Device Programming

Tuur Vanhoutte

December 13, 2020

Contents

1 .NET **1**
- 1.1 Languages . . . 1
- 1.2 Applications . . . 1
  - 1.2.1 Desktop . . . 1
  - 1.2.2 Web & Server . . . 2
  - 1.2.3 Mobile . . . 2
  - 1.2.4 Gaming . . . 2
  - 1.2.5 IoT . . . 2
  - 1.2.6 AI . . . 2
- 1.3 Xamarin . . . 2
  - 1.3.1 Xamarin - UI Technology . . . 3
  - 1.3.2 Xamarin - Code Sharing strategy . . . 3
- 1.4 Summary . . . 3

2 C# Syntax **4**
- 2.1 Python vs C# (Summary) . . . 4
- 2.2 Datatypes . . . 4
- 2.3 Collections . . . 4
  - 2.3.1 Arrays . . . 4
  - 2.3.2 Dictionary <TKey, TValue> . . . 5
  - 2.3.3 List<T> . . . 5
- 2.4 Selections . . . 5
- 2.5 Loops . . . 6
- 2.6 Classes . . . 6
- 2.7 Instantiate objects . . . 7
- 2.8 Properties . . . 7
  - 2.8.1 Fields vs properties . . . 7
  - 2.8.2 Default values for properties . . . 7
- 2.9 Constructor . . . 7

3 Streamreader **8**
- 3.1 Namespaces . . . 8
- 3.2 System.Reflection . . . 8
- 3.3 System.IO . . . 9
- 3.4 Embedded files . . . 9
  - 3.4.1 Read an embedded file in Xamarin (using Reflection) . . . 10
  - 3.4.2 Processing the file's content . . . 10
- 3.5 Summary . . . 10

4 Navigation **10**
- 4.1 Modal vs Modeless . . . 10
- 4.2 Navigate forward . . . 11
- 4.3 Navigate back . . . 11
  - 4.3.1 Go back - Modeless page . . . 11
  - 4.3.2 Go back - Modal page . . . 12
- 4.4 Navigation stack . . . 12
- 4.5 Page types . . . 13
- 4.6 Exchanging data . . . 13
- 4.7 Summary . . . 13

5 Object Orientation **14**
5.1 Inheritance . . . 14
5.1.1 Constructor . . . 14
5.1.2 Access modifiers . . . 15
5.1.3 Properties/methods: VIRTUAL and OVERRIDE . . . 15
5.1.4 Properties/methods: ABSTRACT and OVERRIDE . . . 16
5.2 Polymorphisme . . . 17
5.2.1 Disadvantage . . . 18
5.3 Interfaces . . . 18
5.3.1 Summary . . . 19
5.3.2 Example . . . 19
5.4 Composition vs Aggregation . . . 19
5.4.1 Example . . . 20
5.4.2 Interfaces in Xamarin(.Forms) . . . 20
5.5 Summary . . . 20

6 Asynchronous programming: async and await keywords **21**
6.1 Why? . . . 21
6.2 Example: newspaper app . . . 21
6.2.1 Problem . . . 21
6.2.2 Solution . . . 22
6.3 Creating an asynchronous function . . . 22
6.3.1 With void-functions: use Task . . . 22
6.3.2 With a function that returns a type T . . . 23
6.4 Calling an asynchronous function . . . 23
6.5 Summary . . . 23

7 XAML in Xamarin.Forms **24**
7.1 Benefits . . . 24
7.2 Pages . . . 24
7.2.1 Adding a XAML Page . . . 24
7.3 Steps to build a UI with XAML . . . 24
7.3.1 Describing a screen . . . 24
7.3.2 C# code with XAML . . . 25
7.3.3 XAML Initialization . . . 26
7.3.4 Naming elements in XAML . . . 26
7.3.5 Handling events in XAML . . . 26
7.3.6 Handling events in the code behind . . . 27
7.4 Layout in Xamarin.Forms . . . 27
7.4.1 Motivation . . . 27
7.4.2 Layouts . . . 27
7.4.3 Views . . . 27
7.4.4 Working with sizes . . . 28
7.4.5 Layout algorithm . . . 28
7.4.6 Default view sizing . . . 28
7.4.7 View preferences . . . 28
7.4.8 Sizing requests . . . 29
7.4.9 Size units . . . 29
7.4.10 Platform rendering . . . 29
7.4.11 Alignment . . . 29
7.4.12 Fill . . . 30
7.5 Margin & Padding . . . 30

