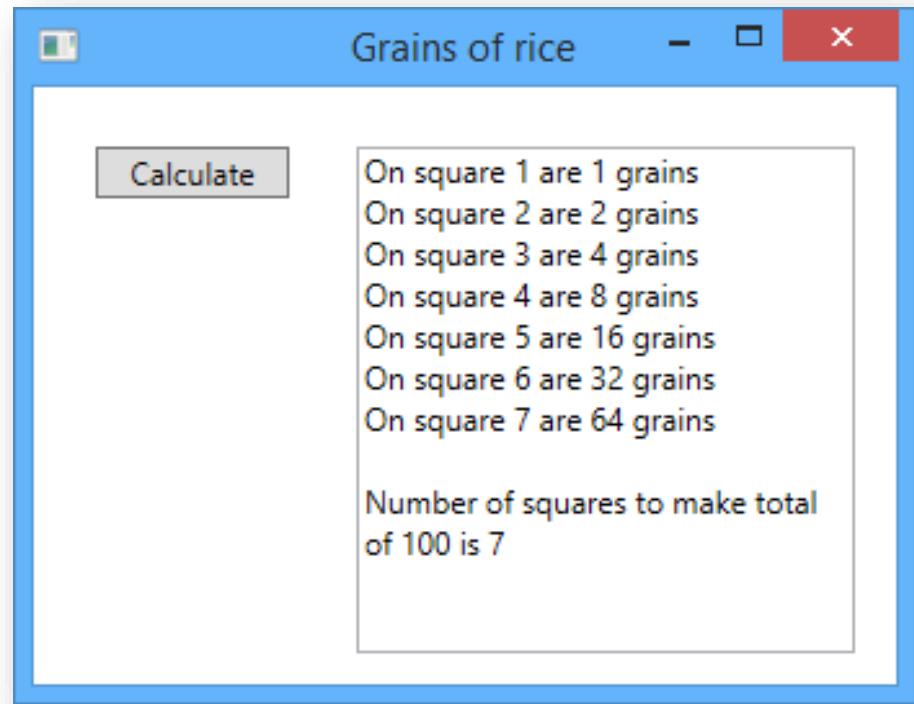
- *Denk logisch na:*
  - Hoe kan je het programma “1 To 10” veranderen, zodat het in stappen van 5 springt? Dus de uitvoer wordt: 0, 5, 10
  - Hoe kan je het programma “1 To 10” veranderen, zodat het van 10 naar 1 aftelt?
- *Beschouw het programma “Boxes”. Hoeveel keer wordt de body van de for lus uitgevoerd, als je de slider vanaf uiterst links, 7 stappen naar rechts beweegt?*

# while

- Een **while** lus kan je gebruiken om herhalingen uit te voeren waarbij de lusvoorwaarde veel vrijer te bepalen is door de programmeur
- Voorbeelden:
  - Zolang als een variabele < 10, doe je ...
  - Zolang als een invoerveld niet is ingevuld, doe je ...
  - Zolang als de grootte van een bestand < 10kB, doe je ...
  - Zolang als je een woord niet hebt gevonden in een tekst, doe je ...
- In een **while** lus is het aantal iteraties (herhalingen) dus niet op voorhand bepaald!

# while: voorbeeld

- Per tegel verdubbelt het aantal graankorrels
- Op de hoeveelste tegel heb ik in totaal honderd graankorrels (of meer)?



# while

```
private void calculateButton_Click(object sender, RoutedEventArgs e)
{
    int numberOfSquares = 1;
    int amountOfRice = 1;
    int total = 1;
    resultTextBox.Clear();
    DisplayCounts(numberOfSquares, amountOfRice);

    while (total < 100)
    {
        numberOfSquares = numberOfSquares + 1;
        amountOfRice = amountOfRice * 2;
        DisplayCounts(numberOfSquares, rice);
        total = total + amountOfRice;
    }

    resultTextBox.AppendText(Environment.NewLine +
        "Number of squares to make total of 100 is " +
        numberOfSquares);
}
```

# while

```
private void DisplayCounts(int square, int rice)
{
    resultTextBox.AppendText($"On square {square} are {rice}
                                grains");
    resultTextBox.AppendText(Environment.NewLine);
}
```

- *Hoeveel keer wordt de body van de while lus uitgevoerd?*
- *Verander het programma, zodat het aantal rijstkorrels kan ingegeven worden door de gebruiker.*

# while

- Elke while lus bestaat uit de volgende componenten
  - Initialisatie vóór de lus (`total = 1;`)
  - Stopvoorwaarde (`total < 100`)
  - Opdrachten die de stopvoorwaarde beïnvloeden  
(`total = total + amountOfRice;`)
- Typische fouten
  - Te weinig iteraties (bv: `count < 10` of `count <= 10`)
  - Oneindige lussen (bv: vergeten `count` te verhogen)
  - Verkeerde operator voor gelijkheid (`count == 10`)

# And, Or, Not

- &&, ||, !
- Net zoals bij de if structuur kunnen deze operatoren gebruikt worden om ingewikkelde stopcondities te maken
- Haakjes!

# do . . . while

- Alternatieve herhalingsstructuur
  - Test op het einde van elke herhaling
  - De herhaling gebeurt minstens één keer

# do . . . while

```
for (int count = 0; count <= 9; count++)  
{  
    numbersTextBox.AppendText(count + " ");  
}
```

```
int count = 0;  
while (count <= 9)  
{  
    numbersTextBox.AppendText(count + " ");  
    count++;  
}
```

```
int count = 0;  
do  
{  
    numbersTextBox.AppendText(count + " ");  
    count++;  
} while (count <= 9);
```

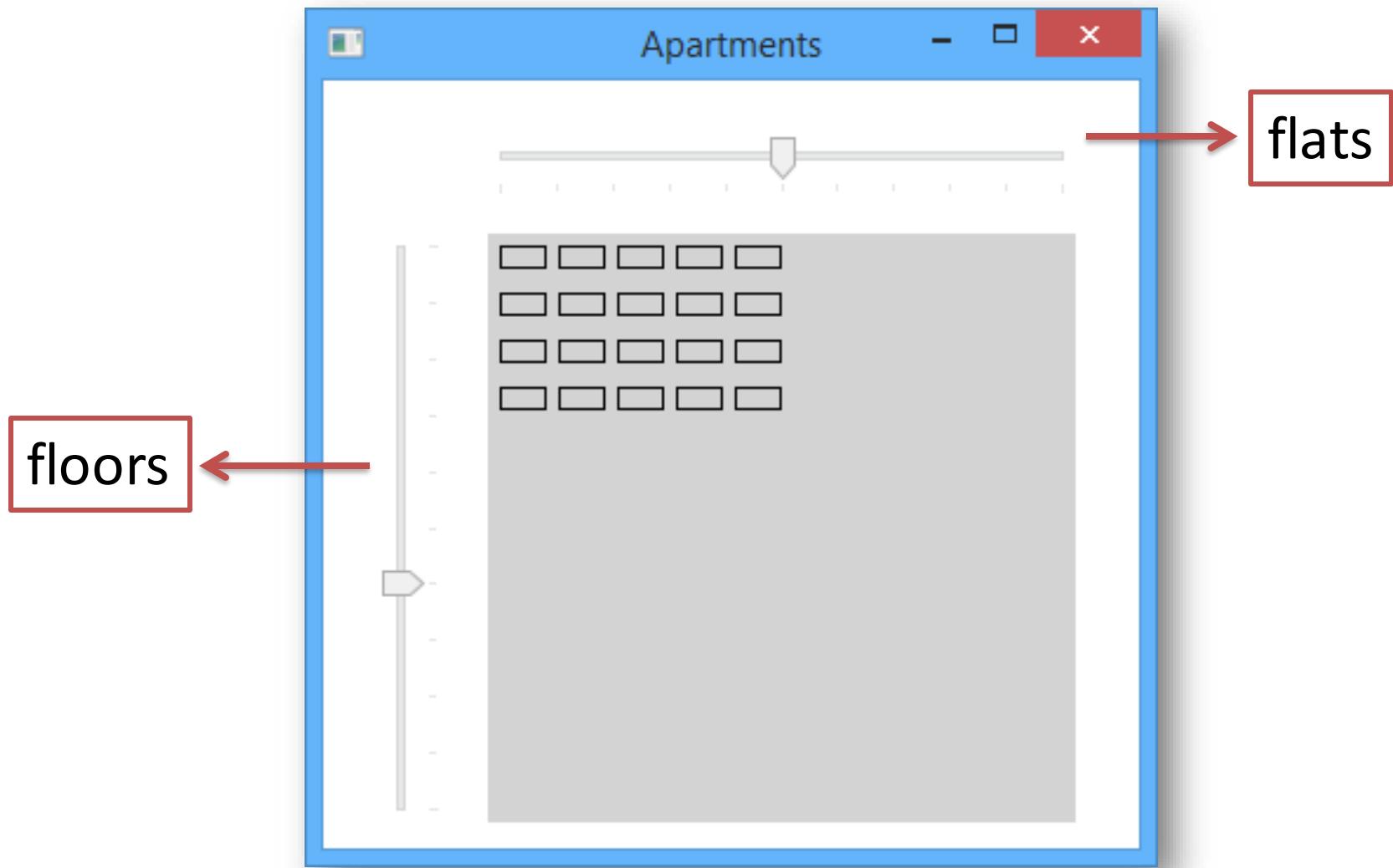
Deze 3 fragmenten  
hebben identiek  
hetzelfde resultaat!

# Bemerking

- Elke lus kan herschreven worden door middel van een while lus
- Waarom dan zoveel lus varianten?
  - Kortere schrijfwijze, bv een for is korter dan een while
  - Persoonlijke voorkeur
  - Soms lijkt code duidelijker als ze met de ene of andere variant is geschreven

**TIP:** werk zoveel mogelijk met `for`, vervolgens zoveel mogelijk met `while` en vermijd de varianten van `do ... while`

# Geneste lussen



# Geneste lussen

```
private void DrawFlats(int floors, int flats)
{
    double x, y;

    apartmentsCanvas.Children.Clear();
    SolidColorBrush brush = new SolidColorBrush(Colors.Black);
    y = 5;
    for (int floor = 0; floor < floors; floor++)
    {
        x = 5;
        for (int flat = 0; flat < flats; flat++)
        {
            DrawRectangle(apartmentsCanvas, brush, x, y, 20, 10);
            x = x + 25;
        }
        y = y + 20;
    }
}
```

# Geneste lussen

```
private void DrawRectangle(Canvas paperCanvas, SolidColorBrush  
brush, double x, double y, double width, double height)  
{  
    Rectangle rect = new Rectangle();  
    rect.Width = width;  
    rect.Height = height;  
    rect.Margin = new Thickness(x, y, 0, 0);  
    rect.Stroke = brush;  
    paperCanvas.Children.Add(rect);  
}
```

# Geneste lussen

- Als je dergelijke oefeningen zelf moet opstellen, denk je vaak best “bottom up”
  - DrawOneFlat() → code om één flat te tekenen van n verdiepingen
    - Dit is dus een simpeler probleem
  - Herhaal de vorige methode voor het gevraagde aantal flats
- Als je methodes gebruikt, bevordert dit eveneens de leesbaarheid

# Geneste lussen

```
private void DrawOneFlat(int floors, double startX,
                        Canvas apartmentsCanvas, SolidColorBrush brush)
{
    double y = 5;
    for (int floor = 0; floor < floors; floor++)
    {
        DrawRectangle(apartmentsCanvas, brush, startX, y, 20, 10);

        y = y + 20;
    }
}
```

# Geneste lussen

```
private void DrawFlats(int floors, int flats)
{
    double x, y;
    apartmentsCanvas.Children.Clear();
    SolidColorBrush brush = new SolidColorBrush(Colors.Black);
    x = 5;
    for (int flat = 0; flat < flats; flat++)
    {
        DrawOneFlat(floors, x, apartmentsCanvas, brush);
        x = x + 25;
    }
}
```

# Combineren van controlestructuren

- Bestudeer het programma: “Bouncing ball”
- *Waarom verschijnt er een “spoor” van de bal?  
Je ziet a.h.w. zeer veel ballen ...*
- *Verander dit programma, zodat je effectief één bal ziet stuiteren*
  - Hint: *DispatcherTimer*

# Visual Studio tip

- Om snel een lus in te typen schrijf je het eerste woord van de lus (bv. `for`) gevolgd door tweemaal tab.
- Een skelet van de structuur wordt gegenereerd
- Dit werkt ook voor `if` en `switch`

# Debuggen

Hoofdstuk 9

# In dit hoofdstuk ...

- Verschillende typen programmeurfouten
- Gebruik van de debugger
- Breakpoints, single stepping en run to click
- Safe navigation en nullable types
- XAML debugging
- Veelvoorkomende fouten

# Inleiding

- Bug = fout in een programma
- Hoe opsporen?
  - Primitieve hulpmiddeltjes
    - MessageBox.Show(...)
  - Debugger
    - Stap voor stap door het programma lopen en de inhoud van variabelen bekijken
  - Lezen van broncode (“Code reading”)
    - Programmeurs checken mekaar's code en trachten zo fouten eruit te halen
  - Inspecteren van
    - Log bestanden: dit zijn tekstbestanden die een programma genereert tijdens de uitvoering.
    - Core dump: als een programma crasht, dan genereert het een bestand met daarin de volledige toestand van de processor, geheugen, etc.
- Oplossen van bugs is voor omvangrijke programma's geen eenvoudige taak en is een discipline op zich

# Debuggen: enkele bemerkingen

- Fouten opsporen is soms erg moeilijk
  - Bv. een programma met 10 Timers gedraagt zich iedere keer weer anders.
  - Of een crash zich voordoet hangt soms af van omgevingsfactoren, bv. wel of geen netwerkverbinding
  - Een debugger is niet altijd bruikbaar: bv. een programma met 10 Timer objecten *kan* je niet stilzetten om stap voor stap te doorlopen. Meer nog, soms zijn bugs “verdwenen” precies omdat je de code stap voor stap uitvoert!

# Debuggen: enkele bemerkingen

- Fouten oplossen is soms nog moeilijker
  - Je weet in welk stuk code de fout zich voordoet, je wil deze code veranderen
  - Introduceer je geen nieuwe bugs in deze code?
  - Introduceer je geen nieuwe bugs in andere stukken code *precies omdat* je deze code hebt veranderd?
  - Soms is het beter gewoon met een *work-around* te werken. Je verandert de code niet, maar voorziet toch een oplossing door bijvoorbeeld een andere manier te geven om hetzelfde te bereiken

# Debuggen: enkele bemerkingen

- Je hebt de fout opgelost: hoe bereik je nu je klanten?
  - Patches op het internet
  - Service packs
  - ...
- Probleem: voor welke versies werken deze patches en voor welke versies levert dit juist nieuwe problemen?

# Waar kunnen bugs ontstaan?

- Compilatie (at compile time)
  - Syntaxfouten
  - Dit zijn de “eenvoudigste” fouten omdat een compiler je de foutenboodschap geeft
  - Met een goede IDE (zoals Visual Studio) krijg je zelfs tijdens het typen onmiddellijk feedback

# Waar kunnen bugs ontstaan?

- Koppelen (at link time)
  - Dit is een mechanisme om andere (binaire) code met het eigenlijke programma te verbinden
  - Bijvoorbeeld: `MessageBox.Show(...)`
    - Waar is de code voor `MessageBox`?
    - Op welke manier wordt dit “at runtime” gevonden?
    - → .NET Managed Execution Environment (appendix)

# Waar kunnen bugs ontstaan?

- Uitvoering (at runtime)
  - Dit zijn de feitelijke “bugs” die vaak moeilijk te vinden en/of op te lossen zijn
  - Soms krijg je foutberichten (Exceptions)
  - Soms krijg je onverwacht/verkeerd gedrag
    - Deterministisch: je kan precies aangeven wanneer en in welke omstandigheden de fout zich voordoet
    - Indeterministisch: de fout doet zich soms voor en dan weer niet, je kan geen precieze omschrijving geven wanneer en hoe de fout optreedt
      - Voorbeeld: een “memory leak”: als een programma voortdurend geheugen verbruikt en niets vrijgeeft gaat het na een tijd crashen, maar het precieze tijdstip is afhankelijk van het gebruik en de reeds aanwezige vrije ruimte

# Demo: de debugger gebruiken

- Breakpoints
- Watch window
- Single Stepping
- Run to click
- Case Study → *Doe dit zelf aan de hand van de tekst in het handboek*

# Demo: Safe Navigation en nullable types

```
if (random != null)
{
    generatedNumber = random.Next(5, 100);
}
```

= Defensive Programming

Als *random == null* → *generatedNumber* blijft 0

```
generatedNumber = random?.Next(5, 100);
```

? . = Safe Navigation-operator

Als *random == null* → *generatedNumber* wordt *null*

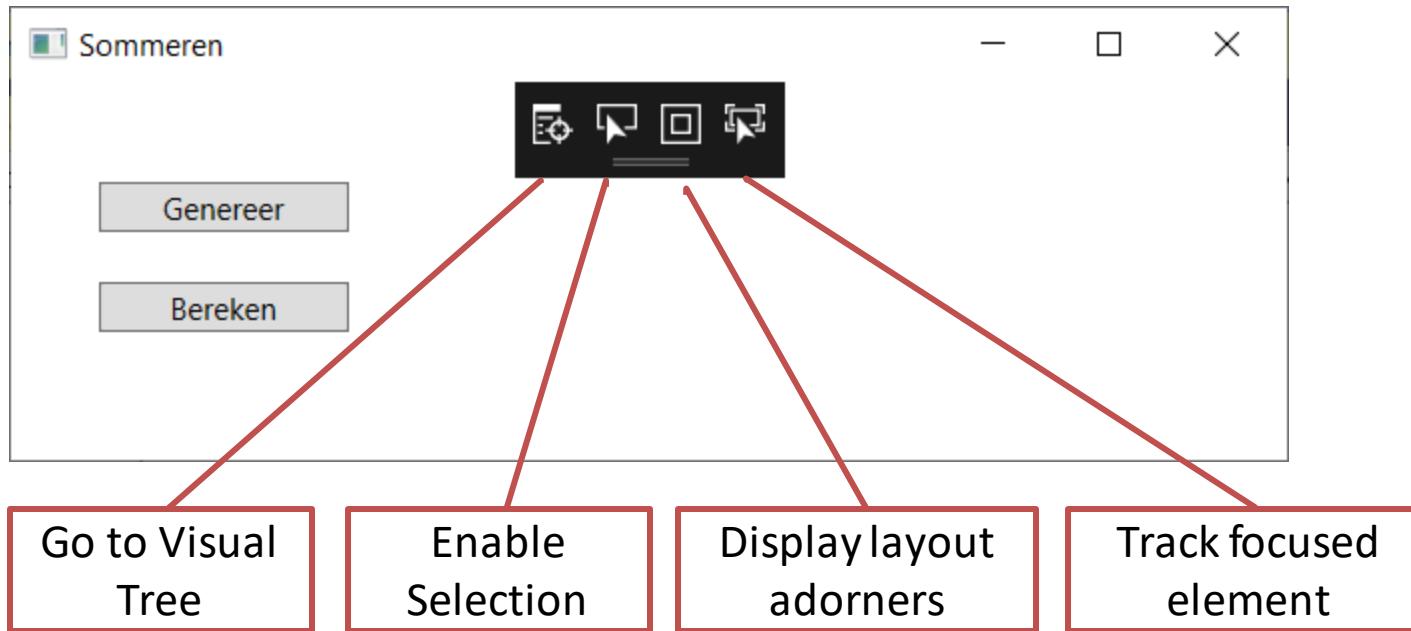
Maar vermits *generatedNumber* een *int* is (kan geen null-waarde krijgen) → als Nullable declareren → *int?*

```
private int? generatedNumber;
```

Nadeel: code wordt moeilijker leesbaar door de vele if-opdrachten

Save Navigation

# Demo: XAML debuggen



XAML-properties wijzigen tijdens runnen: klik met RM op object in Live Visual Tree > Show Properties

# Klassen schrijven

Hoofdstuk 10

# In dit hoofdstuk ...

- Hoe schrijf je een klasse
- Constructormethodes
- Publieke methoden (public)
- Variabelen
- Properties

# Inleiding

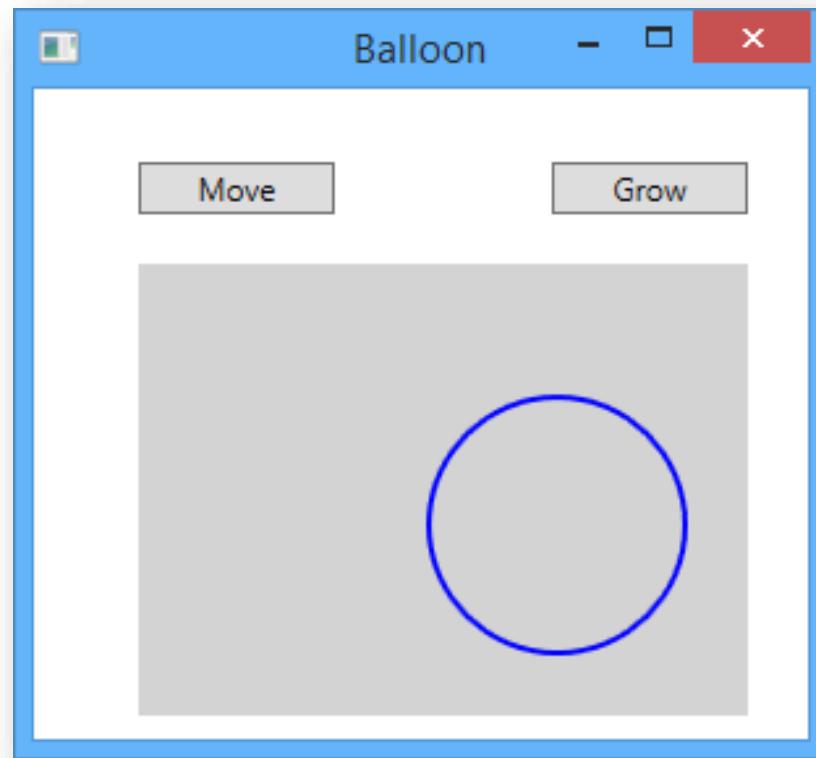
- *Wat is een klasse?*
- *Wat is een object?*
- *Hoe maak je een object?*
- *Welke klassen ken je tot hiertoe in C#?*

# Een klasse

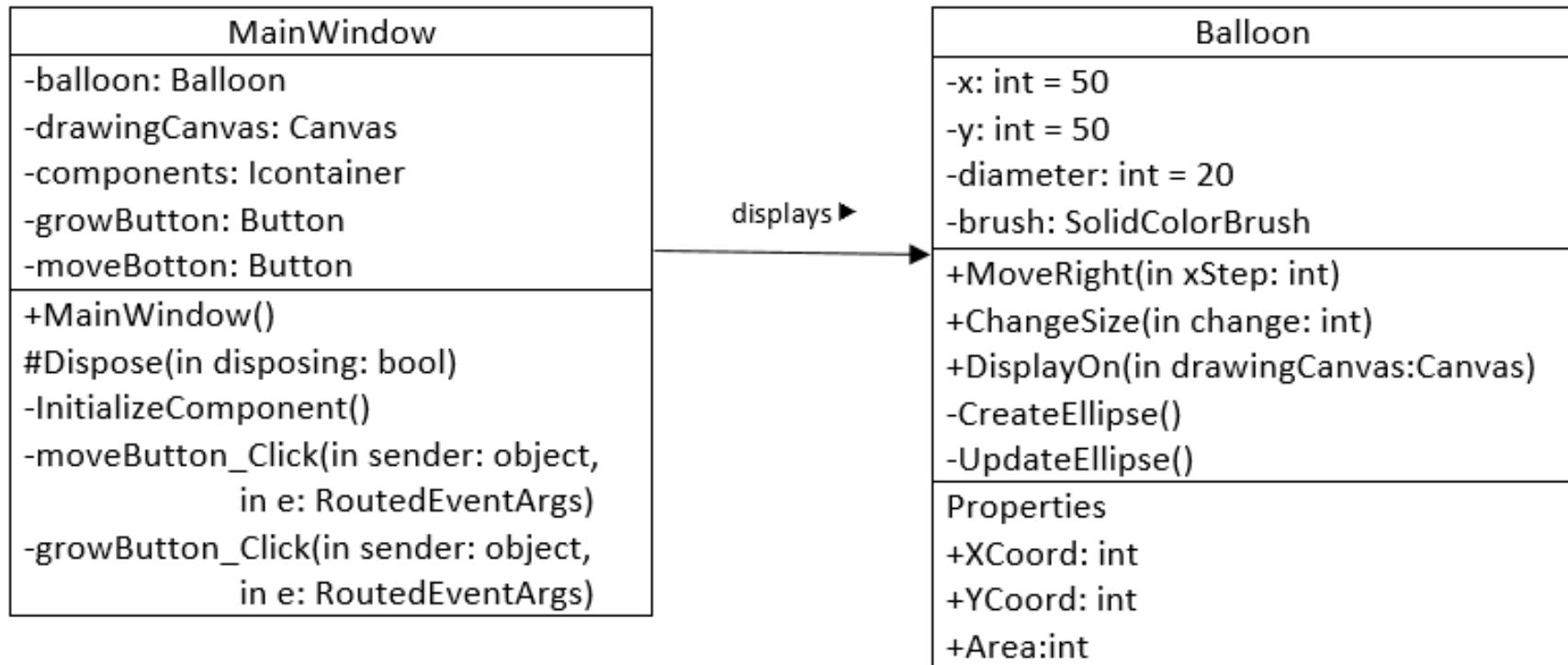
- Heeft **public** methoden
  - Aanspreekpunt voor de buitenwereld
- Heeft **properties**
  - Gecontroleerde toegang tot de toestand
- Heeft (een) **constructormethode(s)**
  - Dezelfde naam als de klasse
  - Gebruik: new
- Heeft **private** methoden
  - Hulpmethoden enkel beschikbaar binnen het object
- Heeft **private member variabelen**
  - Toestand

# Een eigen klasse ontwerpen

- Een programma dat Balloon objecten kan tekenen



# Klassediagram



Je krijgt onmiddellijk een beeld van de structuur van de applicatie

# klassediagram

- Een klasse wordt weergegeven in een rechthoek met compartimenten
  - Klassenaam
  - Member variabelen
  - Methodes
  - Properties
- Voor een variabele, methode of property staan er modifiers:
  - Een – betekent private
  - Een + betekent public
  - Een # betekent protected
- Lijnen tussen klassen tonen verbanden (associaties)

# Balloon.cs

```
public class Balloon
{
    private int x = 50;
    private int y = 50;
    private int diameter = 20;
    private Ellipse ellipse;

    ...

    public void MoveRight(int xStep)
    {
        x = x + xStep;
        UpdateEllipse();
    }

    public void ChangeSize(int change)
    {
        diameter = diameter + change;
        UpdateEllipse();
    }

    ...
}
```

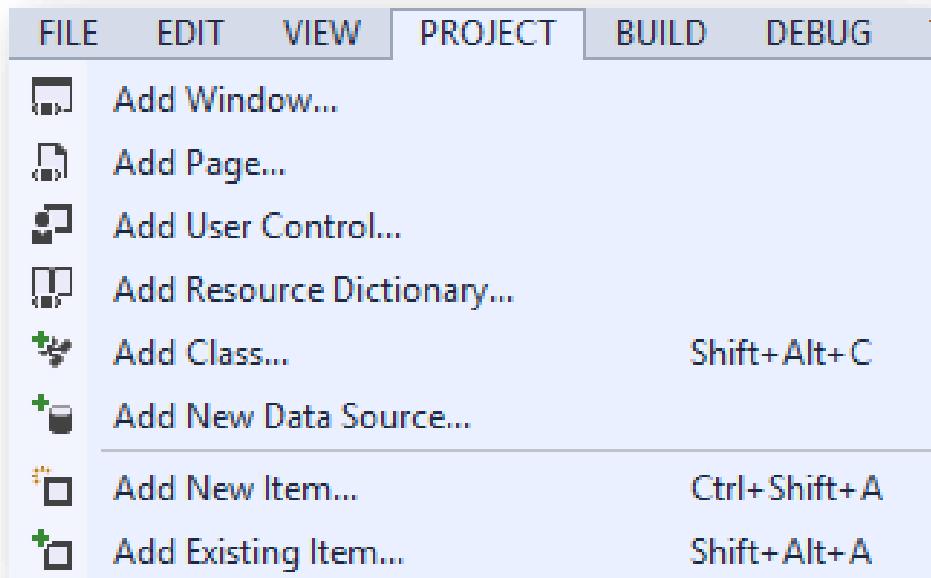
Klassenaam begint met hoofdletter

Lege lijnen + inspringen (leesbaarheid)

1 klasse per bestand (= goede gewoonte)

# Demo

- Voeg de klasse Balloon toe
- **Add Class** in het **Project** menu



# private variabelen

```
public class Balloon
{
    private int x = 50;
    private int y = 50;
    private int diameter = 20;
    private Ellipse ellipse;

    ...
    public void MoveRight(int xStep)
    {
        x = x + xStep;
        UpdateEllipse();
    }

    public void ChangeSize(int change)
    {
        diameter = diameter + change;
        UpdateEllipse();
    }
    ...
}
```

# private variabelen

- Beschrijven de toestand van elk Balloon object
- Zijn niet toegankelijk voor de buitenwereld
- Variabelen `public` maken is een slechte programmeerstijl
- Wel: `public` methodes en properties manipuleren de waarden van `private` variabelen.