7.5.1 Margin . . . 30
7.5.2 Padding . . . 30
7.6 Arrange views with StackLayout . . . 31
7.6.1 Adding children . . . 31
7.6.2 Child ordering . . . 31
7.6.3 Child spacing . . . 31
7.6.4 Orientation . . . 32
7.7 Expansion . . . 33
7.7.1 Expansion direction . . . 33
7.7.2 How much extra space? . . . 33
7.7.3 How to specify expansion? . . . 33
7.7.4 Expansion vs view size . . . 34
7.8 Apply attached properties . . . 34
7.8.1 Motivation . . . 34
7.8.2 What is an attached property? . . . 34
7.8.3 Who consumes attached properties? . . . 35
7.8.4 Apply an attached property in code . . . 35
7.8.5 Apply an attached property in XAML . . . 35
7.9 Grids . . . 35
7.9.1 Grid rows/columns . . . 36
7.9.2 GridLength . . . 36
7.9.3 Grid example . . . 37
7.9.4 Default size . . . 37
7.9.5 Row/Column numbering . . . 38
7.9.6 Grid positioning properties . . . 38
7.9.7 Cell specification . . . 38
7.9.8 Span specification . . . 39
7.9.9 Cell and span defaults . . . 39
7.9.10 Layout options . . . 39
7.9.11 Grid child spacing . . . 40
7.9.12 Auto-generated rows/columns . . . 40
7.10 Scroll a layout with ScrollView . . . 40
7.10.1 How to use ScrollView . . . 40
7.10.2 ScrollView orientation . . . 41
7.10.3 Do not nest scrolling views . . . 41
7.11 Summary . . . 41

1 .NET

.NET is a free, cross-platform, open source developer platform (*) for building many different types of applications.

* = languages + libraries

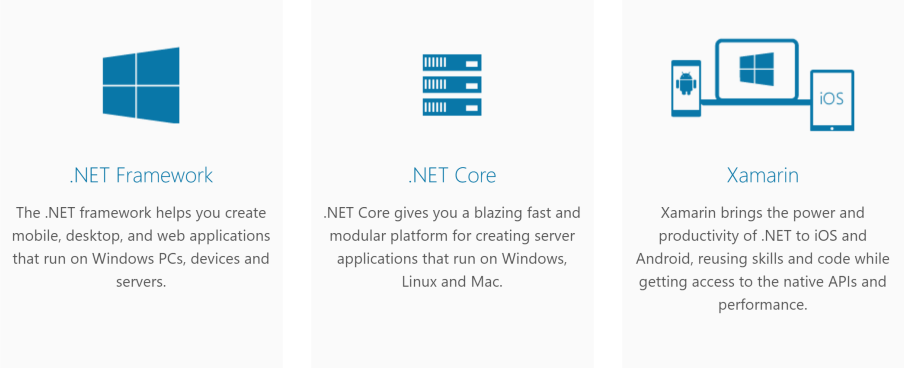


Figure 1: .NET ecosystem

1.1 Languages

- Syntax very similar to C, C++, Java & JavaScript
- Functional programming language, cross-platform, open source
- Approachable English-like language for Object Oriented Programming (OOP)

1.2 Applications

- Desktop
- Web & Server
- Mobile
- Gaming
- IoT
- AI

1.2.1 Desktop

- UWP (Universal Windows Project)
- Xamarin.Mac
- WPF (Windows Presentation Foundation)
- WinForms (Windows Forms)

1.2.2 Web & Server

- ASP.NET
- ASP.NET Core

1.2.3 Mobile

- UWP (Universal Windows Project)
- Xamarin

1.2.4 Gaming

- Unity
- CryEngine

1.2.5 IoT

- UWP
- .NET Core IoT

1.2.6 AI

- Cognitive Services
- Azure Machine Learning
- Machine Learning and AI Libraries
- F# for Data Science and ML

1.3 Xamarin

- 'Target all platforms with a single, shared codebase for Android, iOS, Windows'.
- Developen van Mobile devices lastig: verschillende platformen, verschillende talen voor elk device.
- **Oplossing:** Xamarin
- Extensie op Visual Studio.

Figure 2: Xamarin Logo

1.3.1 Xamarin - UI Technology

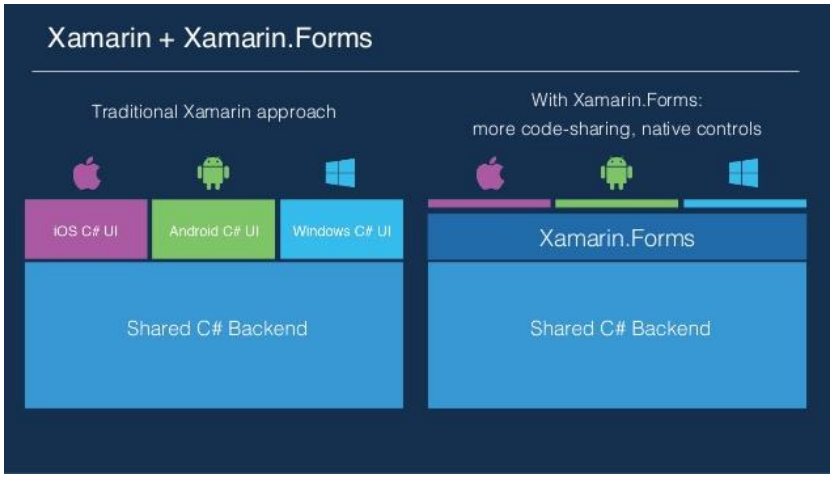


Figure 3: Native vs Xamarin.Forms

1.3.2 Xamarin - Code Sharing strategy

Figure 4: .NET Standard (links) vs Shared Assets Project (rechts)

Met Shared Assets Project maken we de UI voor elk platform apart. Wij gaan vooral werken met .NET Standard.

1.4 Summary

- What devices, platforms, etc. can we target using .NET, and what programming languages can we use?
- What is the basic difference between .NET Standard and Shared Assets projects in Xamarin?
- What is the difference between Xamarin native and Xamarin.Forms? What are the advantages and disadvantages?
- How to set up and understand the structure of a Xamarin project for the labs in this course, and how to debug on the different platforms.

2 C# Syntax

2.1 Python vs C# (Summary)

- Curly brackets { } in plaats van indenting
- C# = statically typed language: datatype is needed when declaring variables
  - int Number = 5;
  - string Name = "Tuur";
- Python = dynamically typed language: datatype not needed
- C# has more ways to create collections (Array, Dictionary, List)

2.2 Datatypes

Type	Omschrijving	Waarde	
Gehele getallen		**Minimum**	**Maximum**
int	integer	-2^{31}	2^{31}
long	long integer	-2^{63}	2^{63}
Reële getallen			
float	Kommagetal (positief / negatief)	$1{,}5 \times 10^{-45}$	$3{,}4 \times 10^{38}$
double	Preciezer kommagetal (positief / negatief)	5×10^{-324}	$1{,}7 \times 10^{308}$
decimal	Geldbedragen		
Tekst			
string	Tekenreeks		
Andere types			
char	1 teken		
bool	Booleaanse waarde	Onwaar (0)	Waar (1)

Figure 5: Datatypes in C#

2.3 Collections

- Array
- Dictionary<TKey, TValue>
- List<T>

Collection type = fixed! $\Rightarrow$ Je kan alleen objecten van het gekozen type toevoegen aan een collection

```
1 //collections of type Person:
2 Person[] teacherArr = new Person[10];
3 List<Person> teacherList = new List<Person>();
4
5 //You can only add Person objects to these collections!
```

2.3.1 Arrays

= meerdere variabelen van hetzelfde type

```
1 //initialize int array with 10 positions:
2 int[] numbers = new int[10];
3 //save number 13 in the first position
```

```
numbers[0] = 13;
//print the value of the first number in the array:
Debug.WriteLine("The first number is:" +numbers[0]);
//intialize and fill another array with 4 numbers:
int[] startPositions = { 4, 1, 9, 3 };
```

2.3.2 Dictionary <TKey, TValue>

```
//declare dictionary with key type & value type
Dictionary<string, int> studentScores = new Dictionary<string, int>();
//add two elements (key value pairs)
studentScores.Add("Jean-Jacques", 13);
studentScores.Add("Jean-Louis", 4);
//get the score of Jean-Jacques
int score = studentScores["Jean-Jacques"];
```

2.3.3 List<T>

```
//declare list, fill one by one:
List<string> emailList = new List<string>();
emailList.Add("stijn.walcarius@howest.be");
emailList.Add("frederik.waeyaert@howest.be");
//get elements out (two ways):
string first = emailList.ElementAt(0);
string second = emailList[1];
//declare + fill list:
List<string> teacherList = new List<string> { "SWC", "FWA" };
```