# public methoden

```
public class Balloon
{
    ...
    public void MoveRight(int xStep)
    {
        x = x + xStep;
        UpdateEllipse();
    }

    public void ChangeSize(int change)
    {
        diameter = diameter + change;
        UpdateEllipse();
    }

    public void DisplayOn(Canvas drawingCanvas)
    {
        drawingCanvas.Children.Add(ellipse);
    }
    ...
}
```

# public methoden

- Beschrijven het gedrag van elk Balloon object
- Vormen de connectie, interface met de buitenwereld
- Zeer vaak manipuleren public methodes de interne toestand van een object op een gecontroleerde manier

# Inkapseling

- Informatie verbergen
- Engels: encapsulation
- *Zeer belangrijk OO principe!*
- Elk object heeft een toestand die verborgen blijft voor de buitenwereld
- Elk object vertoont een bepaald gedrag die deze toestand beïnvloedt
- De buitenwereld kan enkel het object manipuleren via `public`
  - Methodes
  - Properties

# Inkapseling

```
public class Balloon
{
    private int x = 50;
    private int y = 50;
    private int diameter = 20;
    private Ellipse ellipse;

    public void MoveRight(int xStep)
    {
        x = x + xStep;
        UpdateEllipse();
    }

    public void ChangeSize(int change)
    {
        diameter = diameter + change;
        UpdateEllipse();
    }

    public void DisplayOn(Canvas drawingCanvas)
    {
        drawingCanvas.Children.Add(ellipse);
    }
    ...
}
```

Standpunt ontwerper

# Inkapseling

```
public class Balloon  
{
```

Standpuntgebruiker

```
    public void MoveRight(int xStep)
```

```
    public void ChangeSize(int change)
```

```
    public void DisplayOn(Canvas drawingCanvas)
```

```
}
```

# Properties

- Zorgen voor een gecontroleerde toegang tot member variabelen
  - Lezen: *get access*

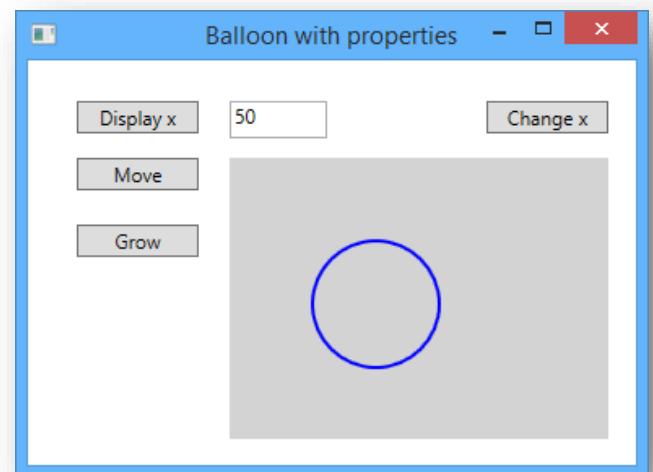
```
name = textBox1.Text;
```
  - Schrijven: *set access*

```
textBox1.Visibility = Visibility.Hidden;
```
- Dit *lijkt* alsof je rechtstreeks member variabelen verandert, maar je doet dit via (property-)methoden!

# Voorbeeld

```
private void changeXButton_Click(object sender, RoutedEventArgs e)
{
    balloon.XCoord = Convert.ToInt32(xCoordTextBox.Text);
}

private void displayXButton_Click(object sender, RoutedEventArgs e)
{
    xCoordTextBox.Text = Convert.ToString(balloon.XCoord);
}
```



# Property definitie

```
public int XCoord
{
    get
    {
        return x;
    }
    set
    {
        x = value;
        UpdateEllipse();
    }
}
```

```
public int XCoord
{
    get { return x; }
    set { x = value; UpdateEllipse(); }
```

Compacte indentatie

VS tip: propfull <tab><tab>

# Property definitie

```
public int XCoord  
{  
    get { return x; }  
    set { x = value; }  
}
```

Indien de body van een property uit slechts 1 regel bestaat, kan je een nog kortere schrijfwijze hanteren

```
public int XCoord  
{  
    get => x;  
    set => x = value;  
}
```

=> = lambda-operator

# Property met enkel get of set

```
public int XCoord  
{  
    get { return x; }  
}
```

Read-only property

Of korter met de  
lambda-operator

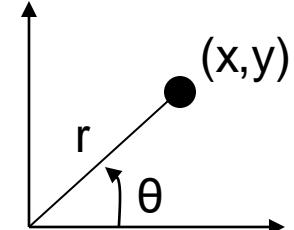
```
public int Xcoord => x;
```

```
public int XCoord  
{  
    set  
    {  
        x = value;  
        UpdateEllipse();  
    }  
}
```

Write-only property

# Waarom properties?

- De interne voorstelling wordt verborgen
  - Bv: stel dat je in plaats van met een  $(x,y)$  stelsel met een polair assenstelsel  $(r,\theta)$  zou werken, dan moeten de property methods de omrekening doen naar  $x$  en  $y$
- Onderscheid tussen get en set is mogelijk
- Toegangscontrole is mogelijk, bv enkel  $x$  en  $y$  tussen -100 en 100 is toegestaan



# Autoproperties

- Vaak wil je een member variabele enkel instellen en uitlezen.
- Nooit public member variabelen gebruiken!  
Altijd via properties werken.
- Autoproperties: handige schrijfwijze voor property met member variabele achter de schermen.
- VS: prop <tab><tab>

```
public Color FillColor { get; set; }
```

# Methode of property?

- Methode = actie, gedrag, vaardigheid, ...
  - Naam = werkwoord
- Property = toestand, informatie, eigenschap,
  - ...
  - Naam = zelfstandig naamwoord
- Soms is het onderscheid een kwestie van smaak en/of stijl
  - ChangeColor() in plaats van Color

# Constructor

- Speciale methode om een object te creëren.
- Deze methode heeft dezelfde naam als de klasse.
- Parameters zijn mogelijk.
- Aanroep met new.
- Als je geen constructor definieert, dan is er toch (impliciet) een default constructor zonder parameters. Deze initialiseert alle member variabelen op hun default waarde.
- Van zodra je zelf één (of meer) constructors schrijft, dan vervalt deze default constructor.
- VS: ctor <tab> <tab>

# Constructor: voorbeeld

```
public Balloon(int initialX,  
              int initialY,  
              int initialDiameter)  
{  
    x = initialX;  
    y = initialY;  
    diameter = initialDiameter;  
  
    CreateEllipse();  
    UpdateEllipse();  
}
```

```
Balloon balloon1 = new Balloon(10, 10, 50);
```

// wanneer compileert dit niet?

```
Balloon balloon2 = new Balloon();
```

# Overloading of default waarden

- Vaak maak je meerdere versies van een constructor
- Telkens met een variatie van de parameters
- = overloading
- Default waarden kunnen hier vaak voor kortere code zorgen

# Overloading constructor

Voor elke combinatie van parameters een overloaded constructor

```
public Balloon()
{
    x = 50;
    y = 50;
    diameter = 20;

    CreateEllipse();
    UpdateEllipse();
}
```

```
public Balloon(int initialX)
{
    x = initialX;
    y = 50;
    diameter = 20;

    CreateEllipse();
    UpdateEllipse();
}
```

# Overloading constructor

Voor elke combinatie van parameters een overloaded constructor

```
public Balloon(int initialX,  
               int initialY)  
{  
    x = initialX;  
    y = initialY;  
    diameter = 20;  
  
    CreateEllipse();  
    UpdateEllipse();  
}
```

```
public Balloon(int initialX,  
               int initialY,  
               int initialDiameter)  
{  
    x = initialX;  
    y = initialY;  
    diameter = initialDiameter;  
  
    CreateEllipse();  
    UpdateEllipse();  
}
```

# Default waarden

```
public Balloon(int initialX = 50,  
               int initialY = 50,  
               int initialDiameter = 20)  
{  
    x = initialX;  
    y = initialY;  
    diameter = initialDiameter;  
  
    CreateEllipse();  
    UpdateEllipse();  
}
```

Dit is equivalent met

```
public Balloon()  
{  
    x = 50;  
    y = 50;  
    diameter = 20;  
  
    CreateEllipse();  
    UpdateEllipse();  
}
```

# private methoden

- Interne hulpfuncties (binnen het object)
- Als je deze methodes binnen andere methodes van het object gebruikt, hoef je geen object ervoor te zetten. Eventueel mag het woord `this`

# private methoden

```
...  
public void DisplayOn(Canvas drawingCanvas)  
{  
    drawingCanvas.Children.Add(ellipse);  
}  
  
private void CreateEllipse()  
{  
    ellipse = new Ellipse()  
    {  
        Stroke = new SolidColorBrush(Colors.Blue),  
        StrokeThickness = 2  
    };  
}  
  
private void UpdateEllipse()  
{  
    ellipse.Margin = new Thickness(x, y, 0, 0);  
    ellipse.Width = diameter;  
    ellipse.Height = diameter;  
}
```

# private methoden

```
public double Area
{
    get
    {
        double area = CalculateArea();
        return area;
    }
}

private double CalculateArea()
{
    double radius;
    radius = diameter / 2.0;
    return Math.PI * radius * radius;
}
```

# Bewerkingen op objecten

- De bewerkingen op primitieve types liggen vast
- De bewerkingen op objecten worden bepaald door de methodes die op deze objecten bestaan

# Objecten vernietigen

```
Balloon balloon = new Balloon(10, 10, 50);  
  
// Dit maakt het eerste object niet meer toegankelijk  
// garbage collector zal dit dan verwijderen  
balloon = new Balloon(20, 10, 30);
```

- Garbage collection is een proces dat uitgevoerd wordt door de “Common Language Runtime” van .NET
- Het zorgt ervoor dat niet meer gebruikte objecten worden verwijderd, zodat het geheugen weer vrijkomt voor nieuwe objecten
- Je kan als programmeur een hint geven dat je een object niet meer nodig hebt door aan de variabele de waarde null toe te kennen

# static methoden en properties

- Soms heeft het logisch gezien geen nut om van een klasse objecten te maken
- Typische voorbeelden zijn bibliotheken van functies
- De methoden en properties spelen dan op klassenniveau i.p.v. op objectniveau
- Sleutelwoord: **static**

# static methoden en properties

```
public static class Math
{
    public static double Sqrt(double value)
    {
        ...
    }
}
```

```
...
double x = Math.Sqrt(64);
...
```

*Hoe zou je zelf de kleuren Black en White definiëren  
in de klasse Colors?*

# Overerving

Hoofdstuk 11



# In dit hoofdstuk ...

- Een nieuwe klasse maken op basis van een bestaande klasse
- Wanneer en hoe **protected** variabelen
- Wanneer en hoe methodes overschrijven
- UML
- **base** en **abstract**
- Opsommingen met **enum**

# Inleiding

- Hoe hergebruik stimuleren
  - Een bestaande klasse die “bijna” alles doet wat je wil kan je m.b.v. overerving uitbreiden
- *Overerving is niet de enige oplossing. Een andere (en vaak betere) methode is compositie (hoofdstuk 20)*

# Een basisklasse: Sphere

```
public class Sphere
{
    protected int xCoord = 100;
    protected int yCoord = 100;
    protected Ellipse ellipse;

    public int X
    {
        set { xCoord = value; Redraw(); }
    }

    public int Y
    {
        set { yCoord = value; Redraw(); }
    }

    ...
}
```

Beschrijft enkel stilstaande objecten  
Deze klasse dient als basis voor Bubble  
(afmeting veranderen) en de klasse  
Ball (horizontaal bewegen)

# Een basisklasse: Sphere

```
...
    public virtual void CreateEllipse(Canvas drawingCanvas)
    {
        ellipse = new Ellipse()
        {
            Stroke = new SolidColorBrush(Colors.Black),
            StrokeThickness = 2,
            Width = 40,
            Height = 40,
            Margin = new Thickness(xCoord, yCoord, 0, 0)
        };
        drawingCanvas.Children.Add(ellipse);
    }
...
}
```

# Een basisklasse: Sphere

```
...
    public virtual void Redraw()
    {
        if (ellipse != null)
        {
            ellipse.Margin = new Thickness(xCoord, yCoord, 0, 0);
        }
    }
}
```

# Een subklasse: Bubble

```
public class Bubble : Sphere
{
    protected int radius = 60;

    public int Size
    {
        set { radius = value; Redraw(); }
    }

    public override void Redraw()
    {
        if (ellipse != null)
        {
            ellipse.Margin = new Thickness(xCoord, yCoord, 0, 0);
            ellipse.Width = 2 * radius;
            ellipse.Height = 2 * radius;
        }
    }
}
```

superklasse

Overschrijft het gedrag van de basisklasse Sphere

# Een subklasse: Bubble

```
public void MoveVertical(int amount)
{
    yCoord += amount;
    Redraw();
}
```

# Betekenis van overerven

- *“Dit betekent dat deze klasse alle items van de klasse **Sphere** overerft, behalve de constructoren. Bovendien heeft deze klasse toegang tot alle items die niet als **private** binnen de klasse **Sphere** staan aangegeven”*
- Ook de **private** items worden dus overgeërfd, ze zijn enkel niet toegankelijk

# protected

- Doel: instantievariabelen beschikbaar stellen voor subklassen
- Dus: de klasse voorbereiden op hergebruik
- Niveaus
  - public: overal toegankelijk
  - protected: enkel toegankelijk vanuit een subklasse en vanuit de klasse zelf
  - private: enkel toegankelijk vanuit de klasse zelf
  - Lokale variabelen: bestaan enkel binnen de body van een methode
- Deze sleutelwoorden zijn eveneens van toepassing op methoden en properties
- Dilemma: wanneer maak ik een variabele **private** en wanneer **protected**?  
→ Hergebruik = moeilijk voorspelbaar

# Aanvullende items

- De subklasse kan vanzelfsprekend aangevuld worden met instantievariabelen, methodes en properties, net zoals een “gewone” klasse
- Voorbeelden
  - **radius**
  - **Size** property

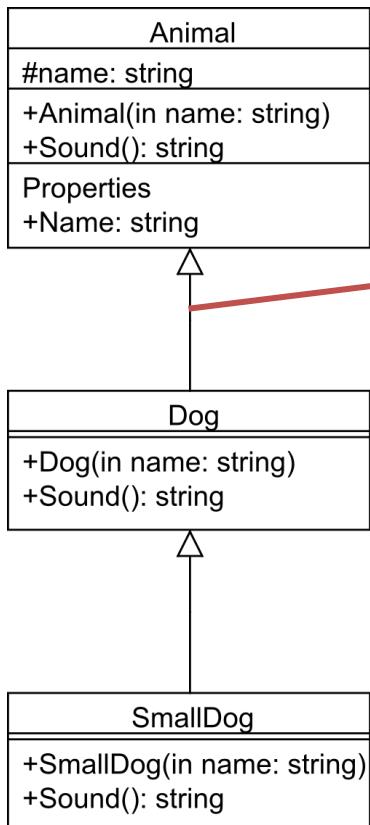
# Overschrijven

- Methoden/Properties uit de basisklasse van een nieuwe implementatie voorzien
- Sleutelwoorden
  - **virtual**
    - Deze methode/property van de basisklasse kan mogelijk overschreven worden
  - **override**
    - Deze methode/property van de subklasse overschrijft een methode/property van de basisklasse
    - Deze methode/property kan mogelijk opnieuw overschreven worden door een verdere subklasse van deze (sub)klasse

# Overschrijven

- *Experiment: tracht eens een methode te overschrijven waarbij:*
  - de superklasse virtual is maar geen override in de subklasse
  - De superklasse niet virtual is, maar wel override in de subklasse
- *Beschouw volgende klassen. Schrijf een implementatie uit op papier alsook een testprogramma*

# UML Diagram



Sound()  
→ "huh?"

Sound()  
→ "Woef woef"

Sound()  
→ "Kef kef"

Overerving wordt aangeduid door een **gesloten, driehoekige pijl** van de **subklasse naar de superklasse**

Overerving noemt men ook wel eens de **IS-EEN** relatie:  
*"Een SmallDog is een Dog"*  
*"Een Dog is een Animal"*  
*"Een SmallDog is ook een Animal"*  
*"Niet alle Dogs zijn SmallDogs!"*

# Testprogramma

*Wat is de uitvoer?*

```
...
Animal animal1, animal2, animal3;
animal1 = new Animal("Woefie");
animal2 = new Dog("Pluto");
animal3 = new SmallDog("Fifike");

PrintAnimal(animal1);
PrintAnimal(animal2);
PrintAnimal(animal3);
...

private void PrintAnimal(Animal animal)
{
    string line = $"animal ({animal.Name}) sounds as: {animal.Sound()}";
    MessageBox.Show(line);
}
```

# Overerving in de praktijk

- Een subklasse erft steeds alles over van al zijn superklassen (behalve constructors; zie later)
- Enkel de `protected` en `public` members zijn beschikbaar, de `private` members blijven achter de schermen aanwezig
- Een klasse kan rechtstreeks slechts van één andere klasse overerven (= Single Inheritance)
  - Java, C#, VB.NET: Single Inheritance
  - C++: Multiple Inheritance
- De ultieme superklasse waar elke andere klasse van overerft: `System.Object`

# System.Object

- De properties en methodes van deze klasse zijn universeel (voor elke klasse) nuttig en noodzakelijk.
- *Waarom zou je **ToString()** willen overschrijven?*

# base

- Referentie naar het (onmiddellijke) super-gedeelte van dit object
- Wordt gebruikt om methode / property / variabele aanroepen te doen van de superklasse
- Veel gebruikt: de overschreven methode doet hetzelfde als de methode van de superklasse + net iets meer

# Voorbeeld

```
public class Square
{
    protected int _side;

    public Square(int side)
    {
        _side = side;
    }

    public virtual int Area()
    {
        return _side * _side;
    }
}
```

```
public class Cube
    : Square
{
    public Cube(int side)
        : base(side)
    { }

    public override int Area()
    {
        return base.Area() * 6;
    }
}
```

# Constructor

- Heeft dezelfde naam als de klasse, aanroep met new
- Als geen constructor is voorzien
  - → impliciet een default constructor zonder params
- Als je wel een constructor voorziet
  - → moet je expliciet een parameterloze constructor declareren indien nodig
- Constructors worden niet overgeërfd!
- Een constructor van een subklasse roept impliciet de default constructor van de superklasse op, tenzij je expliciet een andere constructor van de parent klasse aanroept.

# Balloon voorbeeld

```
public class Balloon
{
    protected int xCoord, yCoord, radius;

    public Balloon()
    {
        xCoord = 10;
        yCoord = 10;
        radius = 20;
    }

    public Balloon(int initialX,
                  int initialY,
                  int initialRadius)
    {
        xCoord = initialX;
        yCoord = initialY;
        radius = initialRadius;
    }

    // remainder of the class
}
```

# Balloon voorbeeld

```
public class DifferentBalloon : Balloon
{
    public DifferentBalloon(int initialX,
                           int initialY)
        : base()
    {
        xCoord = initialX;
        yCoord = initialY;
        radius = 20;
    }

    // remainder of the class
}
```

Deze oproep gebeurt altijd als eerste, tenzij je expliciet een andere constructor van de parent oproept. Dit schrijf je buiten de method body.

Wat zou er gebeuren als in Balloon de constructor zonder parameters wordt weggelaten?

# Balloon voorbeeld

```
public class ModifiedBalloon : Balloon
{
    public ModifiedBalloon(int initialX,
                          int initialY,
                          int radius)
        : base(initialX, initialY, radius)
    {
    }

    // remainder of the class
}
```

Expliciete oproep van de parent constructor.

Merk op: deze constructor dient gedeclareerd, want constructors worden niet overgeërfd!

# Abstracte klassen

- Zeer dikwijls zijn base klassen van een inheritance tree niet instantieerbaar:
  - Deze klassen voorzien enkel gemeenschappelijke kenmerken voor subklassen, zowel properties als methoden
  - Deze klassen voorzien methoden/properties zonder implementatie, die wel in subklassen zinvol kunnen worden ingevuld

# Voorbeeld

```
public abstract class Shape
{
    protected int xCoord, yCoord;
    protected int size;

    public void MoveRight()
    {
        xCoord += 10;
        Redraw();
    }

    public abstract void CreatePhysicalShape(Canvas drawingCanvas);

    public abstract void Redraw();
}
```

Van deze klasse kan je **geen** objecten maken.  
Logisch, want een shape heeft fysisch nog geen betekenis.

Van deze methode bestaat (nog) geen implementatie.  
Deze moet ingevuld worden in de subklassen.  
Logisch, want een Shape is pas toonbaar als we al zijn eigenschappen bepaald hebben.

# Subklasse Circle

```
public class Circle : Shape
{
    private Ellipse ellipse;

    public Circle()
    {
        xCoord = 40;
        yCoord = 40;
        size = 80;
    }
}
```

# Subklasse Circle

```
public override void CreatePhysicalShape(Canvas drawingCanvas)
{
    ellipse = new Ellipse()
    {
        Stroke = new SolidColorBrush(Colors.Black),
        StrokeThickness = 2,
        Width = size,
        Height = size,
        Margin = new Thickness(xCoord, yCoord, 0, 0)
    };

    drawingCanvas.Children.Add(ellipse);
}

public override void Redraw()
{
    if (ellipse != null)
    {
        ellipse.Margin = new Thickness(xCoord, yCoord, 0, 0);
        ellipse.Height = size;
        ellipse.Width = size;
    }
}
```

Een Circle is een Shape met een concrete vorm.

Nu kan de CreatePhysicalShape methode zinvol geïmplementeerd worden.

# Oefening

- Animal, SmallDog, Dog
  - Welke klasse / methode is mogelijk abstract?
  - Pas de code aan

# UML notatie

- Abstracte klassen en methoden worden schuin gedrukt weergegeven

<i>Animal</i>
#name: string
+Animal(in name: string)
+Sound(): string
Properties
+Name: string

# sealed

- In sommige gevallen kan je als programmeur beslissen dat van een klasse niet meer kan overgeërfd worden
  - Veiligheidsoverweging
    - Bv. een klasse die een methode heeft om paswoorden te valideren
- Let hiermee op, want je beperkt dus hergebruik van je klasse!
- Soms zijn toch zinvolle voorbeelden te vinden:
  - De klasse `System.String` is sealed, een stringbewerking ligt voor altijd vast.

# Voorbeeld

```
public sealed class SmallDog : Dog  
{  
    ...  
}
```

# partial

- Om de code van de gebruikersinterface (MainWindow.xaml) netjes te kunnen scheiden van de code van de programmeur (MainWindow.xaml.cs)
- Voor eigen klassen niet aanbevolen

# partial

MainWindow.xaml.cs

```
public partial class MainWindow : Window
{
    public MainWindow()
    {
        InitializeComponent();
    }
}
```

# Enums

- Een type dat een opsomming (enumeratie) voorstelt
- Voorbeelden:
  - Alle dagen van de week: maandag, dinsdag, ...
  - De maanden van het jaar: januari, februari, ...
  - De noten van een toonladder: do, re, mi, fa, sol, ...
- Standaard heeft de eerste waarde uit de enumeratie de int-waarde 0, tenzij je dit aanpast.

# Voorbeeld

```
namespace PlayingCards
{
    public enum SuitType { Clubs, Spades, Diamonds, Hearts }

    public class Card
    {
        public int Value { get; set; }
        public SuitType Suit { get; set; }
    }
}
```

In plaats van te werken met strings (“Clubs”, “Spades”, ...), gebruik je een nieuw type (SuitType) die enkel de juiste waarden toelaat. Achter de schermen wordt voor Clubs de waarde 0 gebruikt, voor Spades de waarde 1, enz.

# Voorbeeld

```
Card twoOfHearts = new Card  
{  
    Value = 2,  
    Suit = SuitType.Hearts  
};
```

Je gebruikt de namen van de enumeraties.  
Dit verhoogt de leesbaarheid van je code.

```
public enum SuitType  
{  
    Clubs = 10, Spades = 20, Diamonds = 30, Hearts = 40  
}
```

Je kan de standaard nummering veranderen.