2.4 Selections

if / else if / else / switch

```
if(findTheoryTeacher == true) {
    email1 = "frederik.waeyaert@howest.be";
    email2 = null;
}
else if(findLabTeachers == true) {
    email1 = "stijn.walcarius@howest.be";
    email2 = " frederik.waeyaert@howest.be";
} else {
    email1 = email2 = null;
}
```

```
switch (teacher){
    case "SWC":
        email = "stijn.walcarius@howest.be";
        break;
    case "FWA":
        email = "frederik.waeyaert@howest.be";
        break;
```

```
    default:
        email = "info@howest.be";
        break;
}
```

2.5 Loops

for / foreach / while / do while

```
for(int i = 0; i < 100; i++) {
    //do something 100 times
}
```

```
List<string> teacherList = new List<string> { "SWC", "FWA" };
foreach(string teacher in teacherList) {
    //do something
}
```

```
while(endOfClass == false){
    //might never be executed
}
```

```
do {
    //executed at least once!
} while(endOfClass == false);
```

2.6 Classes

```
public class Person
{
    //property
    public string Name {
        get {...};
        set {...};
    }

    //constructor
    public Person(string name) {
        this.Name = name;
    }

    //method
    public void Subscribe() {
        //do something
    }
}
```

2.7 Instantiate objects

```
Persons p1 = new Person("Stijn");

// Based on the following constructor in the Person class:
public Person (string name) {
    this.Name = name;
}
```

2.8 Properties

2.8.1 Fields vs properties

- **Fields** store the actual data
- **Properties** are used to access those fields (getters & setters)
- Auto-implemented properties have a hidden field
- Use properties to control field access
- Enhance input/output control using get & set
- Calculated properties are built on other properties
  - No field required
  - Reusability

```
//private field
private int _id;

//property (zetten we altijd public)
public int Id {
    // getter
    get { return _id; }
    // setter
    set { _id = value; }
}
```

2.8.2 Default values for properties

- Setting default values can be useful
- Default values can be set...
  - ...with full properties
  - ...with auto-implemented properties
  - ...in the constructor

2.9 Constructor

- A constructor is called every time you create an instance of a class
- It is used to allow / force the user to provide certain values

- Default constructor is (only) added if a model has no constructors
- **Constructor overloading** = multiple constructors with either . . .
  - . . . a different number of parameters, or
  - . . . a different type of parameters, or
  - . . . the same parameters in a different order
- Constructors should call each other for enhanced efficiency
- Constructors in inheriting classes call the constructors of the base class

3 Streamreader

- Namespaces
- System.Reflection
- System.IO
- Embedded Files

3.1 Namespaces

Door 'using' te gebruiken kunnen we de namespace weglaten:

```
1 // Dan schrijven we:
2 using System.Diagnostics; // in het begin van het bestand
3 Debug.WriteLine("This is a debug message");
4 // In plaats van:
5 Debug.WriteLine("This is a debug message");
```

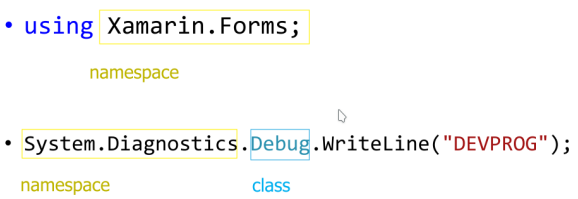


Figure 6: Namespaces

3.2 System.Reflection

*"The classes in the System.Reflection namespace, together with System.Type, enable you to obtain information about loaded assemblies and the types defined within, such as classes, interfaces, and value types. You can also use **reflection** to create type instances at run time and to invoke them."*

```
1 // Using GetType to obtain type information:
2 int i = 42;
3 Type type = i.GetType();
4 Console.WriteLine(type);
5 // Output: System.Int32
```

```
6
7  // Using Reflection to get information of an Assembly:
8  Assembly info = typeof(int).Assembly;
9  Console.WriteLine(info);
10 // Output: System.Private.CoreLib, Version=4.0.0.0, Culture=neutral,
11     PublicKeyToken=7cec85d7bea7798e
```

3.3 System.IO

= Input/Output `https://docs.microsoft.com/en-us/dotnet/api/system.io?view=net-5.0`

- StreamReader `https://developer.xamarin.com/api/type/System.IO.StreamReader/`
  - To read files
- StreamWriter `https://developer.xamarin.com/api/type/System.IO.StreamWriter/`
  - To write files

3.4 Embedded files

- Textfiles, images, etc.
- Generates a **ResourceID** for the file

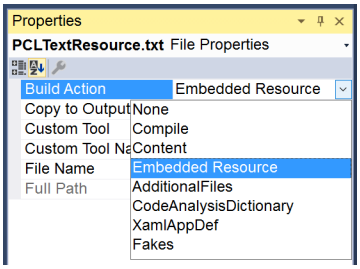


Figure 7: Embedded files inladen in een Visual Studio project: rechtermuisknop op 1 of meerdere files ⇒ properties ⇒ build action = Embedded resources

3.4.1 Read an embedded file in Xamarin (using Reflection)

```
var assembly = typeof(Foo).GetTypeInfo().Assembly;

string resourceID = "namespace_of_file.filename.csv";

Stream stream =
      assembly.GetManifestResourceStream(resourceID);

using (var reader = new System.IO.StreamReader(stream))
{
      //process file content                    .NET reflection
}                                                 namespace
                                                  System.IO
```

Figure 8: Reading a file using StreamReader

-

3.4.2 Processing the file's content

```
using (var reader = new System.IO.StreamReader(stream))
{
      reader.ReadLine(); //ignore title row
      string line = reader.ReadLine(); //read first line
      while (line != null)
      {
            //process line
            //...
            //read the next line
            line = reader.ReadLine();
      }
}
```

Figure 9: Processing the file's content

3.5 Summary

- You understand the importance of **namespaces**, and the techniques of using them in your own projects.
- You can explain the very basics of the **System.IO** and **System.Reflection** namespaces, and what they have to do with reading an embedded file in Xamarin.
- You understand the how and why of the **ResourceID** that's being generated for an embedded file.

4 Navigation

4.1 Modal vs Modeless

- Modal page: requires user input to continue

- Modeless page: user can go back any time he wants; no input required

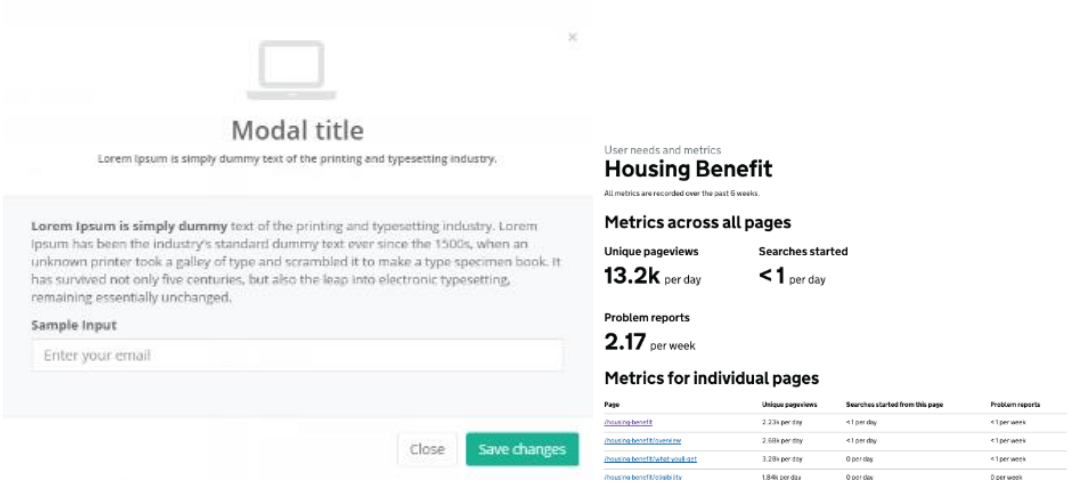


Figure 10: Modal page vs Modeless page

4.2 Navigate forward

```
1 Navigation.PushAsync(new FooPage());
2 Navigation.PushModalAsync(new FooPage());
3
4 // FooPage is hier de XAML page waar we willen naar navigeren
```