# Delegates en Events



*Niet kennen: blz. 289 - 296*

# Berekeningen

Hoofdstuk 12

# In dit hoofdstuk ...

- Hoe getallen weergeven
- Hoe wiskundige functies gebruiken
- Berekeningen
  - Lonen
  - Boekhouding
  - Voorspellingen (bv. weermodellen)
  - Grafisch (bewegen van componenten)

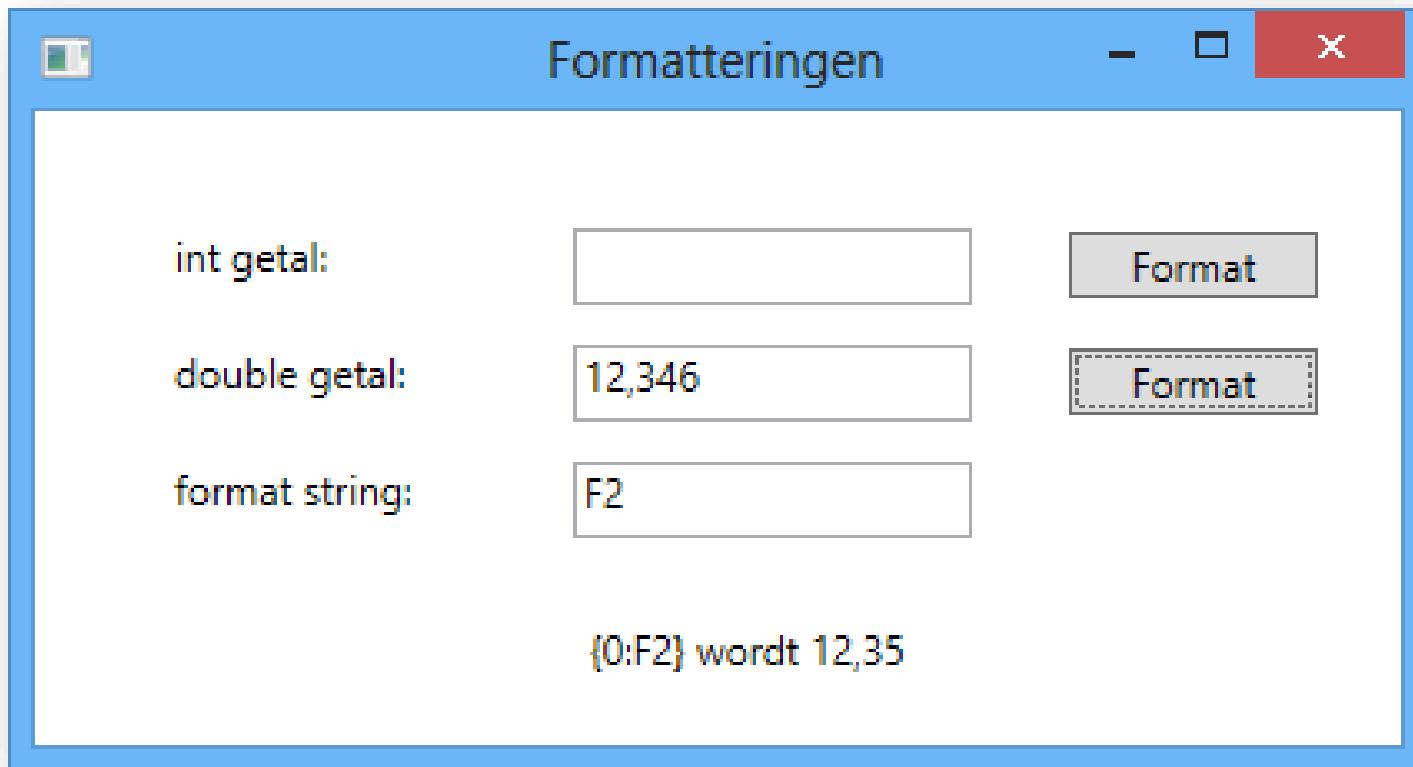
# Literals

- “letterlijke waarde”
- = getallen die je rechtstreeks in een programma schrijft
- Voorbeelden:
  - 10000 (→ int)
  - 12.34 (→ double)
  - 0.0001 of 1.0E-4
  - 12300000.0 of 1.23E7
- Duizentallen mag je groeperen met underscore \_
  - 12\_300\_000
  - 0.000\_1

# Formatteren van getallen

- `Convert.ToString(...)`  
zet een getal om naar een string
- `String.Format(...)`  
zet getallen om naar een string met opmaak
- Parameters  
`String.Format placeholderstring, n1, n2, ...)`
  - placeholderstring: string met placeholders en formattering  
bv {0:F2}
  - n1, n2, ... : de te formatteren variabelen

# Demo

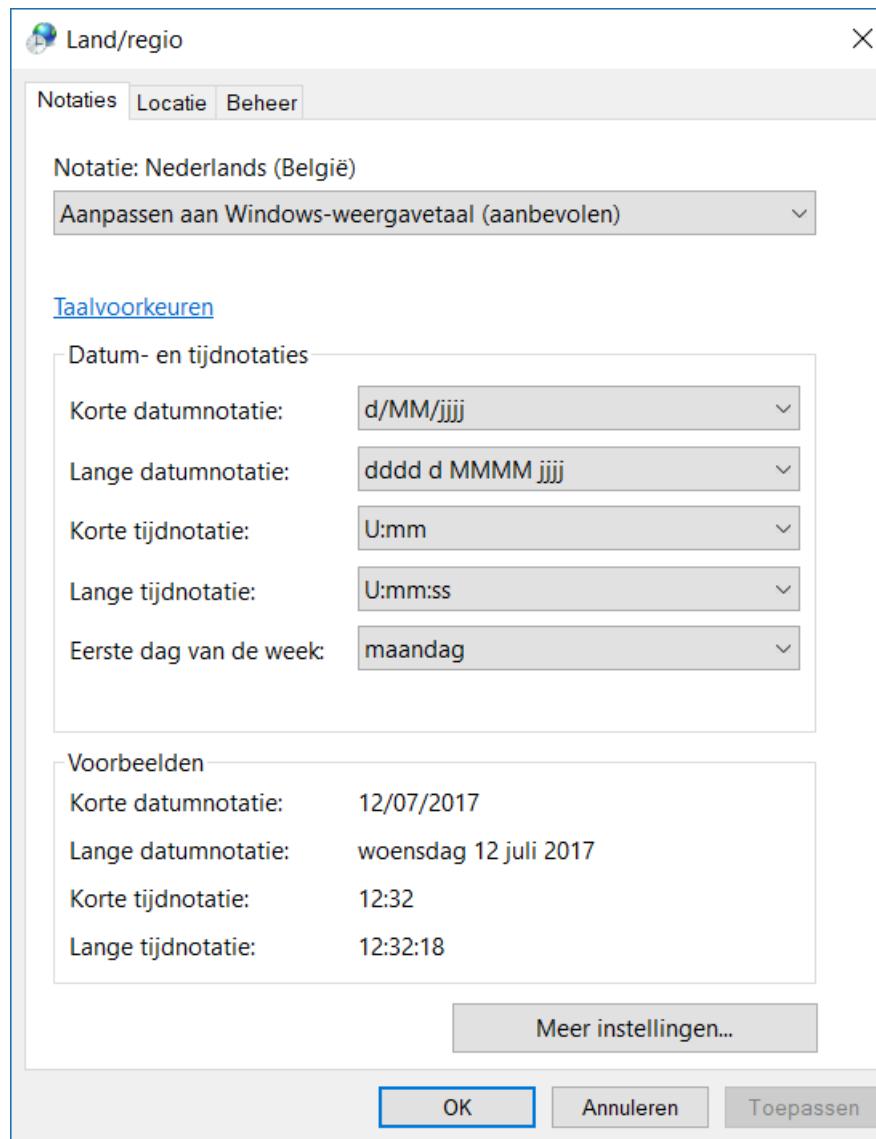


Data type	Voorbeeld	Resultaat
Decimaal	int number = 123; label1.Content = \$"{{number:D}}";	123
Decimaal	int number = 123; label1.Content = \$"{{number:D5}}";	00123
Floating-point	double number = 12.34; label1.Content = \$"{{number:F2}}";	12,34
Floating-point	double number = 12.34; label1.Content = \$"{{number:F0}}";	12
Wetenschappelijk	double number = 12000000; label1.Content = \$"{{number:E2}}";	1,20E007
Geldbedrag	double money = 12.34; label1.Content = \$"{{money:C}}";	€ 12,34
Getal	double number = 1234000; label1.Content = \$"{{number:N2}}";	1.234.000,00

# Landinstellingen

- Getallen noteren in C# code volgens de Amerikaanse conventie
  - Komma (,) om duizendtallen te scheiden
  - Punt (.) om gedeelte na de “komma” aan te geven
- Het resultaat van `String.Format` is afhankelijk van de landinstellingen
- Nederland/Belgische landinstellingen gebruiken , en . omgekeerd
- Ook het standaard valutasymbool is afhankelijk van landinstellingen

# Landinstellingen



# Wiskundige bibliotheek

- Klasse Math
- Math.Sqrt(...)
- Welk type van methoden zijn dit op de klasse Math?
- Lees de [online documentatie!](#)
- Ook constanten PI en E zijn beschikbaar

# Constanten

- Onveranderlijke waarden
- Nooit rechstreeks de waarde schrijven, maar werk met een constante variabele
- Leesbaarheid verhogen
- Conventie: naam start met hoofdletter

```
const double InchesToCm = 2.54;
```

# Casestudy: geld

Interest Calculation

Enter initial amount:

Enter interest rate:

After 1 years the money has become 110 euros and 0 eurocents.

After 2 years the money has become 121 euros and 0 eurocents.

After 3 years the money has become 133 euros and 10 eurocents.

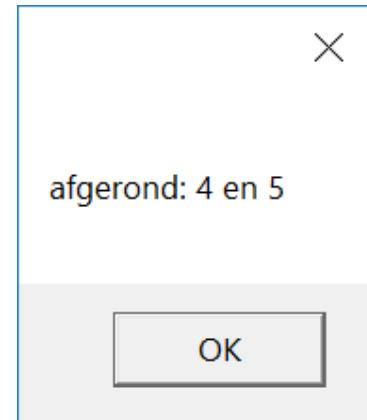
# Casestudy: geld

- `Convert.ToInt32(value)`
  - Omzetten naar int door *afronding*!
- `int x = (int)value; // cast`
  - Omzetten naar int door *afkappen*!
- `Math.Round(value)`
  - Afronden naar dichtstbijzijnde geheel getal
  - Idem `Convert.ToInt32`
- `Convert.ToDouble(value)`
  - Omzetten naar double
- Zie online help: [Convert](#), [Math](#)

# Over afronden in .NET

- `Math.Round()` en `Convert.ToInt32()`
  - Werkt voor de meeste gevallen zoals je verwacht:

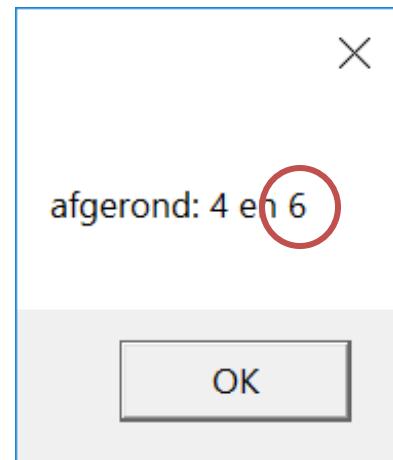
```
double d1 = 4.2;
double d2 = 4.8;
String line = $"afgerond: {Math.Round(d1)} en {Convert.ToInt32(d2)}";
MessageBox.Show(line);
```



# Over afronden in .NET

- `Math.Round()` en `Convert.ToInt32()`
  - Echter soms heel onverwachte resultaten bij afronden op halve waarden

```
double d1 = 4.5;
double d2 = 5.5;
String line = $"afgerond: {Math.Round(d1)} en {Convert.ToInt32(d2)}";
MessageBox.Show(line);
```



# Over afronden in .NET

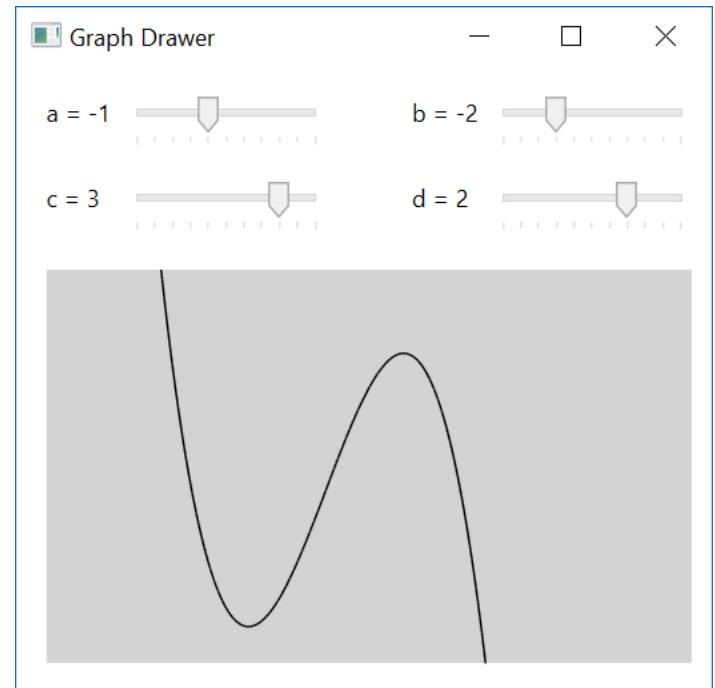
- Verklaring
  - Er wordt hier gebruik gemaakt van een afrondingsmechanisme dat bekend staat als het bankiersalgoritme (“banker's rounding”)
  - Regel: een halve waarde wordt afgerond naar het dichtstbijzijnde EVEN getal
    - $4.5 \rightarrow 4$  (en niet 5 zoals je zou verwachten)
- Meer [info](#)

# Casestudy: iteratie

- `Math.Abs(value)`
  - Absolute waarde
- *Bestudeer de broncode, duid aan waar er gecontroleerd wordt of de vereiste nauwkeurigheid bereikt is.*

# Grafieken

- Bestudeer de broncode
- De kromme wordt gesimuleerd door voldoende rechte lijnen
- Schalen → kromme binnen het Canvas



# Exceptions

- Abnormale condities te wijten aan
  - Programmeerfouten
  - Runtime fouten
- In dit hoofdstuk
  - Te grote getallen → Overflow (PositiveInfinity)
  - Deling door nul
- In hoofdstuk 17 zien we hoe hiermee om te gaan

# Gegevensstructuren: lijsten en ListBox

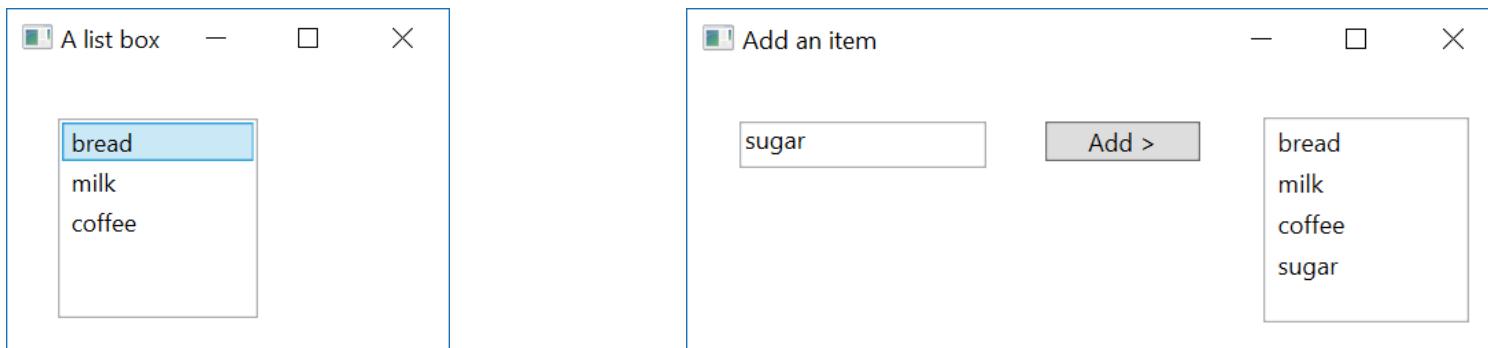
Hoofdstuk 13

# In dit hoofdstuk ...

- Nieuwe control: **ListBox**
  - Items toevoegen en verwijderen
  - Lengte
  - Selectie, zoeken
- Nieuwe datastructuur: **(I)List**
  - Objecten toevoegen en verwijderen
  - Lengte
  - Selectie, zoeken
- **ListBox** is grafisch, **(I)List** werkt achter de schermen

# Een ListBox

- Beschikbaar in de toolbox → er wordt een XAML-element aangemaakt: <ListBox>
- Toont een lijst
- Deze lijst kan tijdens het ontwerpen aan het ListBox-element toegevoegd worden (<ListBoxItem>-element)



# IList

- De datastructuur die achter de schermen de items van een `ListBox` beheert
- Kan ook onafhankelijk van een `ListBox` worden gebruikt
- Opmerking: dit is eigenlijk geen klasse, maar een interface → hoofdstuk 22
- De werkelijke klasse die achter de schermen speelt is: [ItemCollection](#)

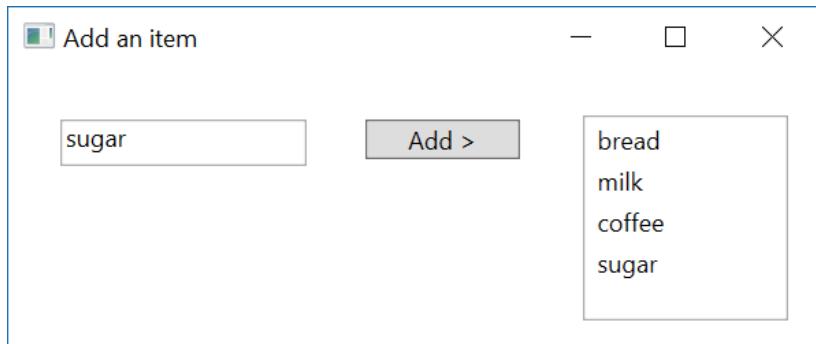
# IList

```
IList list = shoppingListBox.Items;  
int numberOfItems = list.Count;  
  
// korter:  
int numberOfItems = shoppingListBox.Items.Count;
```

*In welke namespace vind je **IList**? Waarom  
is er een **using** statement nodig in de code  
van **MainWindow**?*

# Items toevoegen

```
private void addButton_Click(object sender,  
                           RoutedEventArgs e)  
{  
    ListBoxItem newItem = new ListBoxItem();  
    newItem.Content = itemTextBox.Text;  
    shoppingListBox.Items.Add(newItem);  
}
```



Een toevoeging aan de  
IList veroorzaakt  
automatisch een update van  
de bijhorende ListBox

# Items toevoegen

```
private void addButton_Click(object sender,  
                           RoutedEventArgs e)  
{  
    ListBoxItem newItem = new ListBoxItem();  
    newItem.Content = itemTextBox.Text;  
    shoppingListBox.Items.Add(newItem);  
}
```

Zou je de waarde van *itemTextBox.Text* ook als *string* kunnen toevoegen aan *shoppingListBox*?

```
string newItem = itemTextBox.Text;  
shoppingListBox.Items.Add(newItem);
```

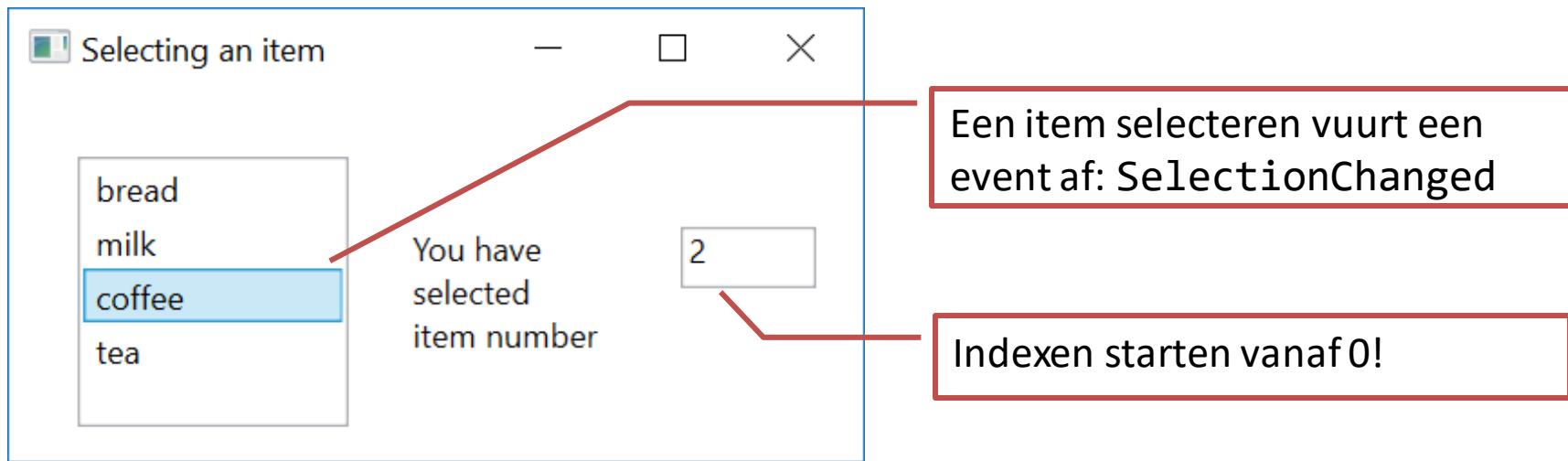
Zo ja, is dit aangeraden?

# De lengte van een lijst

```
private void countButton_Click(object sender,  
                           RoutedEventArgs e)  
{  
    MessageBox.Show($"{shoppingListBox.Items.Count}");  
}
```

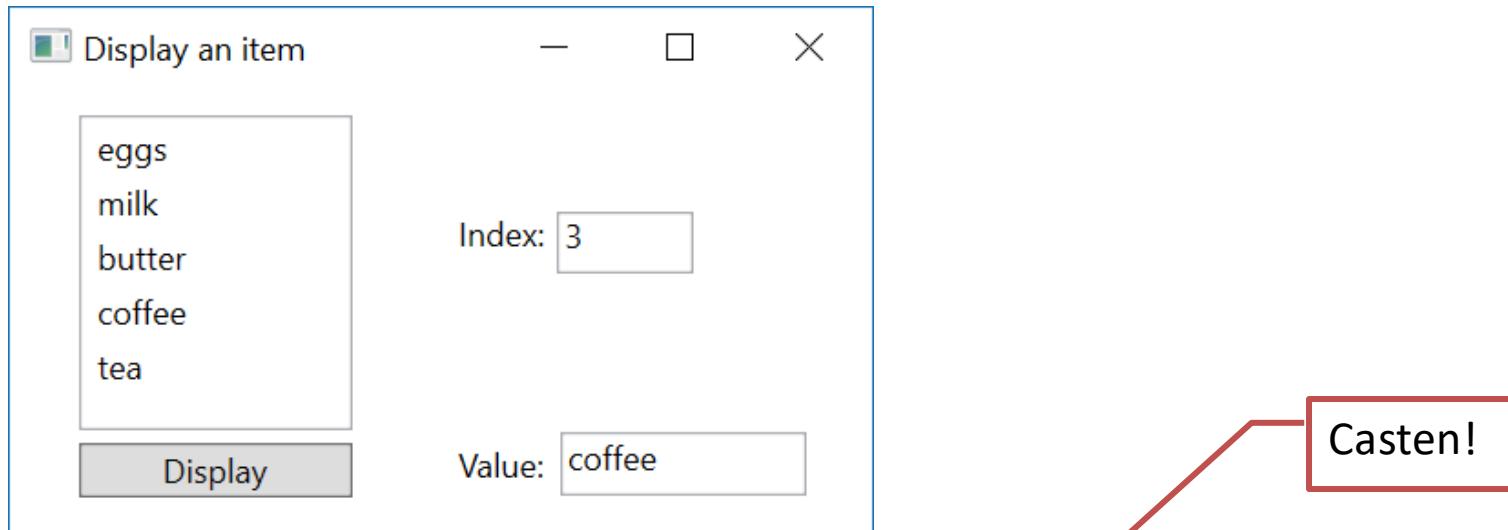
Je vraagt het aantal  
elementen op dat in de  
bijhorende **IList** zit

# Indices (selecting an item)



```
private void shoppingListBox_SelectionChanged(object sender,  
                                            SelectionChangedEventArgs e)  
{  
    numberTextBox.Text = $"{shoppingListBox.SelectedIndex}";  
}
```

# Indices (display an item)



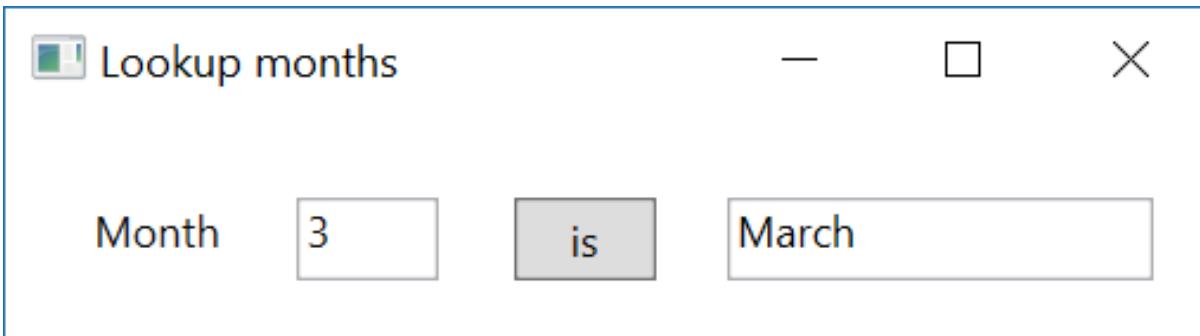
```
private void displayButton_Click(object sender, RoutedEventArgs e)
{
    int index = Convert.ToInt32(indexTextBox.Text);
    ListBoxItem item = (ListBoxItem)shoppingListBox.Items[index];
    valueTextBox.Text = Convert.ToString(item.Content);
}
```

# Verwijderen en invoegen

```
shoppingListBox.Items.RemoveAt(3);  
shoppingListBox.Items.Clear();  
shoppingListBox.Items.Insert(5, newItem);
```

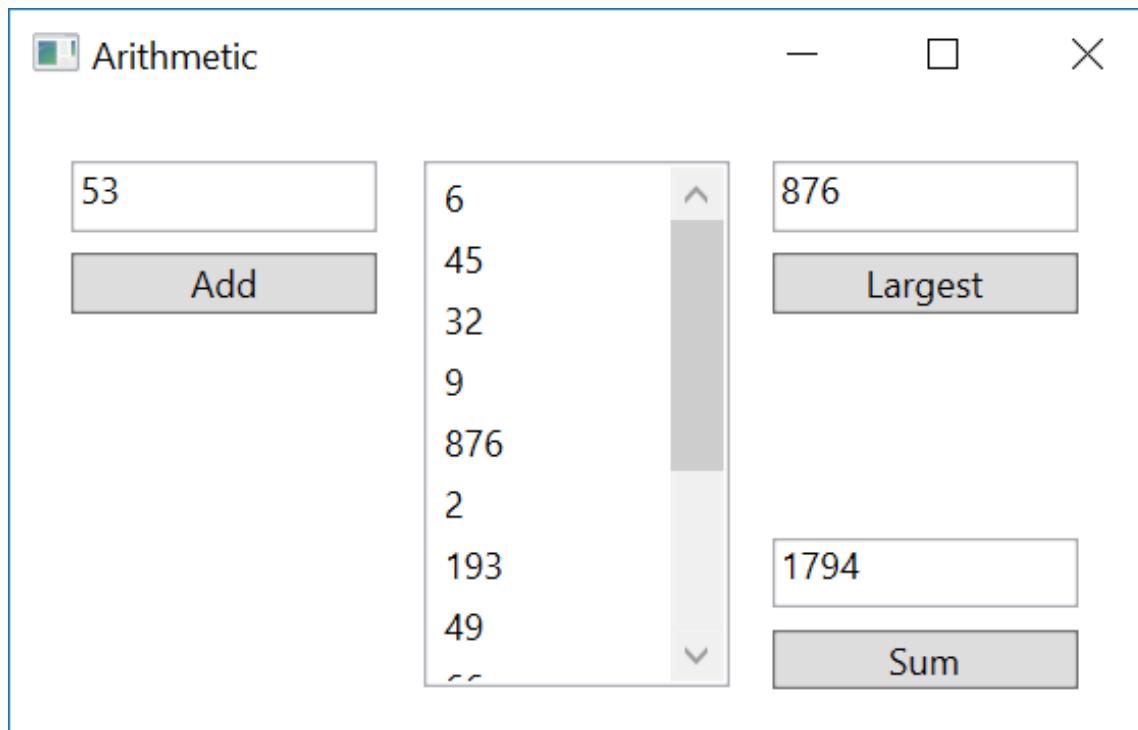
- *Welk datatype mag newItem zijn?*
- *Welk datatype is newItem best?*
- *Waarom?*

# Opzoektabel



```
private void lookupButton_Click(object sender,
                               RoutedEventArgs e)
{
    int monthNumber =
        Convert.ToInt32(monthNumberTextBox.Text);
    ListBoxItem item =
        (ListBoxItem)monthListBox.Items[monthNumber - 1];
    string monthName = Convert.ToString(item.Content);
    monthNameTextBox.Text = monthName;
}
```

# Rekenen met een ListBox



*Bestudeer zelf de code*

# foreach

Dit kan koper door een speciale  
foreach constructie

```
int sum = 0;  
int number;  
ListBoxItem item;  
  
for (int index = 0; index < numberListBox.Items.Count; index++)  
{  
    item = (ListBoxItem)numberListBox.Items[index];  
    number = Convert.ToInt32(item.Content);  
    sum += number;  
}  
sumTextBox.Text = Convert.ToString(sum);
```

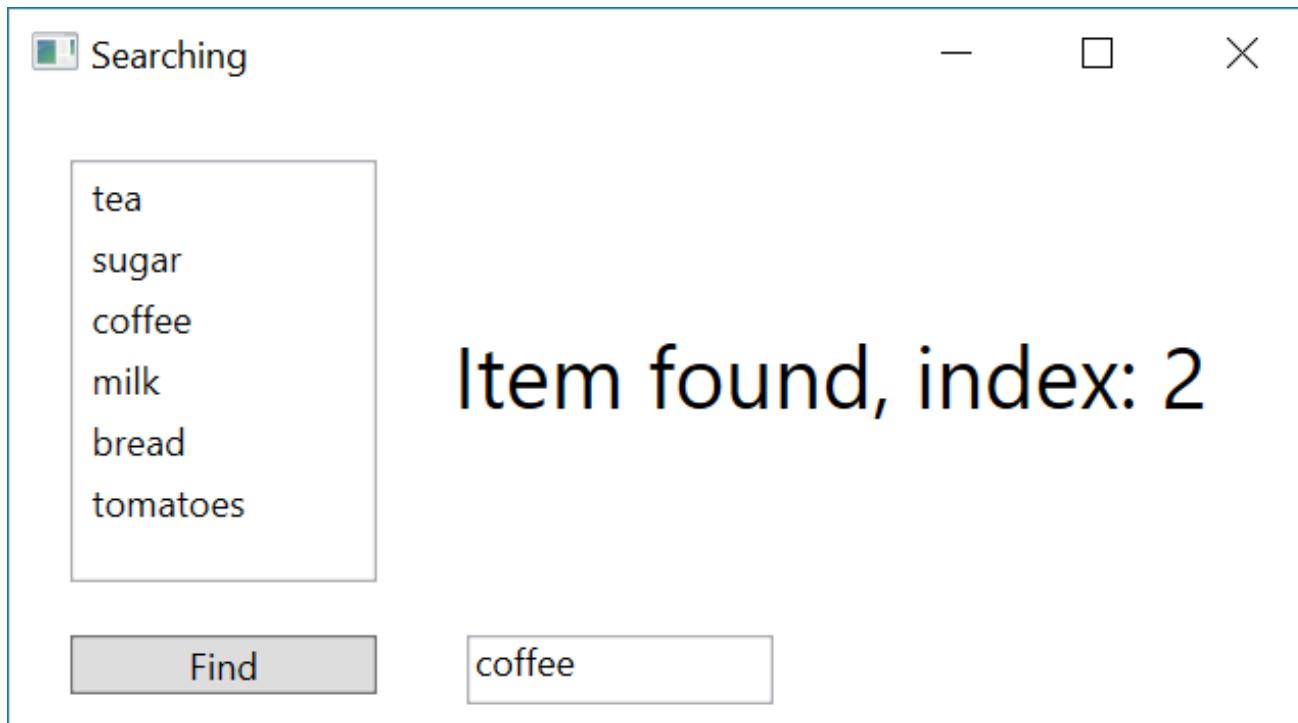
# foreach

```
int sum = 0;  
  
foreach (ListBoxItem item in numberListBox.Items)  
{  
    sum += Convert.ToInt32(item.Content);  
}  
sumTextBox.Text = Convert.ToString(sum);
```

# foreach

- Erg kort
- Maar: je beschikt niet over de indexwaarde van de lijst
  - Niet zo flexibel!
- Alleen te gebruiken als je alle waarden van een lijst wil doorlopen

# Zoeken



*Bestudeer zelf de code*

# Gebruik van een lijst - Genericiteit

- De `List` klasse kan ook op zichzelf gebruikt worden
- Algemene datastructuur
- Preciezer: `List` is een klasse die de `IList` interface implementeert
- Generisch: je kan aanduiden van welke klasse de objecten zijn die in de `List` zitten

# List: voorbeeld

```
private IList<string> list; // member variable  
...  
list = new List<string>(); // in mainwindow constructor
```

```
private IList<string> list = new List<string>();
```

list kan enkel string objecten bevatten.  
Dit zal door de compiler gecontroleerd worden.

# List: voorbeeld

```
using System.Collections.Generic;
```

Benodigde namespace → standaard aanwezig

```
list.Add("bread");  
list.Add("milk");  
list.Add("coffee");
```

Toevoegen van items

# list: voorbeeld

```
for (int index = 0; index < list.Count; index++)  
{  
    ListBoxItem item = new ListBoxItem();  
    item.Content = list[index];  
    shoppingListBox.Items.Add(item);  
}
```

```
foreach (string itemText in list)  
{  
    ListBoxItem item = new ListBoxItem();  
    item.Content = itemText;  
    shoppingListBox.Items.Add(item);  
}
```

een lijst doorlopen

# Opmerking

- In het vorige voorbeeld werden de strings gekopieerd van een List naar de lijst van een ListBox
- Dit is overbodig werk, want je kan gebruik maken van “Data Binding”

```
shoppingListBox.ItemsSource = list;
```

# Opmerking

- Een volledige behandeling van data binding valt echter buiten het bestek van de cursus
- Meer info [online](#)

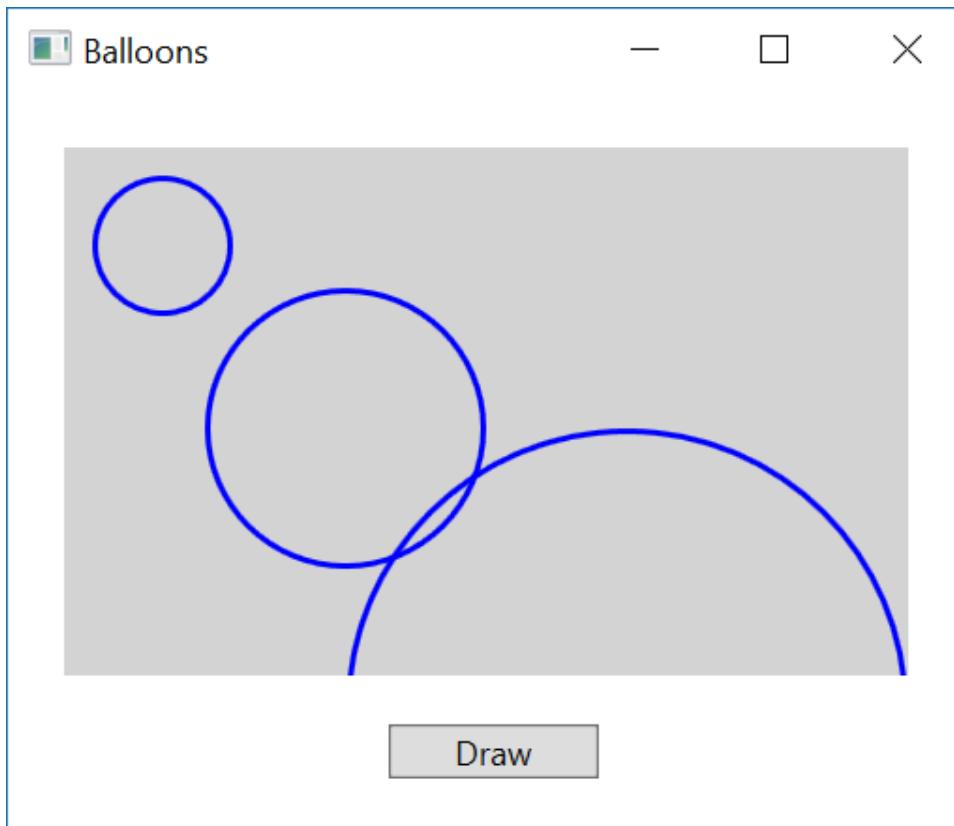
*Voeg eens een nieuw element toe aan list →  
ListBox wordt niet automatisch bijgewerkt.*

*Wijzig de List-klasse eens in  
ObservableCollection-klasse →  
ListBox wordt automatisch bijgewerkt*

# Methodes en Properties van List

*Bestudeer dit zelf in de [online Help](#)*

# Lijsten van objecten



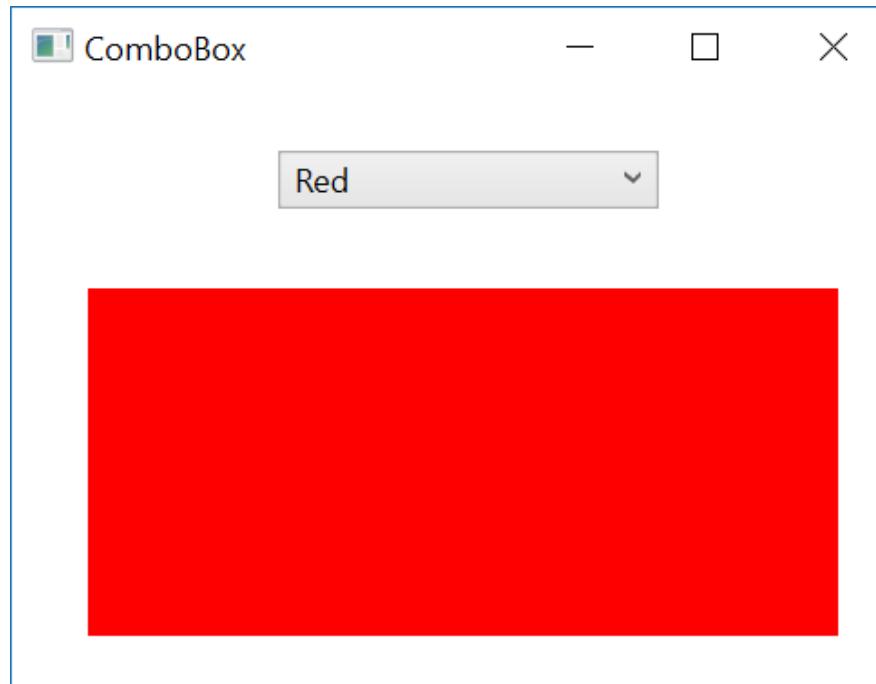
- *Bestudeer de code: welke klasse van objecten wordt in een List bewaard?*
- *Welk List methodes worden gebruikt?*

# ComboBox

- Voeg een *ComboBox* toe op een form
- Welk event wordt er afgevuurd als een item wordt geselecteerd?
- Hoe kan je het geselecteerde item opvragen?
- Welke klasse hebben deze items?
- Naar welke klasse cast je het best?

# Oefening

*Een ComboBox bevat verschillende kleuren. Een label wordt naargelang de gekozen kleur ingevuld*



# Zoeken in lijsten met delegates en lambda's



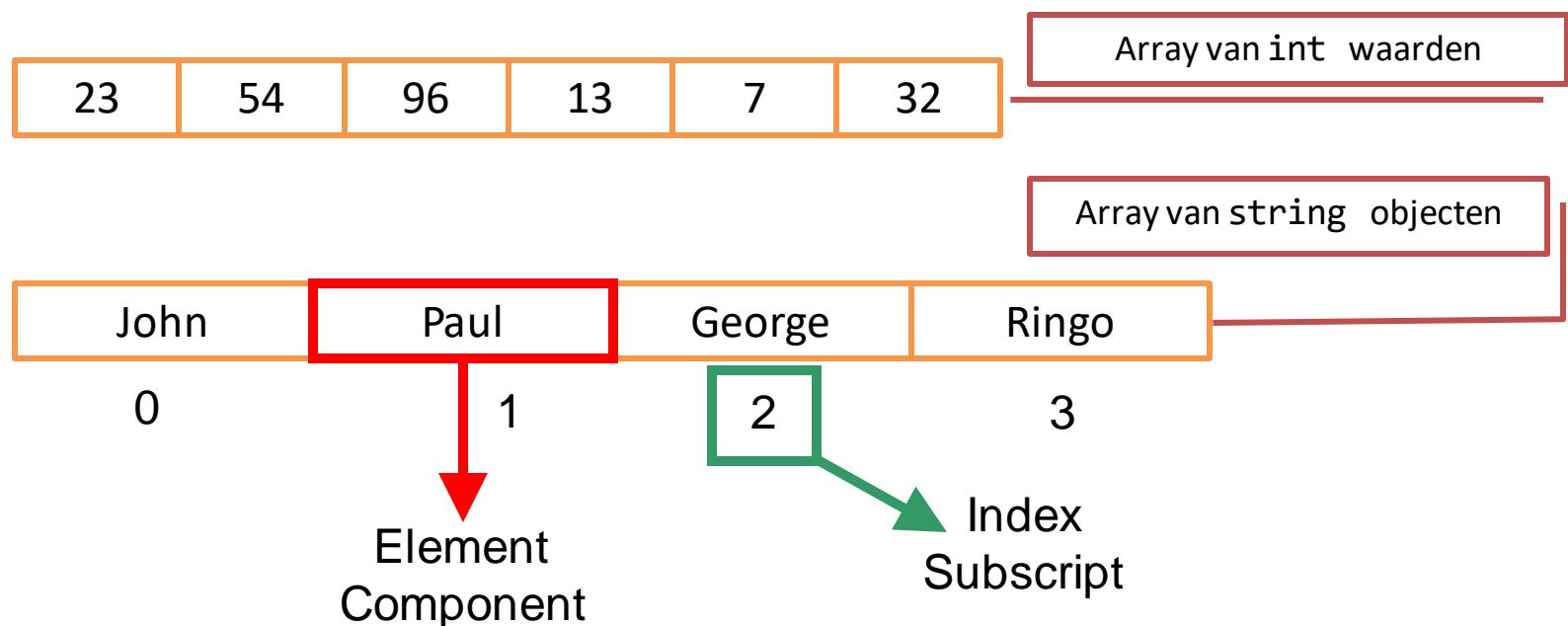
*Niet kennen: blz. 343 - 346*

# Arrays

Hoofdstuk 14

# Inleiding

Een array is een object, dat andere objecten groepeert



# Een array creëren

```
int[] ages = new int[6];
string[] band = new string[4];
```

- Deze statements declareren twee variabelen: `ages` en `band`
- Automatisch worden twee array objecten aangemaakt die 6 resp. 4 elementen kunnen bevatten

# Indices

- Starten van 0
- Zijn gehele waarden (int)
- Let op dat je index niet buiten het bereik van de array valt!
  - Welk soort fout krijg je dan?

```
ages[2] = Convert.ToInt32(ageTextBox.Text);
band[3] = bandMemberTextBox.Text;

ageTextBox.Text = $"the first age is {ages[0]}";
bandMemberTextBox.Text = $"the 4th band member is {band[3]}";

// Fout statement
ages[6] = 45;
```

# Sommeren van de elementen

```
int sum = 0;
for (int i = 0; i < ages.Length; i++)
{
    sum += ages[i];
}
```

*Volgende code is verkeerd, waarom?*

```
int sum = 0;
for (int i = 0; i <= ages.Length; i++)
{
    sum += ages[i];
}
```

# De lengte van een array

- Property Length
  - Maximum aantal elementen in de array
- Lengte van een array is vast
  - Kan wel: met new een nieuw array object toekennen aan de array variabele
  - Het vorige array object wordt opgeruimd door de garbage collector

# Arrays doorgeven als parameters

```
private void PrintArrays(string name, string[] arrayObject)
{
    outputTextBox.Clear();
    outputTextBox.AppendText($"Array {name} bevat:");
    outputTextBox.AppendText(Environment.NewLine);

    for (int i = 0; i < arrayObject.Length; i++)
    {
        outputTextBox.AppendText(arrayObject[i]);
        outputTextBox.AppendText(Environment.NewLine);
    }
}
```

*Bestudeer zelf de functie Sum() in het handboek*

# foreach en arrays

```
private int Sum(int[] array)
{
    int total = 0;
    for (int index = 0; index < array.Length; index++)
    {
        total += array[index];
    }
    return total;
}
```

```
private int Sum(int[] array)
{
    int total = 0;
    foreach (int n in array)
    {
        total += n;
    }
    return total;
}
```

Korter, maar met beperkingen (H13)

# Het gebruik van constanten

Definieer array dimensies altijd in termen van constanten. Zo vermijd je bugs als je later de dimensies wil wijzigen

```
const int MaxAges = 6;  
const int MaxBands = 4;
```

```
int[] ages = new int[MaxAges];  
string[] band = new string[MaxBands];
```

Code conventie:  
constanten beginnen met  
een hoofdletter

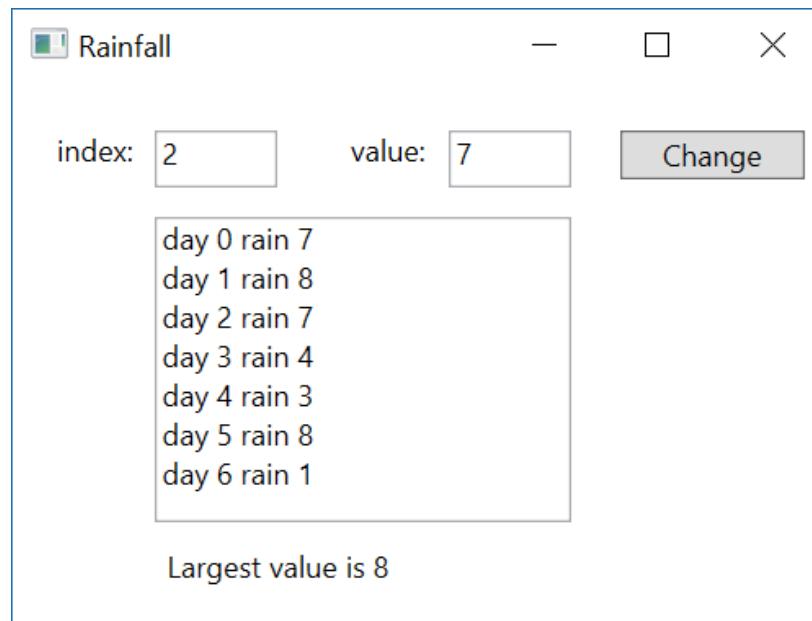
# Een array initialiseren

- Met een lus statement
- Als alle elementen op voorhand gekend zijn, kan je een initialisatie combineren met de declaratie

```
string[] band = {"John", "Paul", "Ringo", "George"};
```

# Een voorbeeldprogramma

- *Bestudeer zelf de broncode*
- *Verklaar de fout die je krijgt als je onmiddellijk op de knop klikt, zonder waarden in te vullen. Welke oplossing stel je voor?*



# Een array als opzoektabel

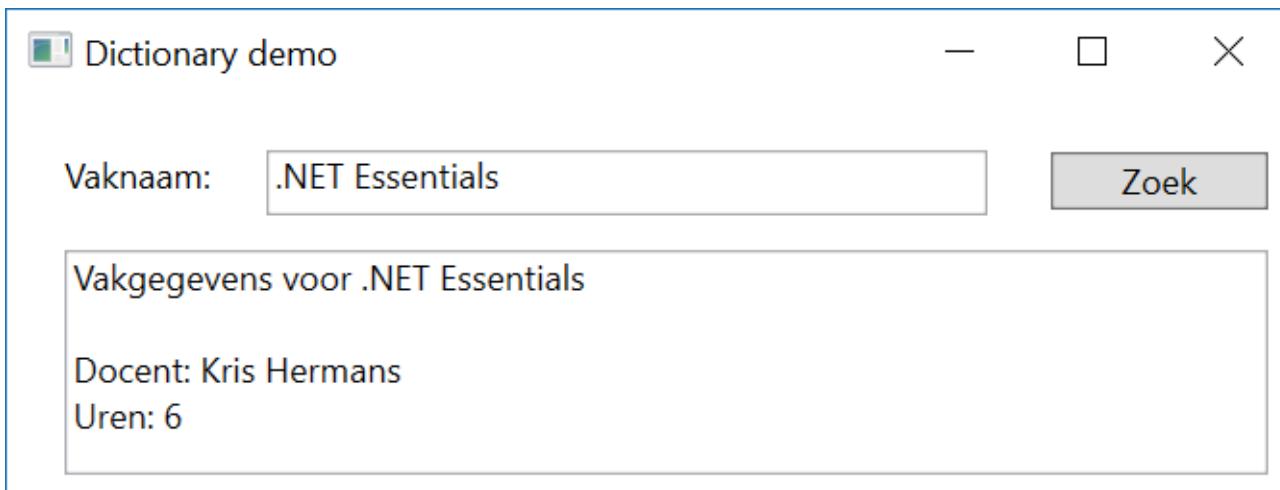
- Probleem: je hebt een zoeksleutel en je wil de bijhorende waarde opzoeken
- In het geval van arrays is de zoeksleutel de index. Typisch gebruik: weekdagen, maanden, seizoenen, ...
- Meer algemeen kan je ook met objecten als zoeksleutel werken. Hiervoor dienen andere datastructuren, bijvoorbeeld: `System.Collections.Generic.Dictionary`
- Een `Dictionary` associeert met elke “key” een “value”. Deze “value” kan *onmiddellijk* op basis van deze “key” gevonden worden.

# Voorbeeld 1

```
int dayNumber;
string dayName;
string[] name = {"Monday", "Tuesday", "Wednesday",
                 "Thursday", "Friday", "Saturday",
                 "Sunday"};

// zoek de dag, gegeven een nummer
dayName = name[dayNumber];
```

# Voorbeeld 2: Dictionary



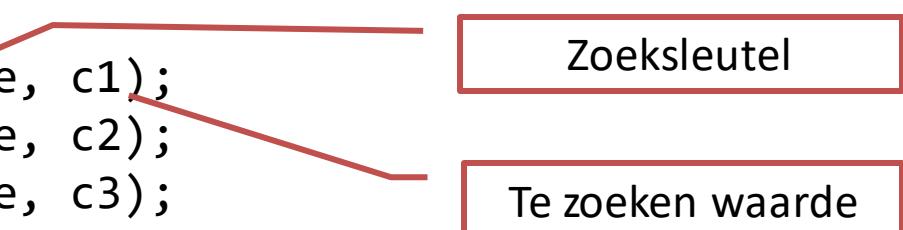
# Voorbeeld 2: Dictionary

```
public class Course
{
    public Course(string name, string teacher, int hours)
    {
        Name = name;
        Teacher = teacher;
        Hours = hours;
    }

    public string Name { get; set; }
    public string Teacher { get; set; }
    public int Hours { get; set; }
}
```

# Voorbeeld 2: Dictionary

```
private Dictionary<string, Course> lookup =  
    new Dictionary<string, Course>();  
  
public MainWindow()  
{  
    InitializeComponent();  
  
    Course c1 = new Course(".NET Essentials", "Kris Hermans", 6);  
    Course c2 = new Course("Logisch & Algoritmisch Denken",  
                           "Nele Custers", 7);  
    Course c3 = new Course("Webdesign", "Rita Lambrechts", 4);  
  
    lookup.Add(c1.Name, c1);  
    lookup.Add(c2.Name, c2);  
    lookup.Add(c3.Name, c3);  
}
```



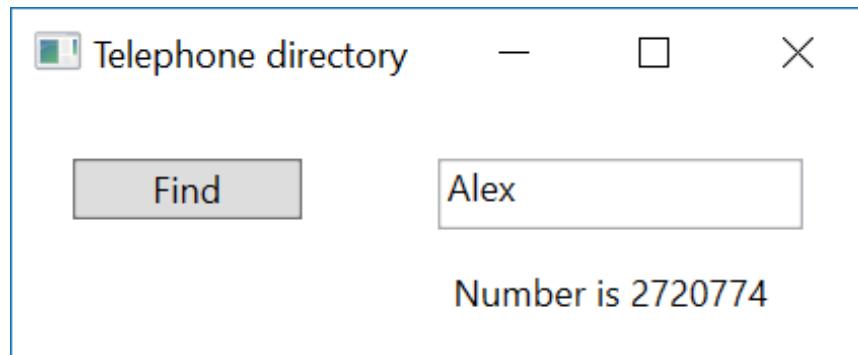
# Voorbeeld 2: Dictionary

```
private void searchButton_Click(object sender, RoutedEventArgs e)
{
    string key;
    Course course;

    if (keyTextBox.Text != "")
    {
        key = keyTextBox.Text;
        if (lookup.ContainsKey(key))
        {
            course = lookup[key];

            resultTextBox.Clear();
            resultTextBox.AppendText($"Vakgegevens voor {key}\n\n");
            resultTextBox.AppendText($"Docent: {course.Teacher}\n");
            resultTextBox.AppendText($"Uren: {course.Hours}");
        }
        else
        {
            MessageBox.Show($"Vak: {key} niet gevonden.");
        }
    }
}
```

# Zoeken



```
names[0] = "Alex";
numbers[0] = "2720774";

names[1] = "Megan";
numbers[1] = "5678554";

names[2] = "END";
```

# Zoeken

```
private void Find()
{
    int index = 0;
    string wanted = nameTextBox.Text;

    while (names[index] != wanted && (names[index] != "END"))
    {
        index++;
    }
    if (names[index] == wanted)
    {
        resultLabel.Content = $"Number is {numbers[index]}";
    }
    else
    {
        resultLabel.Content = "Name not found";
    }
}
```

# Zoeken

- *Hoeveel elementen moet je in het beste/slechtste geval doorlopen om het te zoeken element te vinden?*
- *Hoe evolueert de zoektijd als het aantal elementen in de array stijgt?*
- *Herschrijf dit voorbeeld met een Dictionary en onderzoek opnieuw de zoektijd. Conclusie?*

# Arrays van objecten

- Tot hertoe kennen we volgende datastructuren:
  - Arrays
  - List
  - Dictionary
  - ...
- Elk van deze structuren kan objecten bevatten, zowel afkomstig uit de bibliotheken of instanties van eigen klassen
- *Bestudeer zelf de code van het voorbeeld*

# Tweedimensionale arrays

Hoofdstuk 15

# In dit hoofdstuk ...

- Tweedimensionale arrays:
  - Declareren
  - Initialiseren
  - Indices gebruiken
  - Grootte bepalen
  - Als parameter doorgeven