- PushAsync vs PushModalAsync
- Navigation object: controls the navigation stack

4.3 Navigate back

4.3.1 Go back - Modeless page

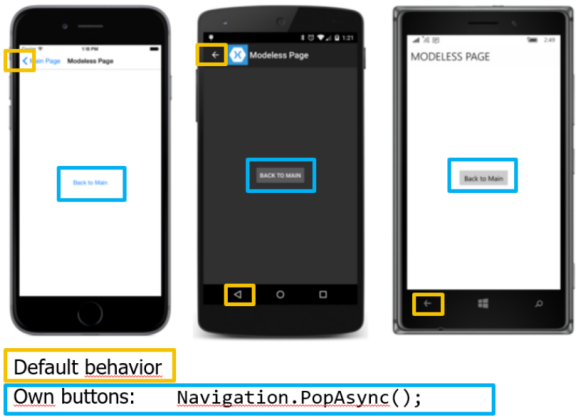


Figure 11

4.3.2 Go back - Modal page

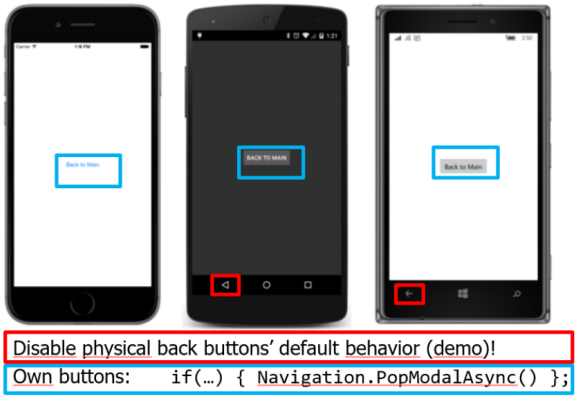


Figure 12

4.4 Navigation stack

Figure 13: Pushing to the stack

Figure 14: Pop-ing from the stack

Figure 15: InsertPageBefore

Figure 16: RemovePage

Figure 17: PopToRoot

4.5 Page types

- ContentPage
- MasterDetailPage (zie Demo_MasterDetail)
- NavigationPage (zie Demo_Navigation)
- TabbedPage (zie Demo_TabbedPage)
- TemplatedPage
- CarouselPage

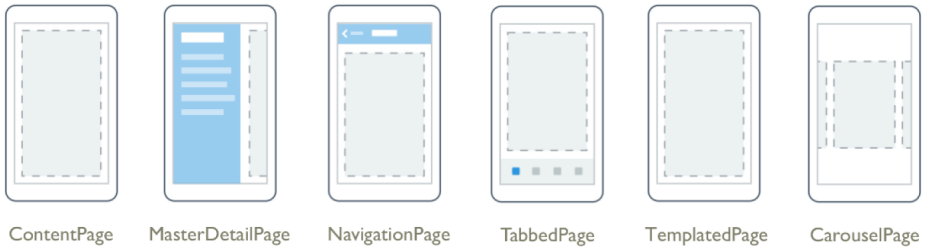


Figure 18: Xamarin's page types

4.6 Exchanging data

How to exchange data between several pages:

1. Constructor (Demo_MasterDetail)
2. Properties (Demo_TabbedPage)

4.7 Summary

- The different **page types** and how to use them.
- The difference between **Modal** and **Modeless** pages, and how to manage navigation for both.
- You know how to **exchange data** between pages in the navigation process.
- You understand the **navigation stack** and how you can manipulate it.
- You can explain the concept of a **master-detail** relation with an example

5 Object Orientation

5.1 Inheritance

= klasses nemen methods en properties over van een andere klasse.

Er ontstaat een hiërarchie.

```
// the base class:

public class Advisor
{
    // properties
    public string Name { get; set; }

    // methods
    public void Advise() { }
}

// the deriving class
// notice the : between the base class and deriving class

public class MinisterOfDefense : Adviser {
    // code here
}
```

- All C# classes, of any type, are treated as if they ultimately derive from System.Object

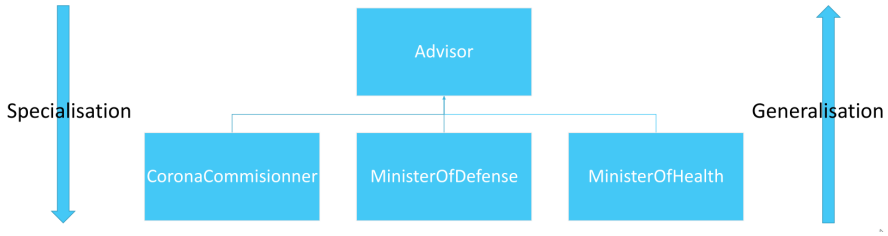


Figure 19: Specialisation & Generalisation

- **Generalize** properties (equal for all) by putting them in the **base class**
- **Specify** properties (specific for one) by putting them in the **deriving class**