# Declaratie

Een array kan ook in twee dimensies gedeclareerd worden, bv de verkoopcijfers van computers van 4 winkels:

```
int[,] sales = new int[4, 7];
```

Kolomnummers = dagen

← 0 1 2 3 4 5 6 →

Rijnummers = winkels ↑ 0	22	49	4	93	0	12	32
1	3	8	67	51	5	3	63
2	14	8	23	14	5	23	16
3	54	0	76	31	4	3	99

# Indices

```
sales[2, 3] = Convert.ToInt32(salesTextBox.Text);  
chessboard[3, 4] = fieldTextBox.Text;
```

- eerste index → rijnummer
- tweede index → kolomnummer

# Indices

Elementen verwerken in een tweedimensionale array  
gebeurt vaak met geneste lus-constructies

```
int[,] sales = new int[4, 7];
int sum = 0;

for (int shop = 0; shop < 4; shop++)
{
    for (int dayNumber = 0; dayNumber < 7; dayNumber++)
    {
        sum += sales[shop, dayNumber];
    }
}
```

# De grootte van een array

```
int[,] info = new int[20, 40];  
  
for (int i = 0; i < info.GetLength(0); i++)  
{  
    for (int j = 0; j < info.GetLength(1); j++)  
    {  
        info[i, j] = i + j;  
    }  
}  
  
MessageBox.Show($"Length: {info.Length}");
```

- `info.GetLength(0)`  
→ de lengte voor de eerste dimensie (rijen) = 20
- `info.GetLength(1)`  
→ de lengte voor de tweede dimensie (kolommen) = 40
- `info.Length`  
→ het maximaal aantal elementen in de array = 800

# Arrays als parameters

Geen dimensie toevoegen in de methode declaratie!

```
private int Sum(int[,] array)
```

# Constanten

- Ook bij tweedimensionale arrays zijn constanten nuttig, om nadien eventueel de dimensies te kunnen aanpassen

```
const int DayMaximum = 7;  
const int ShopMaximum = 6;  
  
int[,] sales = new int[ShopMaximum, DayMaximum];
```

# Array initialisatie

```
for (int row = 0; row < 10; row++)  
{  
    for (int col = 0; col < 10; col++)  
    {  
        table[row, col] = 99;  
    }  
}
```

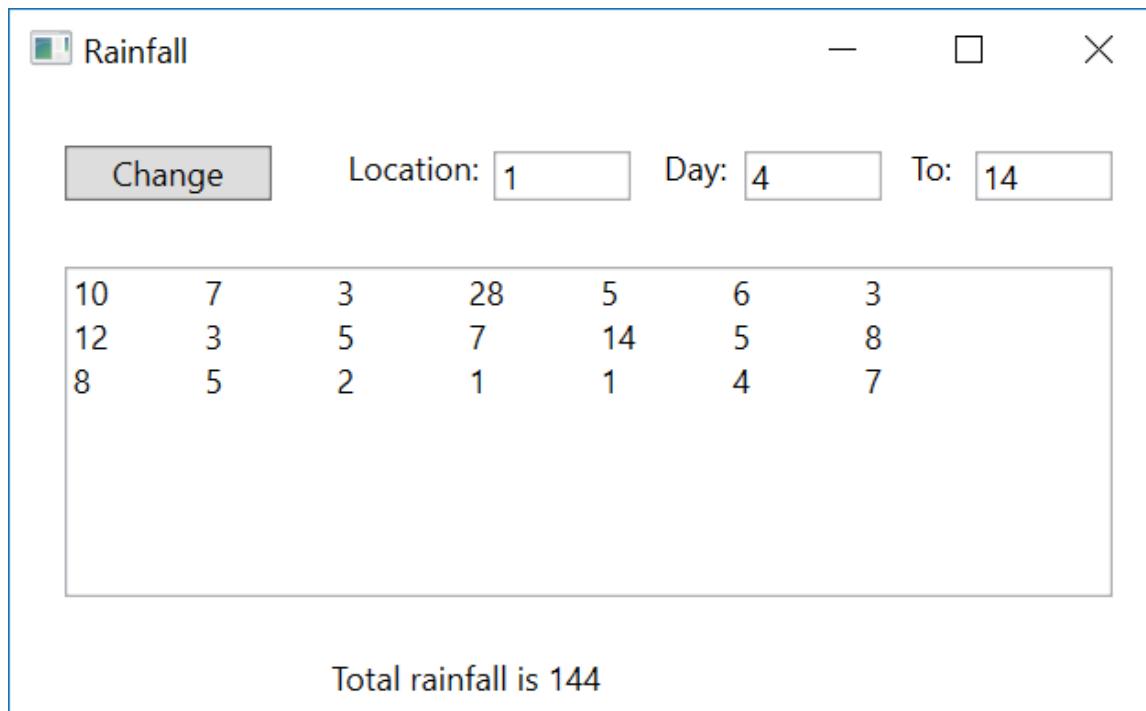
Geneste for-lus

```
int[,] table = {{1, 0, 1},  
                {0, 1, 0}};
```

Bij declaratie

# Een voorbeeldprogramma

*Bestudeer zelf de voorbeeldcode*

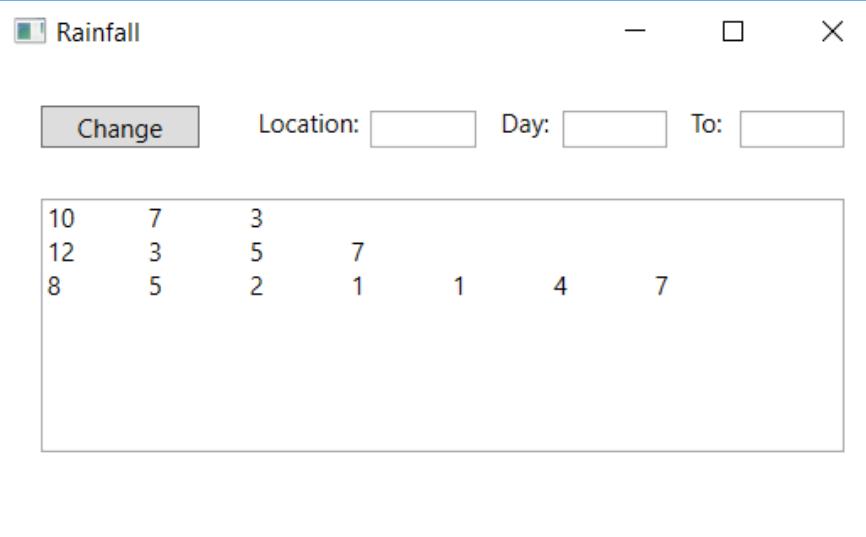


# foreach

```
private void CalculateTotal()
{
    int total = 0;
    foreach (int item in rainData)
    {
        total += item;
    }
    totalLabel.Text = $"Total rainfall is {total}";
}
```

# Jagged arrays

= Rijen van rijen



Location:	Day:	To:
10	7	3
12	3	5
8	5	2
		7
	1	
	4	
		7

Vb: indien in het programma van de neerslaggegevens niet alle metingen beschikbaar zijn → verspilling om de ontbrekende gegevens in de tabel op te vullen met nullen (of een andere waarde) → beter Jagged array

private int[][][] rainData;

Declaratie: [][] i.p.v. [,]

# Jagged arrays

```
rainData = new int[3][]; // er zijn 3 locaties

// of alternatief
//rainData[0] = new int[3];
//                // eerste rij (locatie) heeft 3 metingen
//rainData[1] = new int[4];
//                // tweede rij (locatie) heeft 4 metingen
//rainData[2] = new int[7];
//                // derde rij (locatie) heeft 7 metingen

// of alternatief
rainData[0] = new int[] { 10, 7, 3 };
rainData[1] = new int[] { 12, 3, 5, 7 };
rainData[2] = new int[] { 8, 5, 2, 1, 1, 4, 7 };

Display();
```

# Jagged arrays

```
rainData[0][2] = 5;
```

Toekenning

Lengte van de rijen opvragen

```
int rows = rainData.Length; // aantal rijen: 3  
int col1 = rainData[1].Length;  
// aantal waarden van de rij met index 1
```

# Jagged arrays

```
private void Display()
{
    dataTextBox.Clear();
    for (int locationIndex = 0; locationIndex < rainData.Length; locationIndex++)
    {
        for (int dayNumber = 0; dayNumber < rainData[locationIndex].Length; dayNumber++)
        {
            dataTextBox.AppendText($" {rainData[locationIndex][dayNumber]} \t");
        }
        dataTextBox.AppendText(Environment.NewLine);
    }
}
```

# Bewerkingen met strings

Hoofdstuk 16

# In dit hoofdstuk ...

- Bewerkingen met `String` objecten
- Efficiënt programmeren met `StringBuilder` objecten

# String en string

- `string` is een basistype in C#
- `System.String` is de klasse uit de basisbibliotheek die strings voorstelt
- Dit zijn synoniemen van elkaar, ze stellen dezelfde objecten voor
- Analoog:
  - `string` → `System.String`
  - `bool` → `System.Boolean`
  - `int` → `System.Int32`
  - `double` → `System.Double`
  - `char` → `System.Char`

# Strings gebruiken: herhaling

```
string x;  
string y = "Nederland";
```

Declaratie

```
x = "België";  
y = "Duitsland";  
y = x;  
x = "";  
x = String.Empty;
```

Toekenning

```
MessageBox.Show("Ik woon in " + x);  
MessageBox.Show($"Ik woon in {x}");
```

Concatenatie  
(Samenvoegen)

```
x += " is een land.";
```

Interpolatie  
(invullen)

```
if (x == "Duitsland")  
{  
    // doe iets  
}  
if (x.Equals("Duitsland"))  
{  
    // doe iets  
}
```

Appending  
(Uitbreiden)

Vergelijken

# Strings gebruiken: herhaling

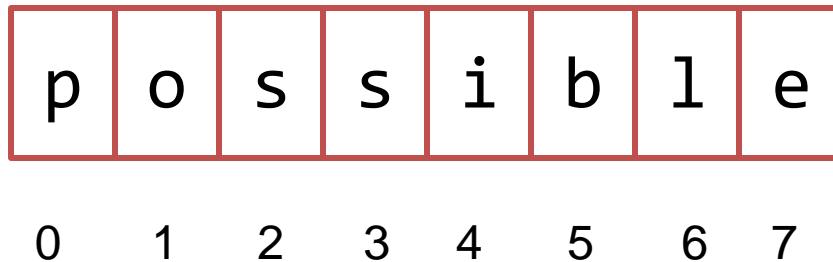
```
string[] cities = new string[10];
```

Array van strings

```
int n = 3;  
x = Convert.ToString(n);  
y = "123";  
n = Convert.ToInt32(y);
```

Typeconversies

# Stringindices



Een `String` object beheert een array van letters. Elke letter wordt voorgesteld door een 2-byte positief getal uit de Unicode tekenset (hetgeen backwards compatible is met ASCII)

# Tekens in een string

- Als je dubbele quotes wil opnemen, dien je deze te “escapen”

```
textBox1.Text = "Het woord \"Object\"";
```

- Nieuwe regel en tab:

```
string s = "Tom" + Environment.NewLine + "Jerry";
s = "Tom\r\nJerry";
x = "een\ttwee\tdrie\tvier";
```

- Escapen met \ of @:

```
x = "Het \\-teken noemt men ook \"backslash\"";
y = "C:\\Program Files\\Microsoft Visual Studio 12.0";

x = @"Het \-teken noemt men ook ""backslash""";
y = @"C:\\Program Files\\Microsoft Visual Studio 12.0";
```

# Het type char

- Strings met 1 teken
- 16bit Unicode karakter
- `char` is dan efficiënter dan `string`

```
char initial = 'M';
char tab = '\t';
char letter;

letter = initial;
if (letter == 'B')
...
initial = Convert.ToChar("x");
string letterAsString = Convert.ToString(letter);
```

# Strings vergelijken

- `==` vergelijkt de waarde van `String` objecten
- `CompareTo` vergelijkt strings alfabetisch

```
string naam1 = "Ella";
string naam2 = "Gilles";

int orde = naam1.CompareTo(naam2);

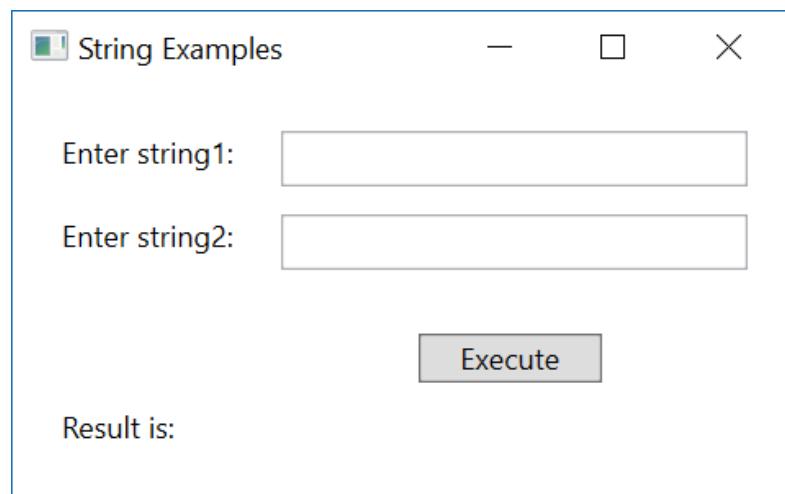
if (orde < 0) // naam1 komt voor naam2
...
else if (orde > 0) // naam2 komt voor naam1
...
else // naam1 en naam2 zijn gelijk
...
```

# Methoden en Properties

## Testprogramma

Je kan nieuwe code uittesten onder  
// place example code here

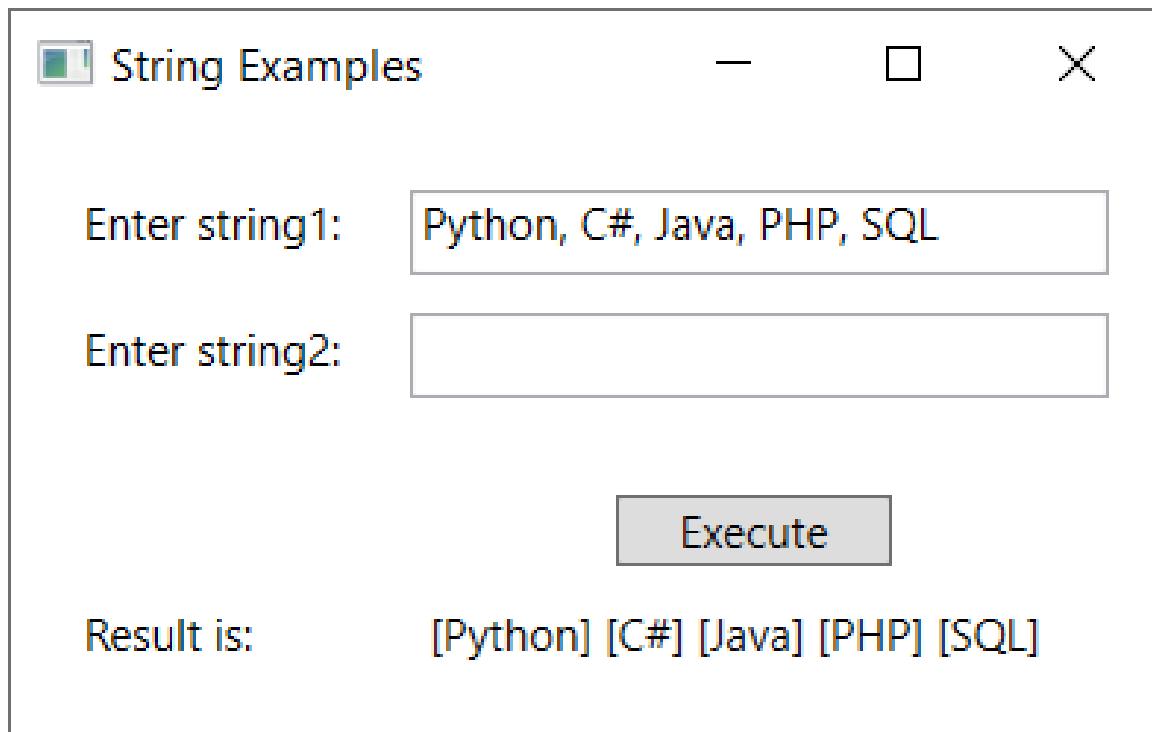
*Formuleer je bedenkingen over deze aanpak ...*



# Methoden en Properties

- Gebruik het testprogramma om volgende methoden uit te proberen
  - ToLower, ToUpper, Trim, Insert, Remove
  - Length, Substring, IndexOf, Split, LastIndexOf, StartsWith, EndsWith
- Bestudeer ook de [online Help](#) van deze methoden en properties

# Voorbeeld: Split



# Voorbeeld: Split

```
private void executeButton_Click(object sender,
                               RoutedEventArgs e)
{
    string string1 = string1TextBox.Text;
    // place example code here
    string resultString = "";
    string[] words;
    char[] separators = { ',', ' ' };
    words = string1.Split(separators);
    for (int place = 0; place < words.Length; place++)
    {
        resultString += $"[{words[place].Trim()}] ";
    }

    resultLabel.Content = resultString;
}
```

# Voorbeelden

- “Een voorbeeld van stringbewerking”  
→ zelf bestuderen
- “Case study: Frasier”  
→ zelf bestuderen

# Bewerkingen met StringBuilder

- Strings zijn *Immutable*
- Elke stringbewerking geeft aanleiding tot een nieuw object

```
string naam = naamTextBox.Text;  
naam = naam.ToUpper();
```

ToUpper() zorgt ervoor dat er een nieuwe string toegekend wordt aan de variabele naam, het oude object wordt weggegooid.

# Bewerkingen met StringBuilder

```
string artikel = "";
while (ErZijnNogWoorden())
{
    artikel = artikel + LeesVolgendWoord();
}
artikelTextBox.Text = artikel;
```

Probleem: zeer veel weggooien van objecten → de garbage collector neemt teveel CPU tijd in beslag ten nadele van het feitelijke programma!

# Bewerkingen met StringBuilder

Een leeg `StringBuilder` object → dient om hierin een string “op te bouwen”

```
StringBuilder artikelBuilder = new StringBuilder();
while (ErZijnNogWoorden())
{
    artikelBuilder.Append(LeesVolgendWoord());
}
artikelTextBox.Text = Convert.ToString(artikelBuilder);
```

Voegt een `string` toe, zonder extra objecten aan te maken en oude weg te gooien

# Exceptions

Hoofdstuk 17

# In dit hoofdstuk ...

- Wat is een exception?
- Waarom zijn ze nuttig?
- De C# exception faciliteiten.

# Inleiding

- Exception → er is een fout opgetreden
- Exception handling → afhandelen van de fout op een gecontroleerde manier
- Voorbeelden:
  - Ongeldige invoer (bv. een letter i.p.v. een getal)
  - Netwerkproblemen (bv. DNS fout)
  - Schijfproblemen (bv. bestand niet gevonden)
  - Hardwareproblemen (bv. geen papier in printer)

# Inleiding

- Syntaxfouten

```
private void calculateButton_Click(object sender, RoutedEventArgs e)
{
    XYZ newValue;
    MessageBox.Show("U heeft geklikt op de knop Calculate");
}
```

Bekijk je fouten in de “Error List”

Error List					
Entire Solution		2 Errors	1 Warning	0 of 1 Message	Build + IntelliSense
Code	Description	Project	File	Line	Suppression S...
CS0246	The type or namespace name 'XYZ' could not be found (are you missing a using directive or an assembly reference?)	MeerdereCatch	MainWindow.xaml.cs	18	Active
CS0103	The name 'MessageBox' does not exist in the current context	MeerdereCatch	MainWindow.xaml.cs	19	Active
CS0168	The variable 'newValue' is declared but never used	MeerdereCatch	MainWindow.xaml.cs	18	Active

# Inleiding

- Run-time fouten
  - een onmogelijke operatie
  - Niet gedetecteerd door de compiler
  - Bv: deling door nul
- Logische fouten
  - Programma produceert foute resultaten
  - Niet altijd een crash tot gevolg!
  - Bv: oneindige lussen
  - Enkel op te lossen door uitvoerig te testen

# Inleiding

- Voorstelling programma met een “normale” werking. Als er nooit fouten zouden optreden is dit correct:

```
MethodeA();  
MethodeB();  
MethodeC();
```

# Inleiding

```
MethodeA();  
if (MethodeA misliep)  
{  
    // handel het methodeA-probleem af  
}  
else  
{  
    MethodeB();  
    if (methodeB misliep)  
    {  
        // handel het methodeB-probleem af  
    }  
    else  
    {  
        MethodeC();  
        if (methodeC misliep)  
        {  
            // handel het methodeC-probleem af  
        }  
    }  
}
```

Oude manier

Ingewikkeld

Niet overzichtelijk

# Jargon

- Als een fout zich voordoet in het programma:
  - Wordt er door de runtime omgeving of door de methode zelf een speciaal object aangemaakt
  - Men zegt dat een exception opgegooid wordt (Engels: *to throw*)
- Hoe afhandelen:
  - Op de gepaste locatie kan men deze exception opvangen (dus niet altijd vlak erna met een if) (Engels: *to catch*)
- Sleutelwoorden: `throw`, `try`, `catch`, `finally`

# Een try-catch voorbeeld

- Demo Exception Square

The image displays two side-by-side screenshots of a Windows application window titled "Exception Square".

**Left Screenshot:** Shows the application's title bar and a text input field containing "2,5". Below the input field is a "Calculate" button. At the bottom, the text "Area is 6,25 square units." is displayed.

**Right Screenshot:** Shows the application's title bar and a text input field containing "2XX5". Below the input field is a "Calculate" button. At the bottom, the text "Error in side, please re-enter." is displayed.

# Een try-catch voorbeeld

```
private void calculateButton_Click(object sender, RoutedEventArgs e)
{
    double side;
    try
    {
        side = Double.Parse(sideTextBox.Text);
        statusLabel.Content = $"Area is {side * side} square units.";
    }
    catch (FormatException)
    {
        statusLabel.Content = "Error in side, please re-enter.";
    }
}
```

Binnen het try blok kan zich een FormatException voordoen, nl: [Parse](#)

De exception wordt afgehandeld in het catch block

Alternatief voor  
Convert.ToDouble

# try-catch: regels

- Zet een **try** blok rond de code die je wil controleren op fouten
- Als in een statement een exception optreedt, stopt de uitvoering onmiddellijk
- Er wordt gesprongen naar het **catch** blok, waar de exception afgehandeld wordt
- Als de exception niet wordt opgevangen, wordt deze doorgegooid naar de oproepende methode
  - Als ook deze ze niet kan opvangen → weer doorgooien
  - Uiteindelijk kom je uit bij de hoofdmethode die je applicatie heeft opgestart  
→ stack trace

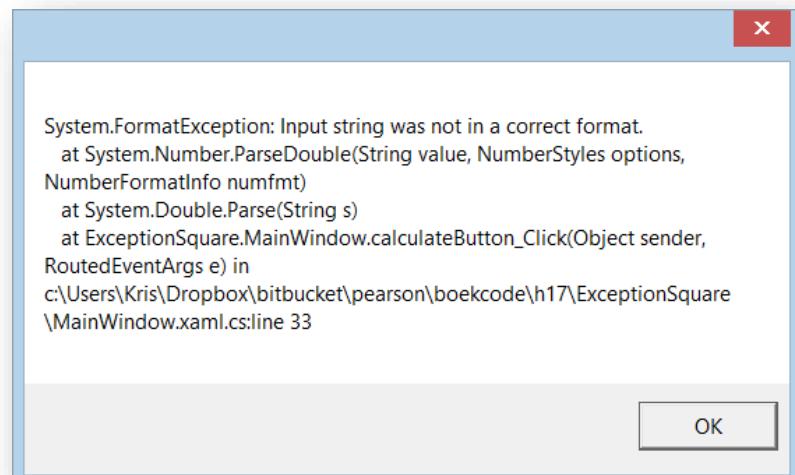
# Een try-catch voorbeeld

```
private void calculateButton_Click(object sender, RoutedEventArgs e)
{
    try
    {
        DoCalc();
    }
    catch (FormatException)
    {
        statusLabel.Content = "Error in side, please re-enter.";
    }
}

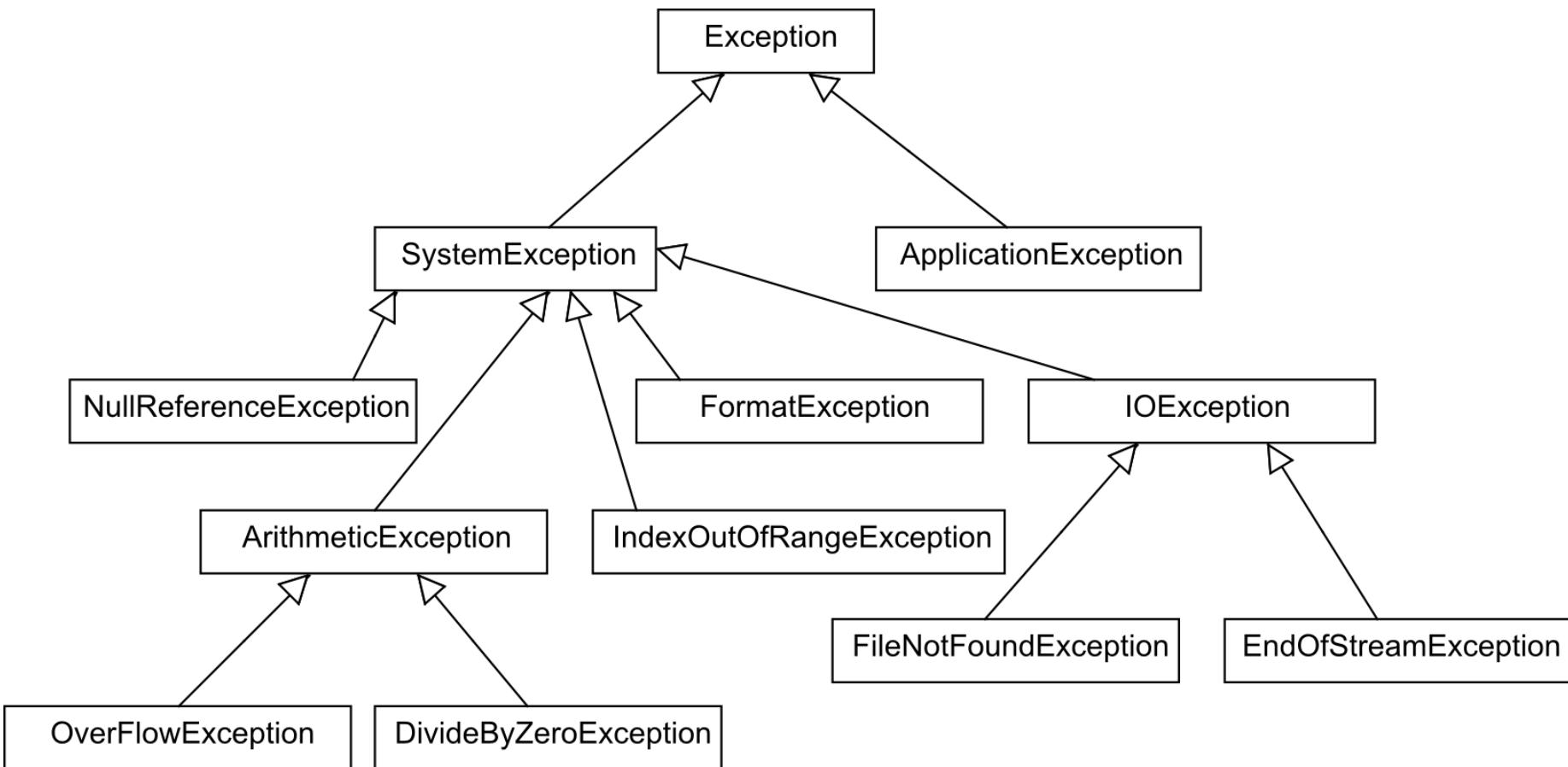
private void DoCalc()
{
    double side = Double.Parse(sideTextBox.Text);
    statusLabel.Content = $"Area is {side * side} square units.";
}
```

# Het exception object

- Bevat nuttige informatie over de aard van de fout
- Tip: lees deze informatie, dit zal je helpen bij het debuggen!
- Properties
  - Message : kort bericht
  - StackTrace : hierarchie van methodes die geleid hebben tot de exception
  - Zie ook: Source, TargetSite, InnerException
- Methode
  - ToString(): string voorstelling van deze exception



# Classificatie



# Meerdere exceptions in 1 catch

```
...  
try  
{  
    SomeOperationWithIO();  
}  
catch (FileNotFoundException)  
{  
    MessageBox.Show("File not found, choose other file");  
}  
catch (EndOfStreamException)  
{  
    MessageBox.Show("End of stream: file corrupt");  
}  
...  
...
```

Ofwel alle specifieke gevallen opvangen,  
zodat je een foutafhandeling hebt per geval

# Meerdere exceptions in 1 catch

```
...  
try  
{  
    SomeOperationWithIO();  
}  
catch (IOException)  
{  
    MessageBox.Show("IOException occurred.");  
}  
...
```

Ofwel 1 catch die alle subklassen van IOException behandelt. Dit is naar de gebruiker toe minder duidelijk.  
(Bestand niet gevonden of corrupt?)

# Combinatie van catch blokken

```
int bottom;
int top = 100;
try
{
    bottom = Int32.Parse(bottomTextBox.Text);
    statusLabel.Content = $"Dividing into 100 gives {top / bottom}";
}
catch (DivideByZeroException)
{
    statusLabel.Content = "Error - zero: re-enter data.";
}
catch (FormatException)
{
    statusLabel.Content = "Error in number: re-enter.";
}
catch (SystemException exceptionObject)
{
    MessageBox.Show(exceptionObject.ToString());
}
```

Vang elk type van exception afzonderlijk op

Regel: hoe algemener de exception (SystemException), hoe later opvangen in het try-catch statement (waarom?)

# Samenvatting

- Methode1()
  - Methode2()
    - Methode3() → Exception treedt op
      - Uitvoering van Methode3() wordt onmiddellijk gestopt
      - Als er een catch statement is, die deze Exception (of een superklasse ervan) opvangt, wordt deze uitgevoerd. De uitvoering gaat vervolgens verder na het try-catch blok.
      - Als er geen catch statement is, gaat de Exception naar de oproeper (Methode2()), indien deze ook geen catch statement heeft naar Methode1(), enz. Als er helemaal geen catch wordt gevonden, breekt het programma af met een foutmelding.
      - Exception propagation (voortplanting)

# Opgooien: een inleiding

```
private int WordToNumber(string word)
{
    int result = 0;
    if (word == "ten")
    {
        result = 10;
    }
    else if (word == "hundred")
    {
        result = 100;
    }
    else if (word == "thousand")
    {
        result = 1000;
    }
    else
    {
        throw new FormatException("Wrong input: " + word);
    }
    return result;
}
```

# Opgooien: een inleiding

```
private void convertButton_Click(object sender, RoutedEventArgs e)
{
    try
    {
        MessageBox.Show(Convert.ToString(WordToNumber("hXndred")));
    }
    catch (FormatException exceptionObject)
    {
        MessageBox.Show(exceptionObject.Message);
    }
}
```

# Hoe afhandelen

- Zinvolle foutmelding naar de gebruiker toe, eventueel vragen om nieuwe invoer
- Bij waarschijnlijke bugs, exceptions loggen naar bestanden of Event logs
- ***Nooit lege catch statements schrijven om exceptions te “verbergen”***

# finally

```
DatabaseConnection resource = ...;  
try  
{  
    resource.Open();  
    voer queries uit naar de database  
}  
catch (SQLException exceptionObject)  
{  
    toon foutmelding  
}  
finally  
{  
    // ruim de connectie op  
    resource.Close();  
}
```

finally wordt altijd  
uitgevoerd, exception of niet!

# Zelf exceptions schrijven

- In bepaalde omstandigheden zijn de ingebouwde exceptions van het .NET framework onvoldoende
- Bijvoorbeeld, je wil fouten die voor een bepaalde toepassing specifiek zijn, op dezelfde manier met exceptions afhandelen
- Hoe?
  - Schrijf zelf een klasse die overerft van Exception
  - **Fout in boek pag. 431:**  
***ApplicationException niet meer gebruiken  
(door Microsoft afgeraden)***

# Zelf exceptions schrijven

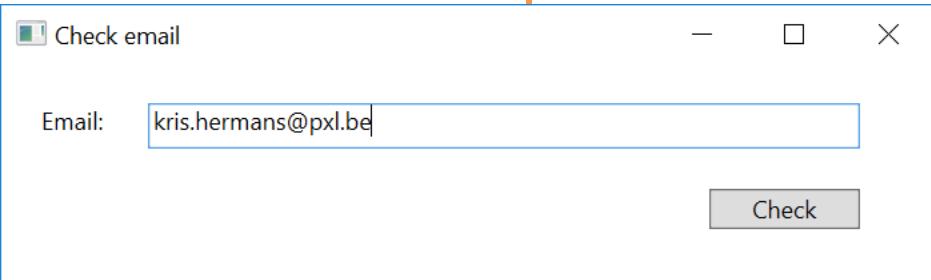
```
public class InvalidEMailException
    : Exception
{
    public InvalidEMailException(string message)
        : base(message)
    {
    }
}
```

Voorzie steeds een constructor die een message parameter doorgeeft. Dit is de eigenlijke foutmelding

# Zelf exceptions schrijven

```
private void checkButton_Click(object sender, RoutedEventArgs e)
{
    try
    {
        CheckAddress(mailTextBox.Text);
    }
    catch (InvalidEMailException ex)
    {
        MessageBox.Show(ex.Message);
    }
}

private void CheckAddress(string email)
{
    if (!email.Contains('@'))
    {
        throw new InvalidEMailException(email +
            " does not contain @-sign!");
    }
    // other validation rules
    ...
}
```



# Uitbreiding van de exception klasse

```
public class InvalidEmailException  
    : Exception  
{  
    public InvalidEmailException(string message)  
        : base(message)  
    {}  
  
    public int Severity { get; set; }  
}
```

Severity geeft een getal:

Groter dan 10: ernstig (onbestaand adres)  
Kleiner dan 10: lichte fout (typo)

# Voorwaardelijk afhandelen

```
try
{
    checkaddress(email);
}
catch (invalidemailexception ex)
{
    if (ex.severity <= 10)
    {
        messagebox.show("please enter a valid address");
    }
    else
    {
        log(ex.message);
    }
}
```

Omslachtig met if in catch-gedeelte.  
Tip: catch-blokken best kort houden!



# Voorwaardelijk afhandelen met catch ... when

```
try
{
    CheckAddress(email);
}
catch(InvalidOperationException ex) when (ex.Severity <= 10)
{
    MessageBox.Show("Please enter a valid address");
}
catch (InvalidOperationException ex) when (ex.Severity > 10)
{
    Log(ex.Message);
}
```

# Bestanden

Hoofdstuk 18



# In dit hoofdstuk ...

- Tekstbestanden
- Lezen en schrijven
- Mappen (Directories)
- MessageBox
- Standaard dialoogvensters
- Menu's
- Meerdere vensters (Windows)

# Inleiding

- Opslaan van gegevens
  - RAM
    - Korte toegangstijd
    - Duurder
    - Tijdelijk
  - Opslagmedia
    - Langere toegangstijd, maar grotere capaciteit
    - Goedkoper
    - Relatief permanent

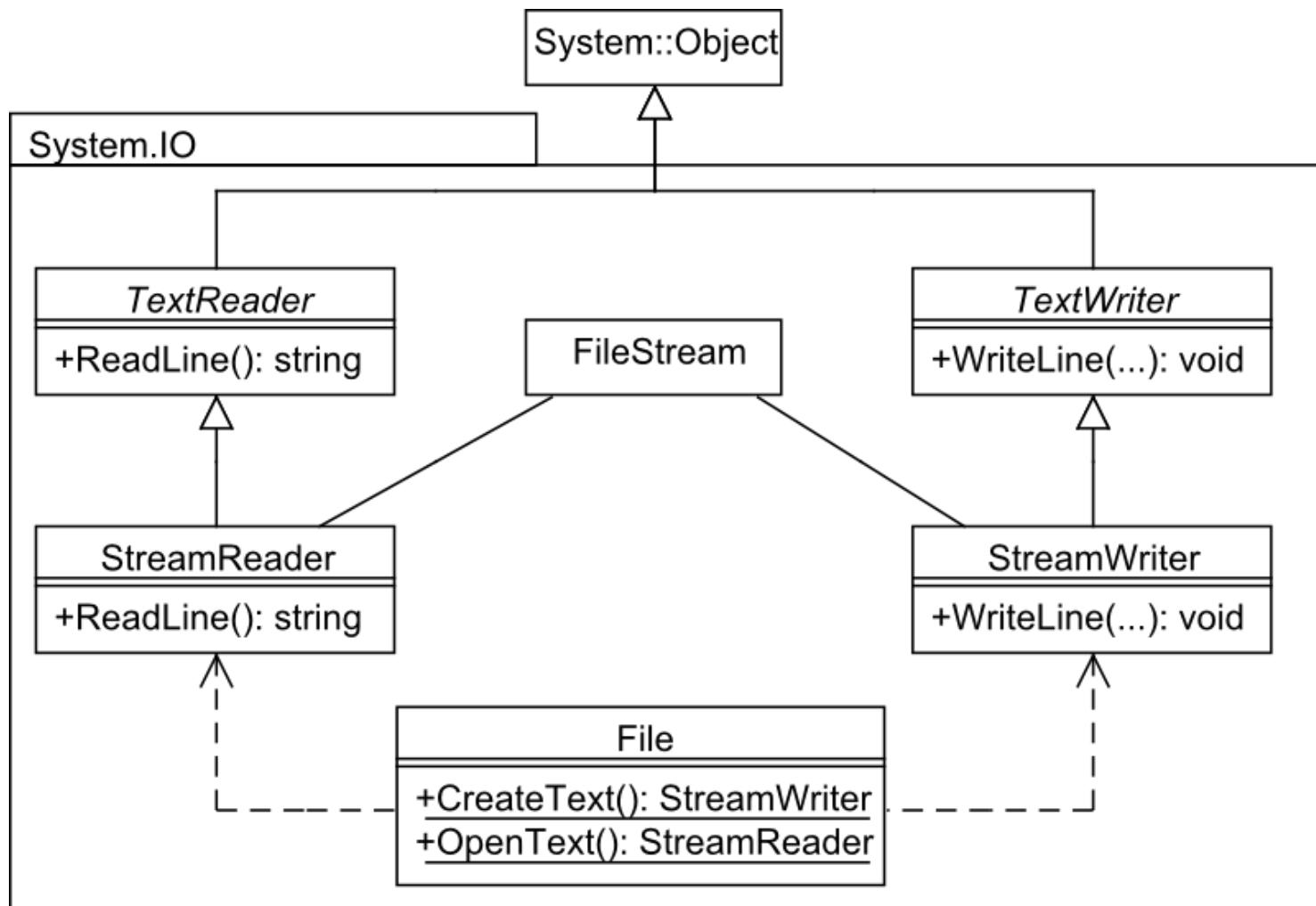
# De basisbegrippen van streaming

- Een bestand wordt beschouwd als een stroom gegevens die in één doorlopende (sequentiële) beweging wordt gemanipuleerd

1. Open bestand
2. Inlezen gegevens van stream *of*  
Wegschrijven gegevens naar stream
3. Bestand sluiten

- Een bestand is een “system resource”. Niet vergeten te sluiten!  
Hoe kan je dit garanderen?

# Overzichtsdiagram



# Bestandsuitvoer

```
using System.IO;  
...  
private void writeButton_Click(object sender,  
                               RoutedEventArgs e)  
{  
    StreamWriter writer = File.CreateText("myfile.txt");  
    writer.WriteLine("This file will");  
    writer.WriteLine("contain 3");  
    writer.WriteLine("lines of text.");  
    writer.Close();  
}
```

# Bestandsuitvoer

- `using System.IO;`
- `File.CreateText(bestandsnaam)`
  - Maakt een `StreamWriter` object naar een tekstbestand
  - Als dit bestand nog niet bestaat, wordt het aangemaakt.  
Anders wordt het bestaande bestand overschreven
- `writer.WriteLine(string)`
  - Schrijft de `string` naar de stream (en dus in het bestand)  
en sluit af met een `NewLine` teken
- `writer.Close()`
  - Sluit de stream af
  - Dit statement zou beter in een `finally` block staan

# Bestandslocaties

```
StreamWriter outputStream =  
    File.CreateText("myfile.txt");
```

- Het bestand komt terecht in dezelfde locatie als het .exe-bestand
  - Projectfolder\bin\Debug
- Bij uitvoeren van programma geen schrijfrecht (UnauthorizedAccessException), bv:
  - C:\Program Files\File Output

# Bestandslocaties

```
StreamWriter outputStream =  
    File.CreateText(@"C:\myfile.txt");
```

- Ofwel:
  - Access denied
  - Omleiding van het bestand naar andere locatie
  - Lukt alleen met Administrator rechten
- Dus schrijven naar C:\ is geen goed idee!

# Bestandslocaties

- Environment.SpecialFolder bevat constanten naar veelgebruikte locaties, ondermeer om gegevens van de gebruiker weg te schrijven:
  - ApplicationData
  - MyPictures
  - MyDocuments
  - Enz.

# File Output herwerkt:

```
private void writeButton_Click(object sender, RoutedEventArgs e)
{
    string folderPath = Environment.GetFolderPath(
        Environment.SpecialFolder.MyDocuments);
    statusLabel.Content = "Writing to: " + folderPath;

    string filePath = System.IO.Path.Combine(
        folderPath, "myfile.txt");
    StreamWriter writer = File.CreateText(filePath);
    writer.WriteLine("This file will");
    writer.WriteLine("contain 3");
    writer.WriteLine("lines of text.");
    writer.Close();
}
```

Zoek “Mijn Documenten”, bv:  
C:\Users\Kris\Documents

Vorm compleet pad, bv:  
C:\Users\Kris\Documents\myfile.txt

Namespace System.IO vermelden omdat er anders verwarring kan ontstaan met  
de klasse Path uit de namespace System.Windows.Shapes

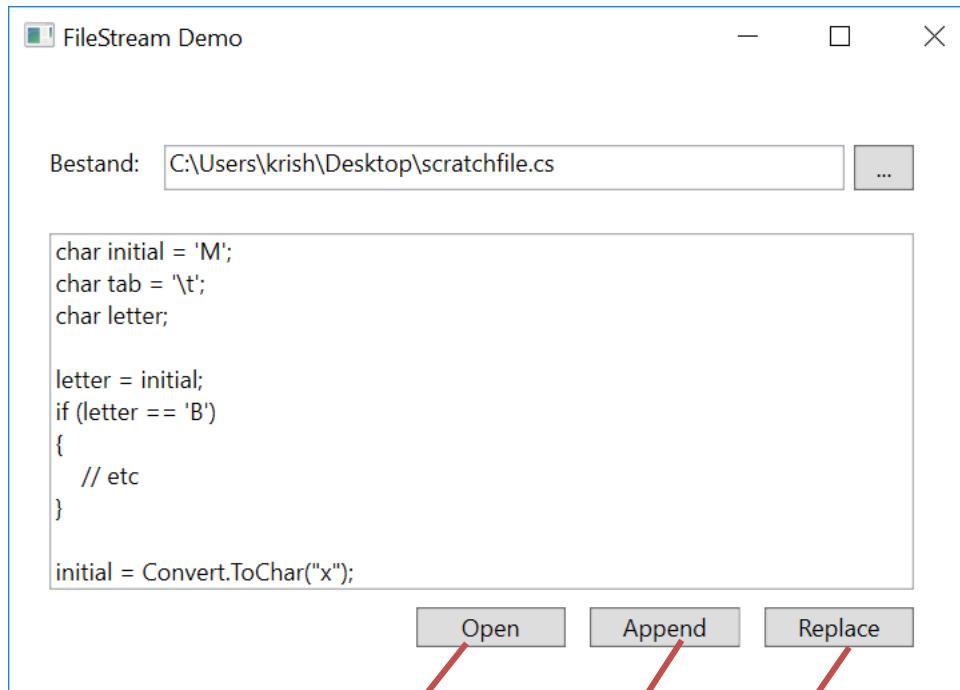
# Bestandsinvoer

```
private void readButton_Click(object sender, RoutedEventArgs e)
{
    string folderPath = Environment.GetFolderPath(
        Environment.SpecialFolder.MyDocuments);
    string filePath = System.IO.Path.Combine(folderPath, "myfile.txt");
    StreamReader reader = File.OpenText(filePath);
    string line = reader.ReadLine();
    while (line != null)
    {
        fileTextBox.AppendText(line);
        fileTextBox.AppendText(Environment.NewLine);
        line = reader.ReadLine();
    }
    inputStream.Close();
}
```

# Bestandsinvoer

- `using System.IO;`
- `File.OpenText(bestandsnaam)`
  - Maakt een StreamReader object naar een (bestaand) tekstbestand
  - Als dit bestand nog niet bestaat:  
`FileNotFoundException`
- `reader.ReadLine()`
  - Leest een (volgende) regel uit het bestand tot null wordt teruggegeven
- `reader.Close()`
  - Sluit de reader af
  - Dit statement zou beter in een `finally` block staan

# FileStreamDemo



- Openen en tonen
- Toevoegen aan bestand
- Bestand overschrijven

# FileStream

- Meer algemene manier om een stream naar een bestand te krijgen
- Verschillende wijzen van toegang
  - FileAccess
  - FileMode
- Eens je een `FileStream` object hebt, kan je ermee een `StreamReader` of `StreamWriter` mee aanmaken
- [Microsoft documentatie](#)

# FileMode

- Append: open op het einde of nieuw
- Create: nieuw of overschrijf
- CreateNew: nieuw of IOException
- Open: open of FileNotFoundException
- OpenOrCreate: open of nieuw
- Truncate: openen en overschrijven

# FileAccess

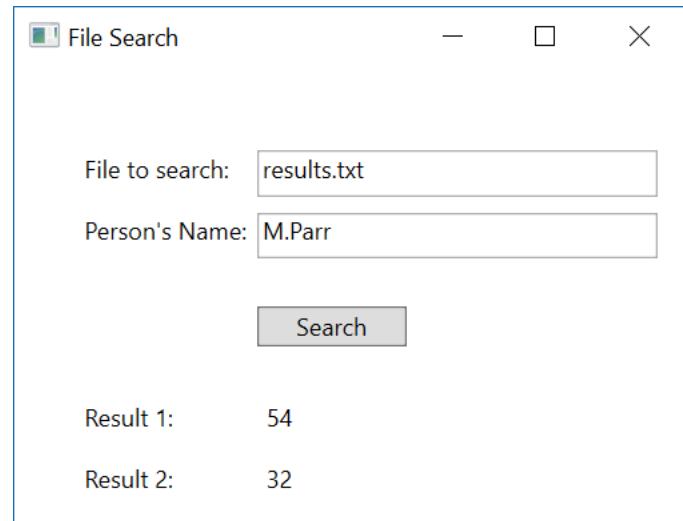
- **Read:** enkel lezen
- **Write:** enkel schrijven
- **ReadWrite:** lezen en schrijven

# FileStream Demo

```
private void openButton_Click(object sender, RoutedEventArgs e)
{
    StreamReader reader = null;
    try
    {
        FileStream fstream = new FileStream(currentFile,
                                              FileMode.Open,
                                              FileAccess.Read);
        reader = new StreamReader(fstream);
        mainTextBox.Text = reader.ReadToEnd();
    }
    catch (FileNotFoundException)
    {
        MessageBox.Show(textBox1.Text + " niet gevonden!");
    }
    finally
    {
        if (reader != null)
        {
            reader.Close();
        }
    }
}
```

# Bestanden doorzoeken

- *Bestudeer de broncode*
- *Waarom de dubbele conditie in de while lus?*
- *Wat gebeurt er als we bij de dubbele conditie een || schrijven i.p.v. een &&, en als de naam niet wordt gevonden?*



# Bestanden en exceptions

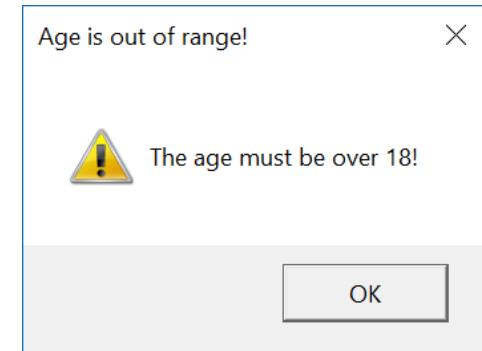
- Vrijwel elke IO operatie kan mislopen
  - FileNotFoundException
  - IOException
- Kijk dus goed naar de online documentatie en vang indien nodig de exception op
- Zeker: streams afsluiten (`Close`) in een `finally` block. Anders zou dit kunnen aanleiding geven tot memory leaks

# MessageBox

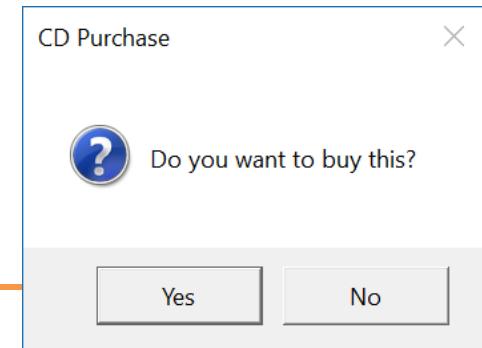
- Verschillende versies (overloading)
  - MessageBox.Show(*boodschap*)
  - MessageBox.Show(*boodschap, titel*)
  - MessageBox.Show(*boodschap, titel, knoppen*)
  - MessageBox.Show(*boodschap, titel, knoppen, pictogram*)
- Vaststellen welke knop geklikt werd
  - MessageBoxResult.Cancel
  - MessageBoxResult.No
  - MessageBoxResult.None
  - MessageBoxResult.OK
  - MessageBoxResult.Yes
  - ...
- [Microsoft Documentatie](#)

# MessageBox

```
MessageBox.Show("The age must be over 18!",  
    "Age is out of range!",  
    MessageBoxButtons.OK,  
    MessageBoxIcon.Exclamation);
```



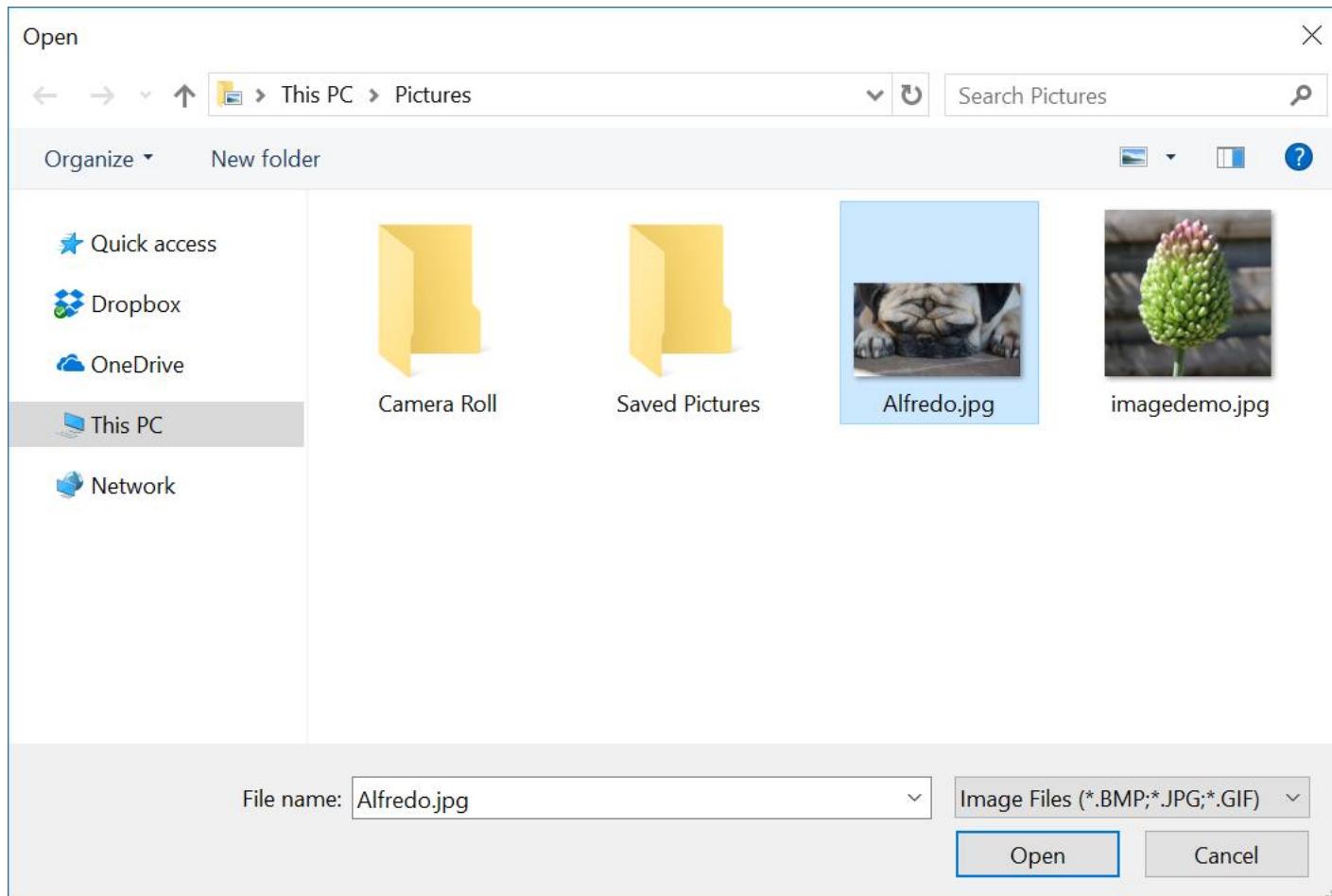
```
if (MessageBox.Show("Do you want to buy this?",  
    "CD Purchase",  
    MessageBoxButtons.YesNo,  
    MessageBoxIcon.Question) == DialogResult.Yes)  
{  
    MessageBox.Show("User clicked yes");  
}  
else  
{  
    MessageBox.Show("User clicked no");  
}
```