5.1.1 Constructor

- Constructors are **not** inherited!
- Constructor without parameter in base class?
  - ⇒ Automatically called by deriving class
- No constructor without parameters in base class?
  - ⇒ Explicitly call it in deriving classes

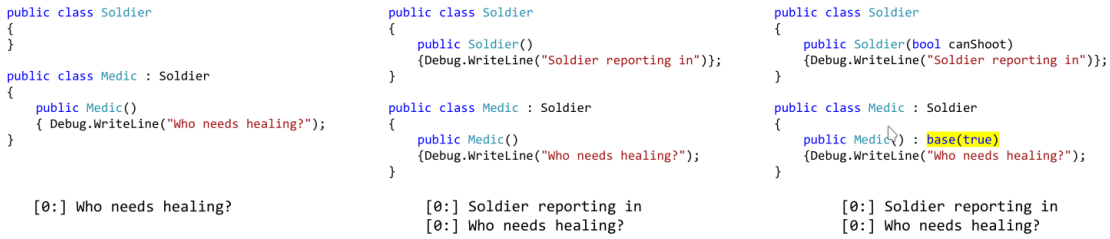


Figure 20: Inheritance: constructor example

5.1.2 Access modifiers

public is the default access modifier.

MODIFIER	APPLIES TO	DESCRIPTION
public	Any type or members	The item is visible to any other code
protected	Any member of a type, also the any nested type	The item is only visible in class and subclasses
private	Any type or members	The item is only visible in the class
internal	Any member of a type, also the any nested type	The item is only visible in it's containing assembly

Figure 21: Inheritance: access modifiers

5.1.3 Properties/methods: VIRTUAL and OVERRIDE

When you want to override a method from the base class, use the **virtual** keyword in the base method, and the **override** keyword in the derived method.

Virtual properties/methods:

- Default implementation in base class
- 'virtual' keyword, to **replace** the way an object behaves
- **CAN** be overriden in subclasses, only if necessary.

```
public class Vliegtuig
{
    public virtual void Vlieg()
    {
        Console.WriteLine("Het vliegtuig vliegt rustig door de wolken.");
    }
}

public class Raket : Vliegtuig
{
    public override void Vlieg()
    {
        Console.WriteLine("De raket verdwijnt in de ruimte.");
    }
}


Vliegtuig f1 = new Vliegtuig();
Raket spaceX1 = new Raket();
f1.Vlieg();
spaceX1.Vlieg();


[0:] Het vliegtuig vliegt rustig door de wolken.
[1:] De raket verdwijnt in de ruimte.
```

Figure 22: virtual and override

5.1.4 Properties/methods: ABSTRACT and OVERRIDE

Abstract properties/methods:

- No default implementation possible in base class
- 'abstract' keyword, to **extend** the way an object behaves
- **MUST** be present in each deriving class

```
public abstract class Animal
{
    public int Name { get; set; }
}

public class Horse : Animal
{
    //...
}

public class Wolf : Animal
{
    //..
}


Animal anAnimal = new Animal(); //ERROR

Wolf littleWolf = new Wolf();  //works fine
```

```
public abstract class Animal
{
    public abstract string MakeNoise();
}

public class Paard : Animal
{
    public override string MakeNoise()
    {
        return "Hinnikhinnik";
    }
}
```

Figure 23: abstract and override

5.2 Polymorphisme

= Objects of a derived class can be treated like objects of the base class at runtime

Example: say we have a class Animal, and two classes Cat and Dog that inherit the Animal class. Then, this is possible:

```
1 List<Animal> animals = new List<Animal>();
2 Animal dog = new Dog();
3 Animal cat = new Cat();
4
5 animals.Add(dog);
6 animals.Add(cat);
```

```
public abstract class Animal
{
    public abstract string MakeNoise();
}

public class Horse : Animal
{
    public override string MakeNoise()
    {
        return "Hinnikhinnik";
    }
}

public class Pig : Animal
{
    public override string MakeNoise()
    {
        return "Oinkoink";
    }
}


Animal someAnimal = new Pig();
Animal anotherAnimal = new Horse();
Debug.WriteLine(someAnimal.MakeNoise()); //Oinkoink
Debug.WriteLine(anotherAnimal.MakeNoise()); //Hinnikhinnik
```

Figure 24: Polymorphism example

5.2.1 Disadvantage

!!! **Multiple inheritance** is **NOT** allowed through classes in C# !!!