# OpenFileDialog

- Namespace Microsoft.Win32
- Instantie aanmaken van OpenFileDialog
- Je kan standaardmap instellen (InitialDirectory)
- Gebruik van Filters (Filter)
- Naam van het gewenste bestand (FileName)
- Eveneens: SaveFileDialog → werkt compleet analoog

# OpenFileDialog



# OpenFileDialog

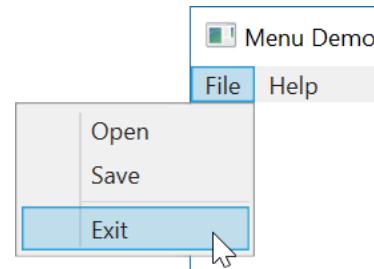
```
 OpenFileDialog dialog = new OpenFileDialog();
string initialFolderPath = Environment.GetFolderPath(
    Environment.SpecialFolder.MyPictures);
dialog.InitialDirectory = initialFolderPath;
dialog.Filter = "Image Files|*.BMP;" +
    "*.JPG;*.GIF|All files (*.*)|*.*";

if (dialog.ShowDialog() == true) //User clicks Open
{
    MessageBox.Show(dialog.FileName);
}
```

Retourneert een *nullable* bool (aangeduid: `bool?`, of `Nullable<bool>`)  
→ expliciet controleren met het `if`-statement

# Een menu creëren

- Wordt volledig via XAML opgebouwd
- Hiërarchie:
  - Window
  - DockPanel
  - Menu
    - MenuItem
- Separator
- Attributen van MenuItem:
  - Header
  - Name
  - Click
- Underscore → sneltoets ALT+onderlijnde letter



# Een menu creëren

```
<Window ...>
    Title="Menu Demo" Height="350" Width="525">
    <DockPanel>
        <Menu DockPanel.Dock="Top">
            <MenuItem Header="_File">
                <MenuItem Header="_Open" />
                <MenuItem Header="_Save" />
                <Separator />
                <MenuItem Header="E_xit" Name="exitItem" Click="exitItem_Click"/>
            </MenuItem>
            <MenuItem Header="_Help">
                <MenuItem Header="_About" />
            </MenuItem>
        </Menu>
        <Grid DockPanel.Dock="Bottom">
            <!-- Hier komt de inhoud-->
        </Grid>
    </DockPanel>
</Window>
```

# Menu event

```
private void exitItem_Click(object sender, RoutedEventArgs e)
{
    Environment.Exit(0);
}
```

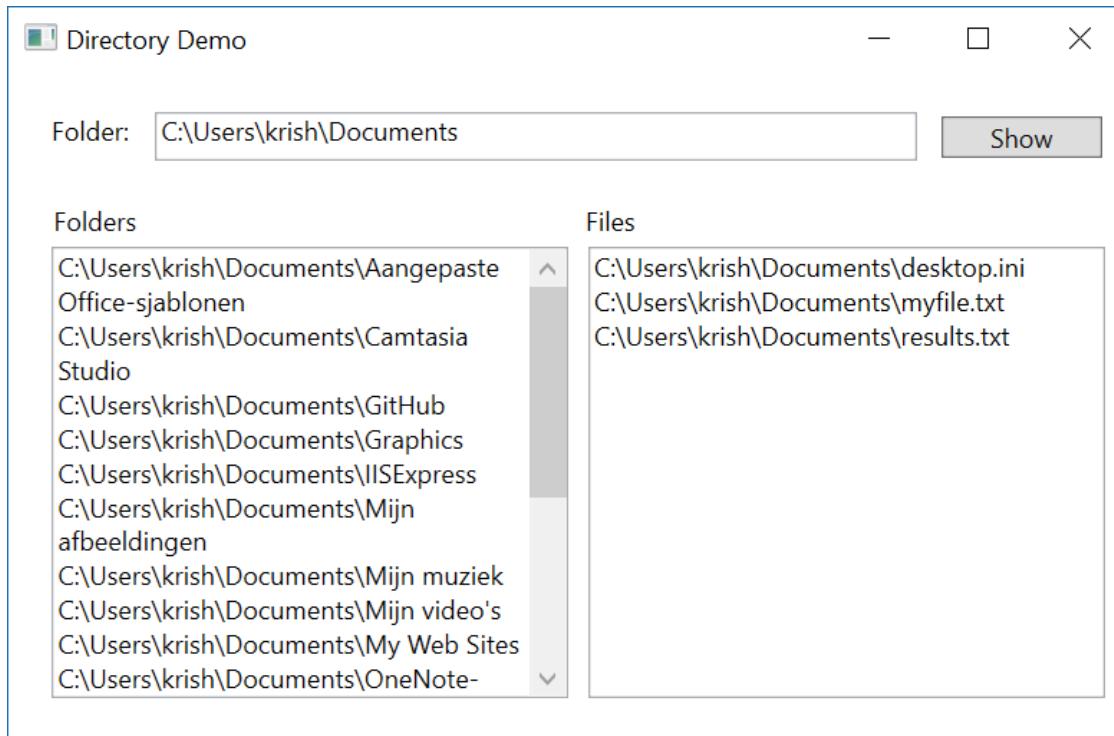
- Gebruik `Environment.Exit(0)` om een programma af te sluiten

# De klasse Directory

- Faciliteiten voor mappen/bestanden
- `using System.IO;`
- `GetFiles(path)`
  - Retourneert een array van strings; de bestandsnamen van files in directory path
- `GetDirectories(path)`
  - Retourneert een array van strings; de namen van de directories in directory path
- Bestudeer zelf de andere methodes in [Directory](#)

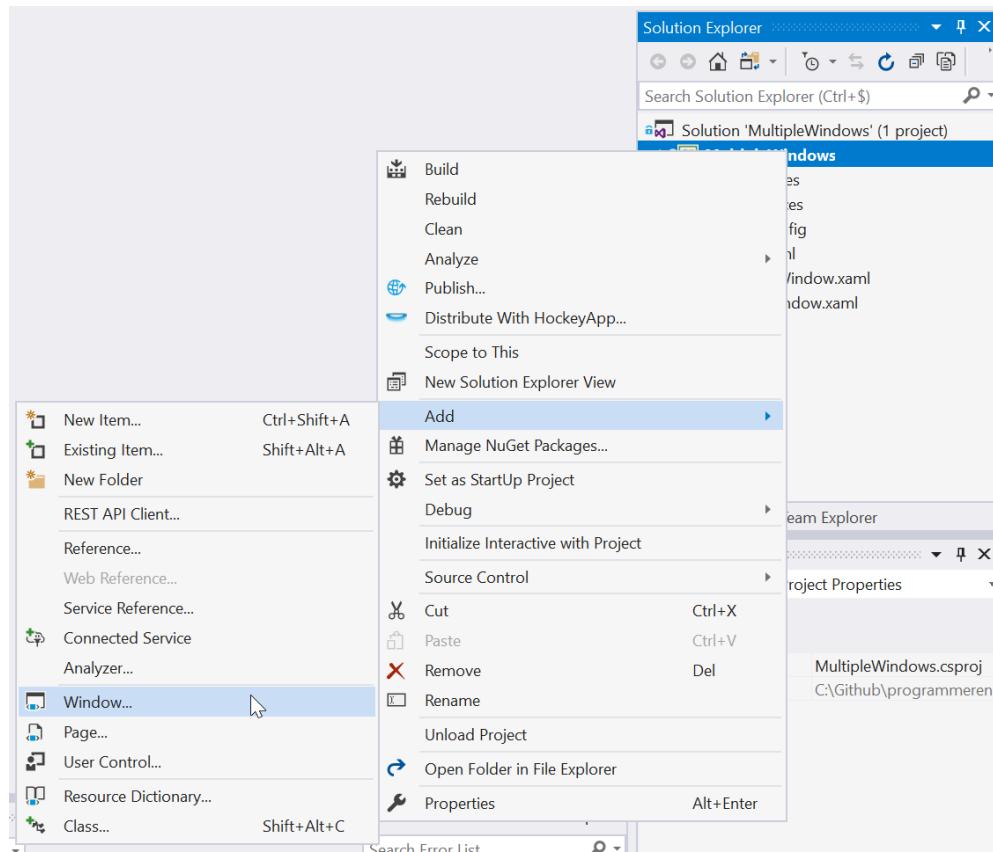
# De klasse Directory

- Directory Demo: *bestudeer zelf de code*



# Meerdere Windows

Een nieuwe Window (venster) kan je toevoegen door rechtsklik op het project → Add → Window...



# Meerdere Windows

De window die opgestart wordt bij de programmastart, zie je in App.xaml

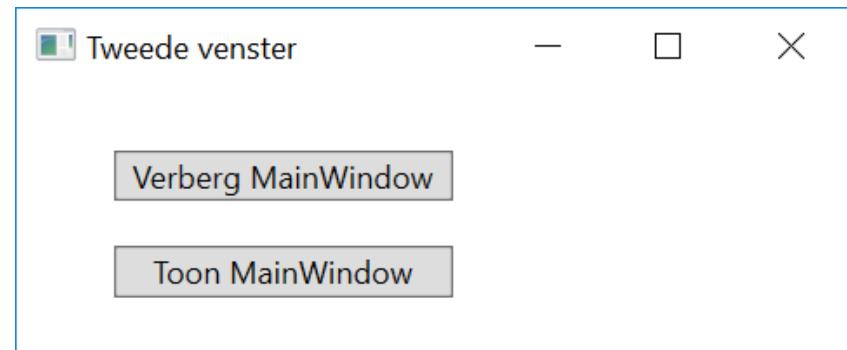
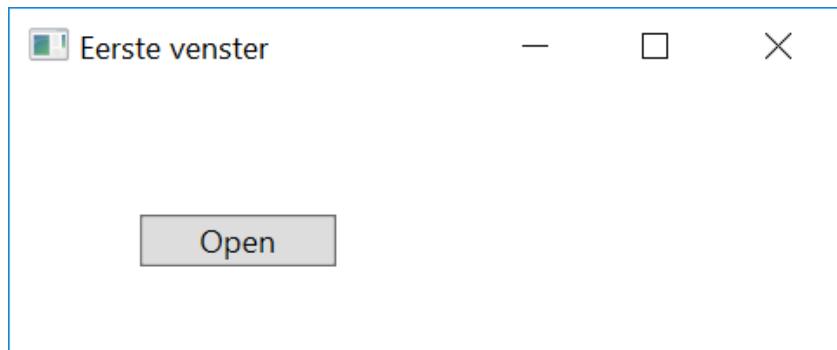
```
<Application x:Class="MultipleWindows.App"  
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"  
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"  
    StartupUri="MainWindow.xaml">  
    <Application.Resources>  
        </Application.Resources>  
</Application>
```

# Meerdere Windows

- Een Window is een klasse, d.w.z. dat je een instantie moet maken om een window te openen. Vervolgens roep je op dit object de methode Show() op.
- Als je vanuit één Window een andere window wil manipuleren, dien je referenties naar deze objecten door te geven (via de constructor in de demo)
- Een Window verbergen zonder de applicatie af te sluiten doe je met Hide()

# Meerdere vensters

- Demo



# Consoleprogramma's

Hoofdstuk 19

# In dit hoofdstuk ...

- Hoe consoletoepassingen maken
- In- en uitvoer bij consoleprogramma's
- Verwerking van argumenten
- De prompt
- Batchbestanden en scripts

# Inleiding

- Programma's die enkel draaien via de commandoregel zijn nog steeds uiterst belangrijk:
  - Sneller dan Windows programma's
  - Kunnen automatisch gestart worden (services)
  - Vereisen weinig of geen interactie
  - Vaak zeer handig
- Voorbeelden:
  - "DOS commando's": dir, xcopy, cd, ...
  - netwerk clients: ftp, ssh, telnet, ...
  - Utilities: zip, csc, ...
- Vaak worden deze programma's met elkaar verbonden tot nieuwe programma's → scripting
  - Batch (.bat) bestanden
  - PowerShell: een geavanceerde "prompt" voor Windows is volledig met .NET geïntegreerd

# Een eerste consoleprogramma

- Met Visual Studio kies je voor ‘Console Application’ als project type
  - Twee mogelijkheden
    - Windows Classic Desktop
    - .NET Core
- Demo Hello
  - We kiezen voorlopig voor “Windows Classic Desktop”

# Een eerste consoleprogramma

```
class Program
{
    static void Main(string[] args)
    {
        Console.WriteLine("Please enter your name:");
        string name = Console.ReadLine();
        Console.WriteLine($"Hi there {name}");
        string wait = Console.ReadLine();
    }
}
```

# Een eerste consoleprogramma

- Program.cs: aangemaakt door de IDE en bevat de Main()-methode
- Main(string[] args): de “start”-methode van het programma, argumenten via array
- Console: lezen en schrijven naar de console via ReadLine en WriteLine
- Het uitvoerbaar programma kan je ook terugvinden in de bin\Debug directory van je project

# DOS commando's

- Bestaan al sinds de voorloper van het Windows besturingssysteem
- Voorbeelden:
  - cd
  - dir
  - xcopy
- Via de switch /? kan je hulp over het commando verkrijgen

# Programma's runnen

- Vanuit de IDE
- Dubbelklikken vanuit Windows Explorer
- Naam intikken in een Consolevenster
- Vanuit een batch bestand

```
REM het volgende batch bestand voert hello.exe
REM tweemaal na mekaar uit
Hello.exe
Hello.exe
```

# Argumenten

- Als je een Main methode maakt met een array van strings, dan zal deze array de commando argumenten bevatten

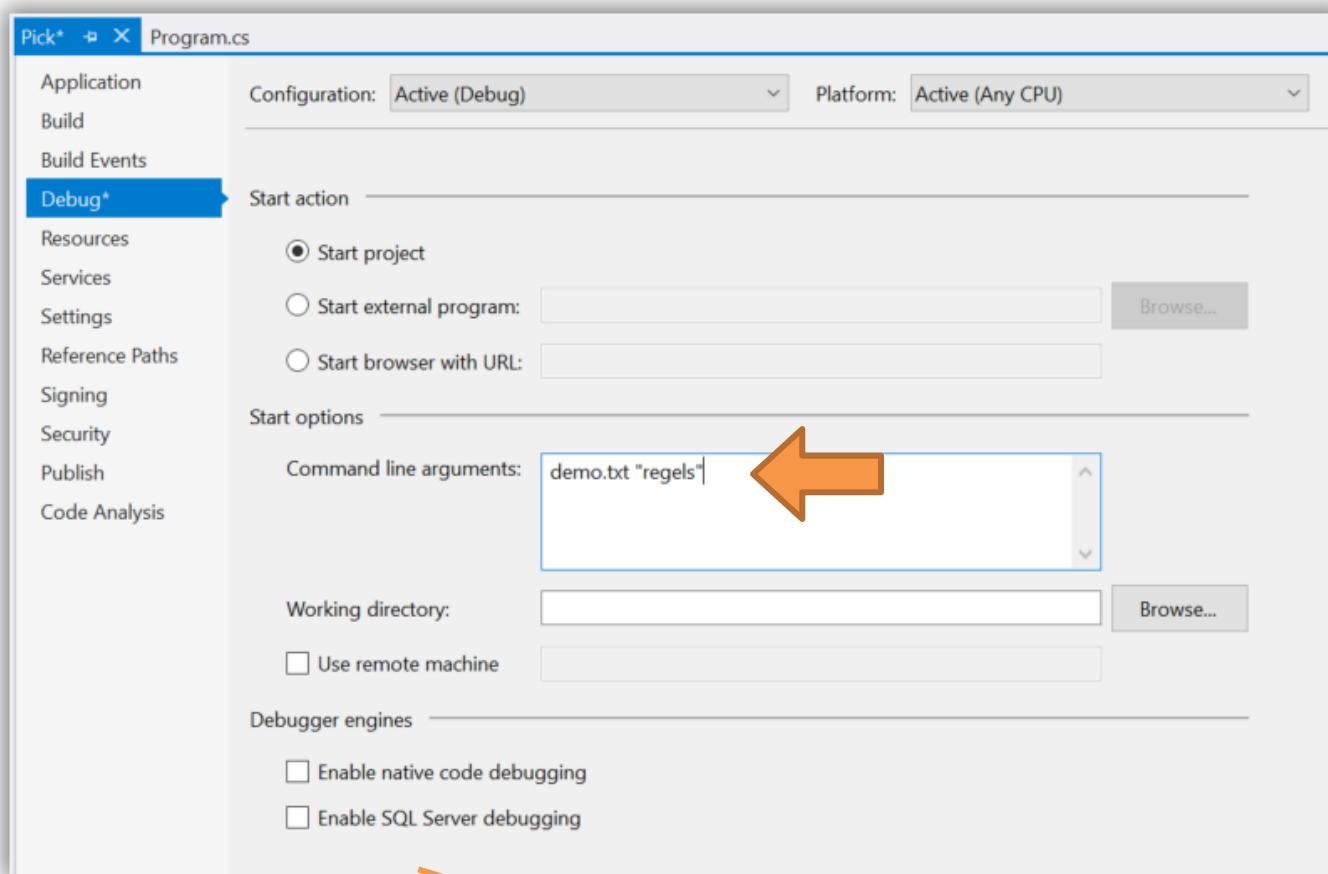
```
static void Main(string[] args)
```

# Argumenten

```
static void Main(string[] args)
{
    string fileName = args[0];
    string wanted = args[1];
    StreamReader reader = File.OpenText(fileName);
    string line = reader.ReadLine();
    while (line != null)
    {
        if (line.IndexOf(wanted) >= 0)
        {
            Console.WriteLine(line);
        }
        line = reader.ReadLine();
    }
    reader.Close();
}
```

Wat gebeurt er als je geen argumenten meegeeft aan het programma?

# Argumenten in VS



Dit scherm verschijnt door te rechtsklikken op het project en te kiezen voor “Properties”

# Pipes maken

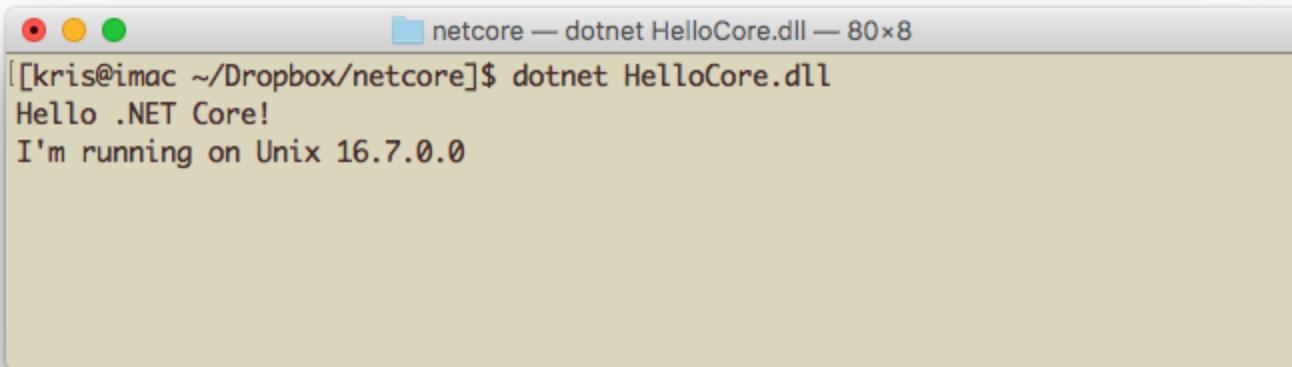
- >
  - De uitvoer naar een bestand sturen en overschrijven als het al bestond
- >>
  - De uitvoer toevoegen aan een bestand
- <
  - De invoer lezen van een bestand
- Demo

# .NET Core

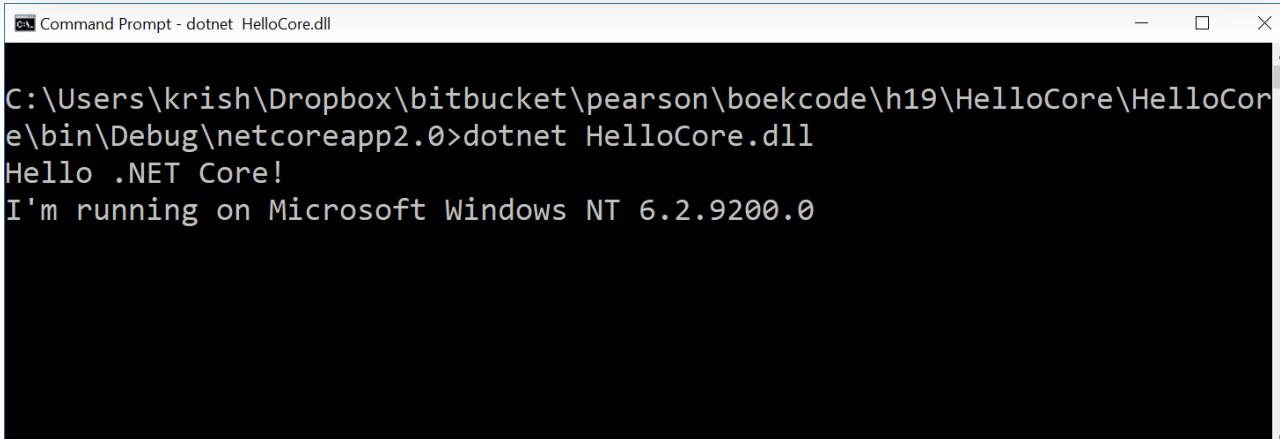
- In plaats van “Windows Classic Desktop” kan je ook kiezen voor “.NET Core”
- .NET Core is een nieuwe / verbeterde versie van het .NET Framework
- Applicatietypes:
  - Console programma's
  - Webtoepassingen
  - ...
  - (nog) geen WPF

# .NET Core

- Kan op verschillende OS draaien



```
netcore — dotnet HelloCore.dll — 80x8
[[kris@imac ~/Dropbox/netcore]$ dotnet HelloCore.dll
Hello .NET Core!
I'm running on Unix 16.7.0.0]
```



```
Command Prompt - dotnet HelloCore.dll
C:\Users\krish\Dropbox\bitbucket\pearson\boekcode\h19\HelloCore\HelloCore\bin\Debug\netcoreapp2.0>dotnet HelloCore.dll
Hello .NET Core!
I'm running on Microsoft Windows NT 6.2.9200.0
```

# Objectgeoriënteerd ontwerp

Hoofdstuk 20

# Inleiding

- De oefeningen tot hertoe: “*trial and error*” programmeren
  - Onmiddellijk programmeren
  - Geen overzicht over het geheel
  - Werkt voor kleine programma’s
  - Moeilijk te verdelen over verschillende programmeurs

# Ontwikkelmethodes

- Beschrijven een proces voor het ontwerpen en schrijven van programma's
- UML is *geen* proces, enkel een hulpmiddel (tekeningen)
- Voorbeelden:
  - Watervalmodel, spiraalmodel
  - Rational Unified Process (RUP)
  - eXtreme Programming (XP)
  - Agile programming

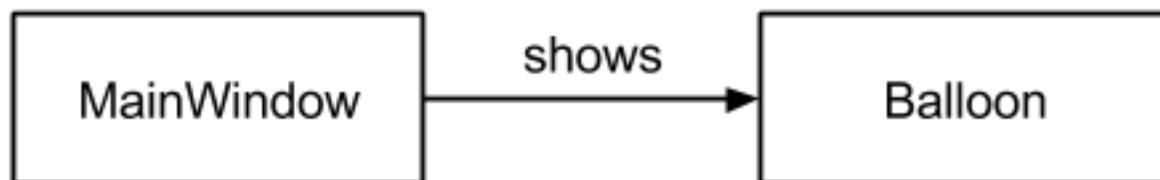
# Het ontwerpprobleem

- Welke klassen hebben we nodig in ons programma? Hoe stellen we die vast?
  - Zoveel mogelijk de objecten “*uit de realiteit*” beschouwen
  - Abstractie: irrelevante details weglaten
- Merk op: dit is meestal een eerste aanzet, extra (hulp)klassen zullen eveneens nodig zijn.

# Het vaststellen van objecten, methoden en properties

*Schrijf een programma dat een ballon representeert en zorg ervoor dat de ballon via een GUI te bewerken is. De ballon wordt getoond als een cirkel in een Canvas. Met behulp van knoppen kan de plaats van de ballon worden veranderd door deze een vaste afstand naar boven of naar beneden te laten bewegen. Met behulp van een schuifregelaar kan de straal van de ballon worden veranderd. De straal wordt getoond op een label.*

- Deze specificatie is reeds zeer specifiek, met een gedetailleerde GUI beschrijving. In de praktijk ontbreekt dit vaak.
- Niet elk zelfstandig naamwoord is een klasse. Vaak komen ook klassen voor die niet in de spec staan.
- →Veel oefenen en samenwerken met ervaren ontwerpers

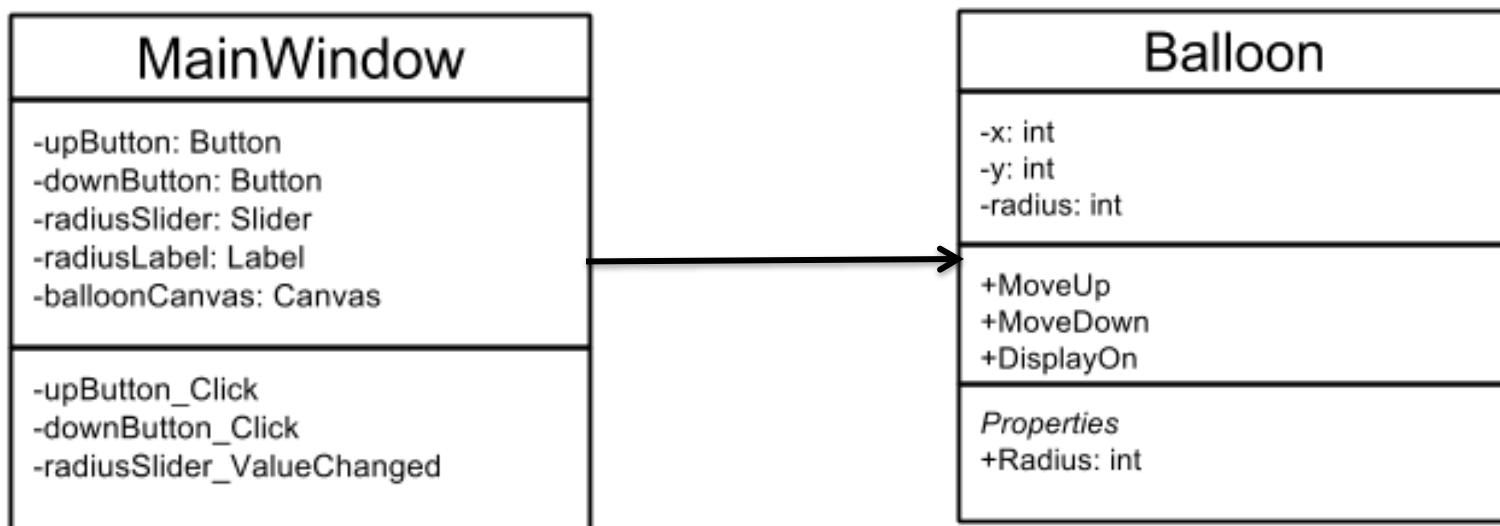


# Het vaststellen van objecten, methoden en properties

*Schrijf een programma dat een ballon representeert en zorg ervoor dat de ballon via een GUI te bewerken is. De ballon wordt getoond als een cirkel in een Canvas. Met behulp van knoppen kan de plaats van de ballon worden veranderd door deze een vaste afstand naar boven of naar beneden te laten bewegen. Met behulp van een schuifregelaar kan de straal van de ballon worden veranderd. De straal wordt getoond op een label.*

- Werkwoorden, acties, ... bepalen de methodes
- Eigenschappen, toestanden, ... bepalen properties
- Soms is er keuze tussen een property en een methode

# UML diagram



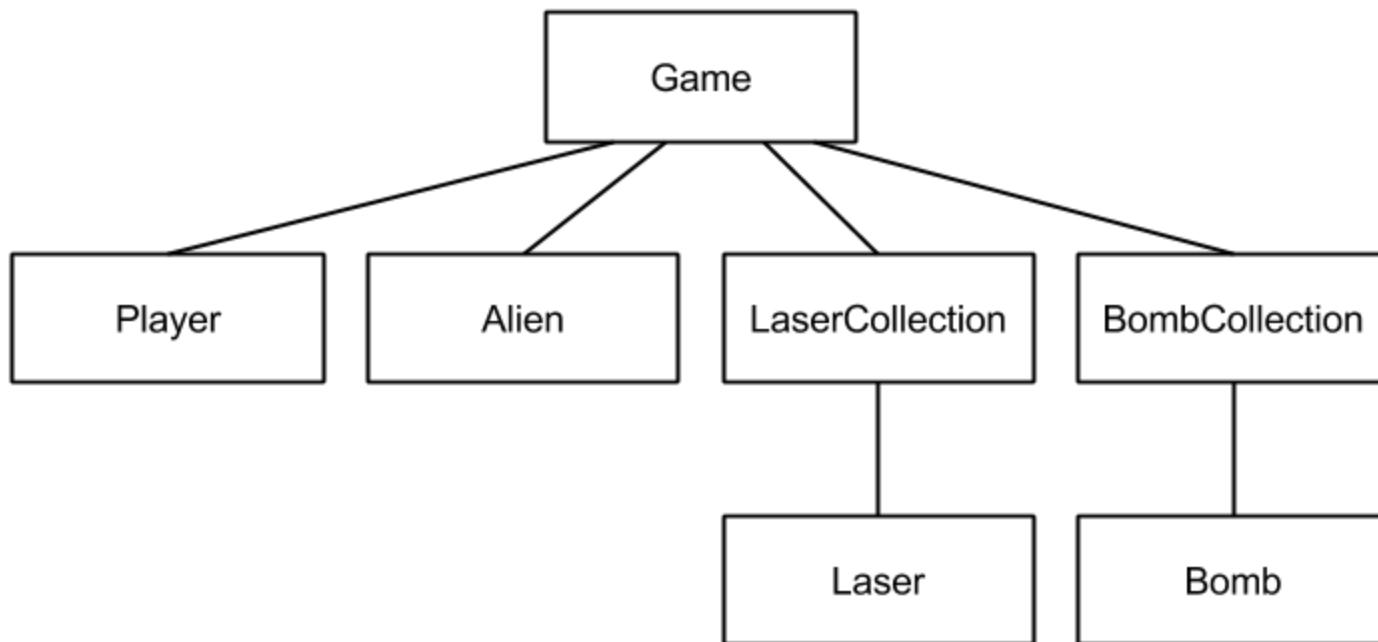
# Case study: Cyberspace Invader

*Het programma toont een speler en een alien. De alien beweegt zijwaarts. Wanneer de alien een muur raakt, keert deze van richting om. De alien gooit periodiek een bom die verticaal naar beneden beweegt. Als een bom de speler raakt, verliest de speler.*

*De speler beweegt naar links of naar rechts door overeenkomstige muisbewegingen. Bij een muisklik vuurt de verdediger een laserstraal af, die naar boven beweegt. Wanneer de alien geraakt wordt door een laserstraal wint de speler.*



# Klassen ontdekken



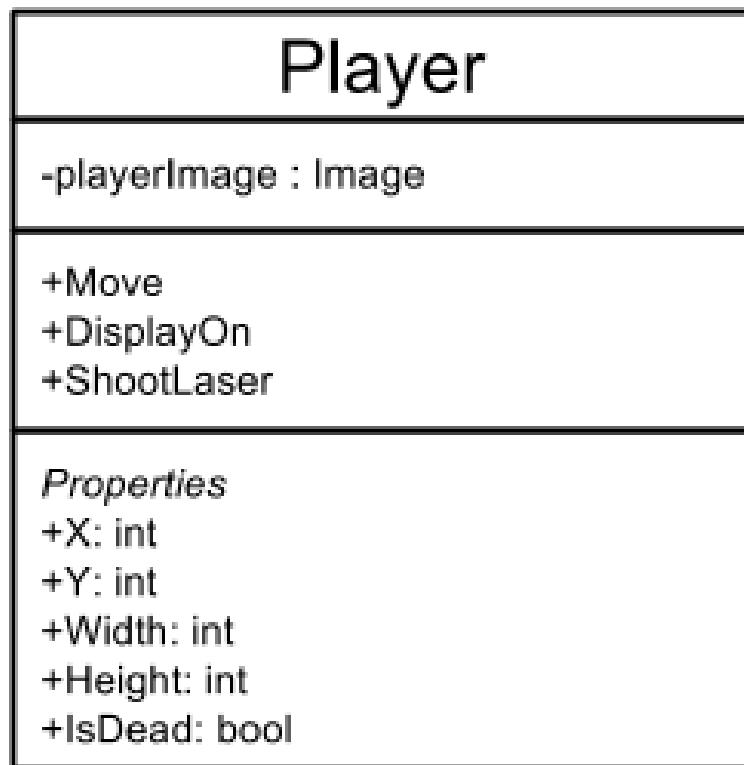
# De klasse Game

## Game

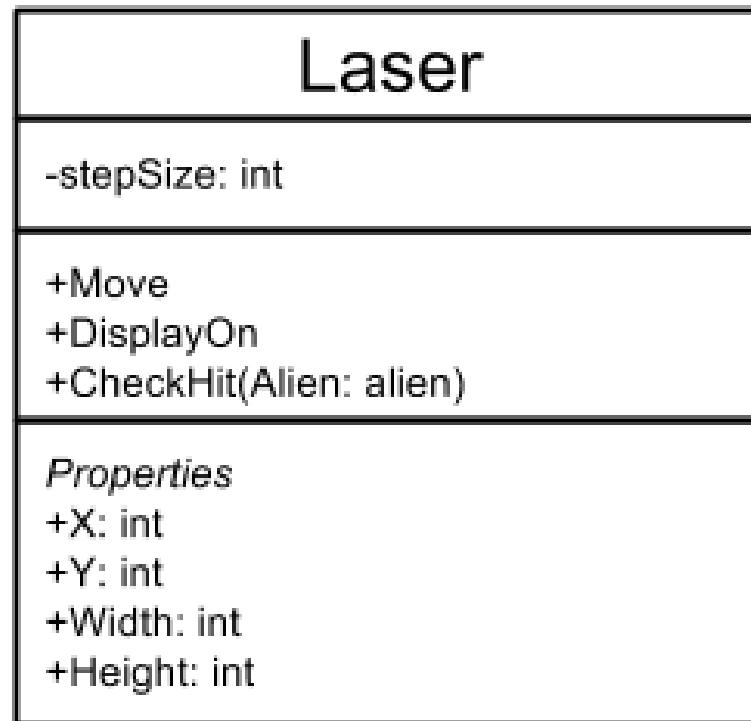
- gameCanvas: Canvas
- animationTimer: DispatcherTimer
- bombTimer: DispatcherTimer

- bombTimer\_Tick
- animationTimer\_Tick
- gameCanvas\_MouseUp
- gameCanvas\_MouseMove

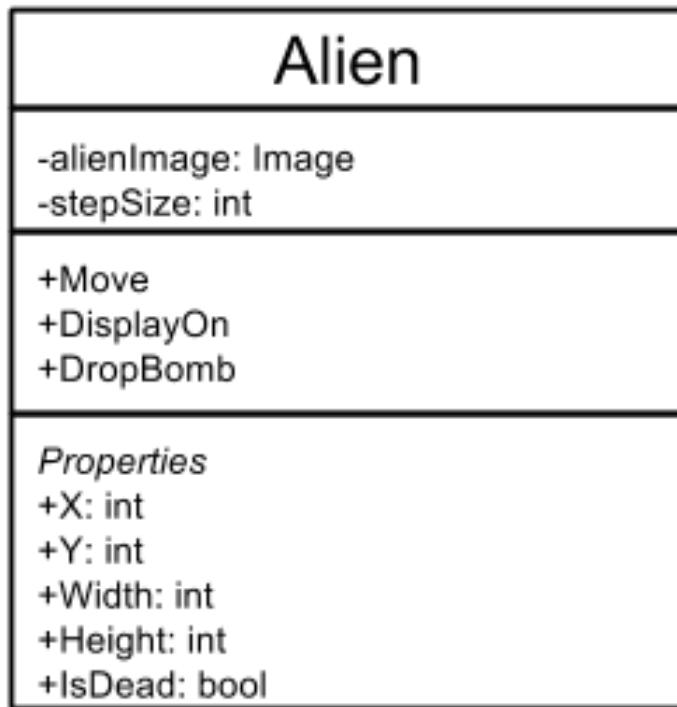
# De klasse Player



# De klasse Laser

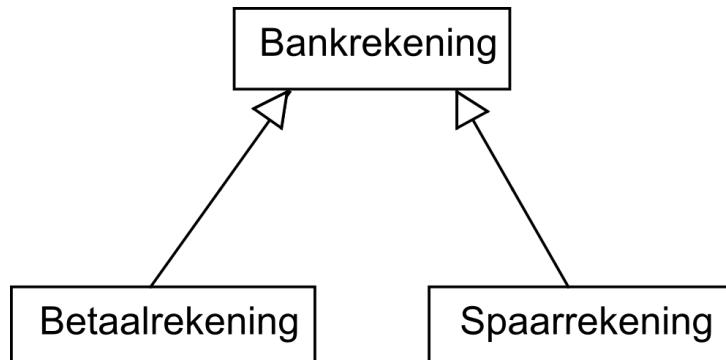
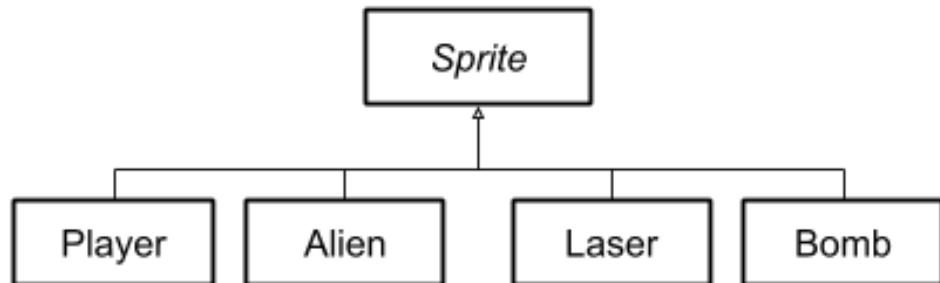


# De klasse Alien



# Relaties

- Compositie
  - Heeft-een, Maakt-een, Bestaat-uit
- Overerving
  - Is-een



# Richtlijnen

- Houdt gegevens **private**
  - Toegang via properties of methodes
- Initialiseer gegevens
  - In de declaratie of constructor
- Vermijd grote klassen
- Kies veelzeggende namen
- Forceer geen overerving
- Bij overerving gedeelde items in de superklasse zetten
- Gebruik refactoring om je ontwerp te verbeteren

# Programmeerstijl

Hoofdstuk 21

# Inleiding

- Belang van een goede programmeerstijl:
  - Programma's worden door meerdere mensen gemaakt
    - Onderlinge afspraken maken
    - Leesbaarheid (irritatie vermijden)
    - Consistentie, uniforme code
  - Hergebruik van code bevorderen
    - Objecten
    - Componenten

# Programmalay-out

- Namen
  - Betekenisvol
  - Begint met Hoofdletter: klasse, methode, property en constante
  - Begint met kleine letter: keyword, parameter, lokale variabele, instantievariabele, naam van besturingselement
  - Kies een naam voor besturingselementen zodanig dat die de functie van het besturingselement aangeeft, zoals calculateButton, ageTextBox, resultLabel
  - *Geen set/get in de naam van een property*

# Programmalay-out

- **Inspringen**
  - In principe niet nodig, maar erg aangeraden
  - IDE helpt hierbij
  - Lange regels opsplitsen (best niet te veel functionaliteit in 1 regel stoppen, ook al is de regel opgesplitst)
- **Witregels**
  - Scheiden van methoden, member variabelen, properties
- **1 klasse per bestand**
- **#region ... #endregion voor het scheiden van blokken code**

# Commentaar

- Niet herhalen wat code al duidelijk maakt
- Duidelijke code behoeft weinig tot geen commentaar
- Wees spaarzaam
- Wel belangrijk: API documentatie
  - Wat betekent de klasse
  - Wat doen public methoden en properties

# Constanten

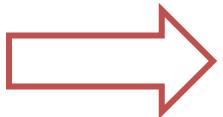
- Verhogen leesbaarheid
- Sommigen prefereren hoofdletters  
`const int MAXIMUM_SIZE = 10;`
- Wij verkiezen Pascal casing  
`const int MaximumSize = 10;`

# Klassen

- Bevorderen hergebruik, flexibiliteit
- Lengte van een klasse beperken
- Lengte van een methode beperken
- Inkapseling
- Naamgeving
- Volgorde:
  - Instantievariabelen
  - Properties
  - Constructors
  - Public methoden
  - Private methoden

# Geneste ifs

```
if (a > b)
{
    if (a > c)
    {
        largest = a;
    }
    else
    {
        largest = c;
    }
}
else
{
    if (b > c)
    {
        largest = b;
    }
    else
    {
        largest = c;
    }
}
```



```
if ((a >= b) && (a >= c))
{
    largest = a;
}
else if ((b >= a) && (b >= c))
{
    largest = b;
}
else
{
    largest = c;
}
```

# Geneste lussen

```
y = 5;  
for (int floor = 0; floor < numberOffloors; floor++)  
{  
    x = 5;  
    for (int flat = 0; flat < numberOfflats; flat++)  
    {  
        DrawRectangle(apartmentsCanvas, brush, x, y, 20, 10);  
        x = x + 25;  
    }  
    y = y + 20;  
}
```

Aparte methode is  
duidelijker

# Geneste lussen

```
for (int floor = 0; floor < floors; floor++)  
{  
    DrawFloor(flats, y, brush);  
    y = y + 20;  
}
```

```
y = 5;  
for (int floor = 0; floor < numberOffloors; floor++)  
{  
    DrawFloor(numberOfflats, y, brush);  
    y = y + 20;  
}
```

# Ingewikkelde voorwaarden

```
const int MaximumSize = 100;
int[] table = new int[MaximumSize];

int wanted = Convert.ToInt32(wantedTextBox.Text);
int index = 0;
while ((index < MaximumSize) && (table[index] != wanted))
{
    index++;
}

if (table[index] == wanted)
{
    resultTextBox.Text = "Found";
}
else
{
    resultTextBox.Text = "Not found";
}
```

# Ingewikkelde voorwaarden

```
const int MaximumSize = 100;
int[] table = new int[MaximumSize];

int wanted = Convert.ToInt32(wantedTextBox.Text);

const int StillSearching = 0;
const int Found = 1;
const int NotFound = 2;
int state = StillSearching;
```

# Ingewikkelde voorwaarden

```
int index = 0;
while (state == StillSearching)
{
    if (table[index] == wanted)
    {
        state = Found;
    }
    else if (index + 1 == MaximumSize)
    {
        state = NotFound;
    }
    index++;
}

if (state == Found)
{
    resultTextBox.Text = "Found";
}
else
{
    resultTextBox.Text = "Not found";
}
```

# Ingewikkelde voorwaarden

```
enum SearchState
{
    StillSearching = 0,
    Found = 1,
    NotFound = 2
}
```

```
const int MaximumSize = 100;
int[] table = new int[MaximumSize];

int wanted = Convert.ToInt32(wantedTextBox.Text);

SearchState state = SearchState.StillSearching;

int index = 0;
while (state == SearchState.StillSearching)
{
    if (table[index] == wanted)
    {
        state = SearchState.Found;
    }
    else if (index + 1 == MaximumSize)
    {
        state = SearchState.NotFound;
    }
    index++;
}

if (state == SearchState.Found)
{
    resultTextBox.Text = "Found";
}
else
{
    resultTextBox.Text = "Not found";
}
```

# Documentatie

- Programmaspecificatie
  - Wat moet het programma doen?
- Screenshots
- Broncode met commentaar
- Ontwerp (UML diagrammen)
- Testen
- Geschiedenis van alle wijzigingen
- Gebruikershandleiding

# Testen



*Niet kennen: blz. 529 - 538*

# Interfaces

Hoofdstuk 22

# In dit hoofdstuk ...

- Interfaces gebruiken voor de structuur van een programma
- Onderlinge aansluiting
- Interfaces versus abstracte klassen

# Inleiding

- Een interface beschrijft de uiterlijke verschijningsvorm van een klasse
  - *Iemand enig idee wat dit is?*
- Niet te verwarren met grafische user interface
- Doel:
  - Ondersteunen van het ontwerp
  - Bevorderen van onderlinge aansluiting  
(Een veelgebruikte term is “*loose coupling*”)
- Het concept interface bestaat eveneens in Java

# Interfaces voor ontwerp

```
public interface IBalloon
{
    void ChangeSize(int diameter);
    void DisplayOn(Canvas drawArea);
    int X { get; set; }
}
```

- Geen `class` sleutelwoord
- Geen access modifiers bij methoden en properties
- Dit is puur een beschrijving van diensten: uiterlijke verschijningsvorm van een klasse
- Het bevat *geen* implementatie!
- `get`: mogelijkheid tot lezen, `set`: mogelijkheid tot schrijven  
→ dit zijn ook *geen* implementaties!

# Interfaces voor ontwerp

```
public class Balloon : IBalloon
{
    private int diameter;
    private int x, y;
    private Ellipse ellipse;

    public Balloon()
    {
        CreateEllipse();
    }

    public void ChangeSize(int diameter)
    {
        this.diameter = diameter;
        UpdateEllipse();
    }
}
```

Een klasse kan één of meerdere interfaces implementeren

De klasse is **verplicht** om alle methodes en properties uit de interface te implementeren

# Interfaces voor ontwerp

```
public void DisplayOn(Canvas drawArea)
{
    drawArea.Children.Add(ellipse);
}

public int X
{
    get { return x; }
    set { x = value; UpdateEllipse(); }
}

private void UpdateEllipse()
{...}

private void CreateEllipse()
{...}

}
```

De klasse is **verplicht** om alle methodes en properties uit de interface te implementeren

# Interfaces voor ontwerp

- Interfaces kunnen ook overerven van andere interfaces

```
public interface IColoredBalloon : IBalloon
{
    void SetColor(Color c);
}
```

# Interfaces voor ontwerp

- Interfaces worden vaak eerst uitgeschreven aan de hand van een analyse
- Vervolgens worden de interfaces verdeeld onder de programmeurs en geïmplementeerd
- Voorbeeld:

*Stel Jan moet interface `IBalloon` implementeren en Mieke moet `IGame` implementeren. Bovendien heeft Mieke de implementatie van `IBalloon` nodig. Omdat je werkt met interfaces hoeft Mieke niet te wachten en kan ze onmiddellijk starten met de implementatie van `IGame`, eventueel gebruik makend van een “Mock” implementatie van `IBalloon`. Een “Mock” implementatie is een zeer eenvoudige implementatie die weinig of geen functionaliteit bevat, maar louter als bedoeling heeft testen of verdere implementaties te bewerkstelligen.*

# Onderlinge aansluiting

- Het afspreken van standaarden zorgt ervoor dat onderlinge uitwisselbaarheid verbetert, bijvoorbeeld:
  - Stopcontacten
  - Audio- en videoaansluitingen
  - DVD standaarden
  - ...
- Het concept interface brengt deze voordelen naar de softwarewereld
- Doordat klassen niet meer rechtstreeks van elkaar afhangen, maar wel van interfaces, spreekt men in dit verband ook wel van “loose coupling”

# Voorbeeld

```
public interface IDisplayable
{
    void DisplayOn(Canvas drawArea);
}
```

Alle klassen die moeten getekend kunnen worden moeten deze interface implementeren

```
public class Square : IDisplayable
{
    private int x, y;
    private int size;
    private Rectangle rectangle;

    public void DisplayOn(Canvas drawArea)
    {
        drawArea.Children.Add(rectangle);
    }

    // other methods and properties
}
```

# Voorbeeld

Het grote voordeel is dat het gedeelte van het programma dat vormen moet tekenen kan gebruik maken van polymorfie (en dus enkel op de hoogte moet zijn van de interface)

```
public class DrawingProgram
{
    private List<IDisplayable> shapeCollection = new List<IDisplayable>();
    ...
    public void UpdateScreen()
    {
        foreach (IDisplayable shape in shapeCollection)
        {
            shape.DisplayOn(myArea);
        }
    }
    ...
}
```

# Toepassing van interfaces: IList

- Een ListBox control heeft achter de schermen een klasse die de items beheert
- Deze klasse is een implementatie van een interface IList

# Toepassing van interfaces: IList

```
public interface IList
    : ICollection, IEnumerable
{
    int Add(object value);
    void Clear();
    bool Contains(object value);
    int IndexOf(object value);
    void Insert(int index, object value);
    void Remove(object value);
    void RemoveAt(int index);

    bool IsFixedSize { get; }
    bool IsReadOnly { get; }
    object Item(int index) { get; set; }
}
```

# Verschil met abstracte klassen

1. Een abstracte klasse kan voor sommige methodes een implementatie voorzien, een interface kan dit nooit
2. Een klasse kan meerdere interfaces implementeren, maar slechts van één (abstracte) klasse overerven
  - Om over te erven hoeft de parent klasse niet abstract te zijn!
3. Een interface zorgt al tijdens compilatie voor controle, bij een abstracte klasse wordt er pas tijdens de uitvoering gekoppeld met een concrete methode/property
4. Een interface alleen wil nog niet zeggen dat deze zal gebruikt worden voor overerving

# Polymorfie

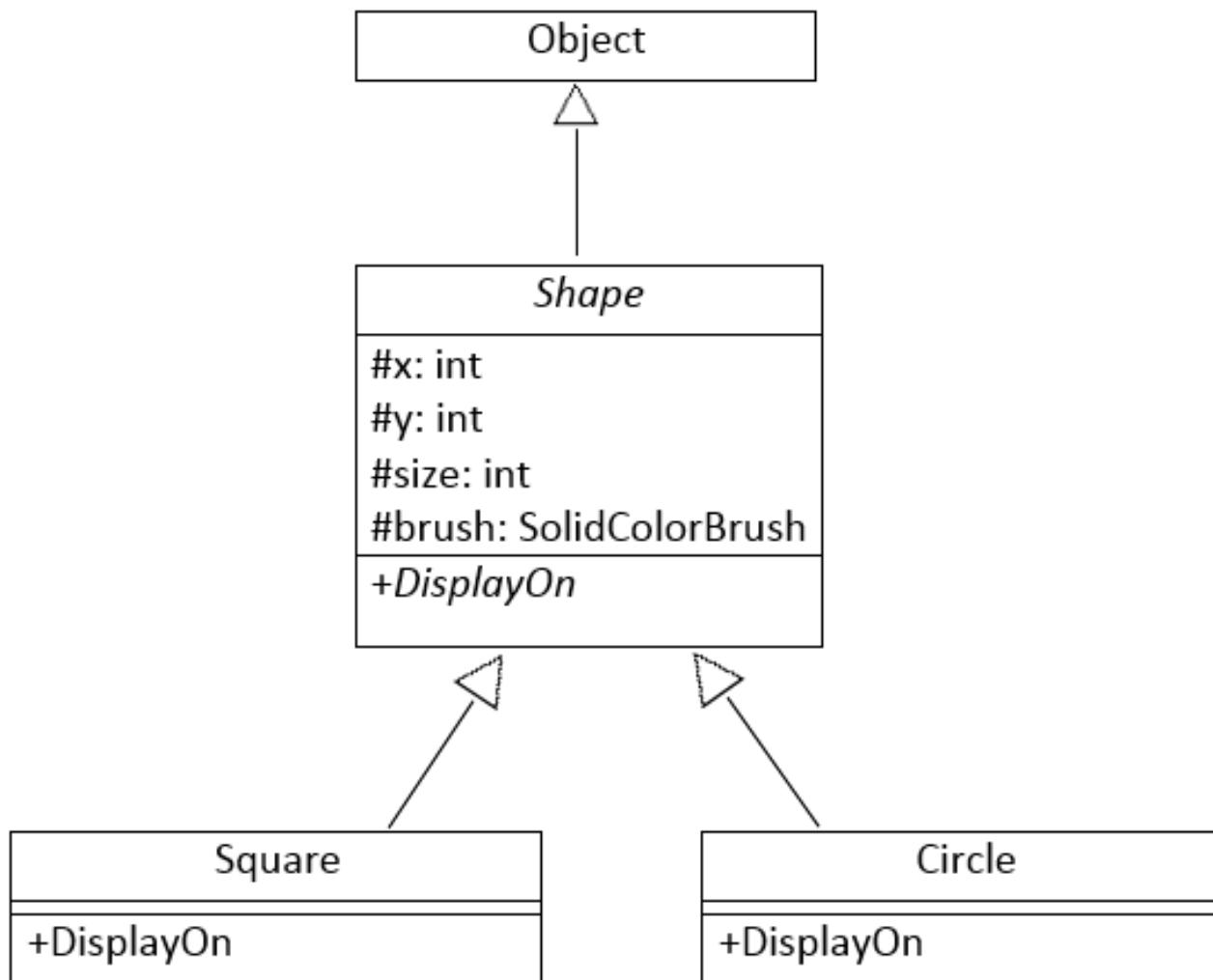
Hoofdstuk 23



# Inleiding

- Polymorfie zorgt ervoor dat een gegeven object verschillende “vormen” kan aannemen (naargelang de interfaces die zijn geïmplementeerd en/of de klasse waarvan is overgeërfd)
- We illustreren aan de hand van een voorbeeld

# Polymorfie in actie



# Polymorfie in actie

```
private void shapesCanvas_MouseUp(object sender,  
                                  MouseButtonEventArgs e)  
{  
    Circle circle1 = new Circle(40, 40);  
    Square square1 = new Square(180, 180);  
    List<Shape> shapes = new List<Shape>();  
  
    shapes.Add(circle1);  
    shapes.Add(square1);  
  
    foreach (Shape shape in shapes)  
    {  
        shape.DisplayOn(shapesCanvas);  
    }  
}
```

Deze lijst kan objecten bevatten die afstammen van Shape

Afhankelijk van het concrete object (klasse) tijdens de uitvoering, wordt de juiste methode opgeroepen. = **Dynamic Dispatch**

# Opmerking

- Omwille van Genericiteit wordt casten overbodig

```
ArrayList shapes = new ArrayList();
```

```
group.Add(circle1);  
group.Add(square1);
```

Niet generische lijst die objecten opslaat zonder hun klasse informatie bij te houden (object)

```
foreach (object obj in shapes)
```

```
{
```

```
    Shape shape = (Shape) obj;  
    shape.DisplayOn(shapesCanvas);
```

```
}
```

Eerst casten naar Shape vooraleer DisplayOn beschikbaar is.