5.3 Interfaces

- Interfaces can be seen as contracts for classes
- Implementing = applying the contract
- An interface **forces** all implementing classes to implement **all** properties and/or methods
- An interface has no default implementation on its own (=you can't create an instance from an interface)

```
1  // You can't create an instance from an interface:
2  interface IAdvisor{
3      void Advise();
4  }
5
6  // This will create an error:
7  IAdvisor advisor = new IAdvisor();
```

5.3.1 Summary

- Contract + NO implementations = interface
- Contract + SOME implementations = abstract (base) class
- Implementation for all properties & methods = normal (base) class

5.3.2 Example

- The IAdvisor interface has an Advise() method (notice that method has no default implementation)
- Every class that implements this interface also must have its own Advice() method
- A class can implement multiple interfaces (see the MicrosoftCEO class)
- In the PrimeMinister class:
  - A list is created with the same type as the interface
  - Every element in that list is of a class that implements the interface
  - Can therefore also be used in a foreach() loop

```
public class MicrosoftCEO : CEO, IAdvisor
{

    public void Advise()
    {
        Console.WriteLine("I think you should allow our monopoly.");
    }

    public void EarnBigBucks()
    {
        Console.WriteLine("I'm getting rich!!!");
    }
    public void FireDepartement()
    {
        Console.WriteLine("You're all fired!");
    }
}

public class MinisterOfDefense : IAdvisor
{
    public void Advise() { }
}

public class MinisterofHealthcare : IAdvisor
{
    public void Advise() { }
}

public class CoronaCommissioner : IAdvisor
{
    public void Advise() { }
}
```

```
interface IAdvisor
{
    void Advise();
}
```

```
public class PrimeMinister
{
    // properties
    public string Name { get; set; }

    // public methods
    public void RunTheCountry()
    {
        List<IAdvisor> allAdvisors = new List<IAdvisor>();
        allAdvisors.Add(new MinisterOfDefense());
        allAdvisors.Add(new MinisterofHealthcare());
        allAdvisors.Add(new CoronaCommissioner());
        allAdvisors.Add(new MicrosoftCEO());

        //Ask advise from each:
        foreach (IAdvisor advisor in allAdvisors)
        {
            advisor.Advise();
        }
    }
}
```

Figure 25: Interfaces example

5.4 Composition vs Aggregation

= Creating an instance of an object from a class in another class.

5.4.1 Example

```
public class PC
{
    private Disk _disk;
}

public class Disk
{
}
```

Figure 26: How can we make an instance of '_disk'?

Option 1

```
public class PC
{
    private Disk _disk = new Disk();
}
```

Option 2

```
public class PC
{
    private Disk _disk;

    public PC(bool parameter)
    {
        if (parameter)
            _disk = new Disk();
        //else _disk == null
    }
}
```

Option 3

```
public class PC
{
    private Disk _disk;
    public Disk Disk
    {
        get { return _disk; }
        set { _disk = value; }
    }
}

// somewhere in code
Disk myDisk = new Disk();
myPC.Disk = myDisk;
```

Figure 27: Solution: 3 options

Option 1: creating it at the start of the class (=composition)

Option 2: using the constructor (=composition)

Option 3: Outside of the class, by creating a new object an assigning that object to the '_disk' property in the class.

The 3rd option is called **Aggregation**. Unlike the other options, the 'myDisk' object keeps existing even if the PC class stops existing.

5.4.2 Interfaces in Xamarin(.Forms)

Interfaces in Xamarin(.Forms) is extremely useful:

Say you need to get data from a specific sensor on your Android/iOS/UWP device. We can create an interface in our project, and implement that interface in each device's code. Once we need to get that data, we can automatically call the correct method for the correct device.

5.5 Summary

- You are convinced by the advantages of **inheritance** and **polymorphism**, and can explain using an example.
- You understand the usage and consequences of the **virtual** and **abstract** keywords for properties and methods.

- You know when to use **abstract classes** and/or **interfaces**, and can explain the difference between those two.
- You understand the specific importance of interfaces in **Xamarin**(.Forms)

6 Asynchronous programming: async and await keywords

6.1 Why?

To make the app continue to respond to use interaction while. . . :

- . . . Reading from or writing to a database or file
- . . . Accessing a web service
- . . . Performing data processing

6.2 Example: newspaper app

- Load the trending newspaper items
- Load user preferences
- Load newspaper items based on preferences ⇒ needs to wait until user preferences are set

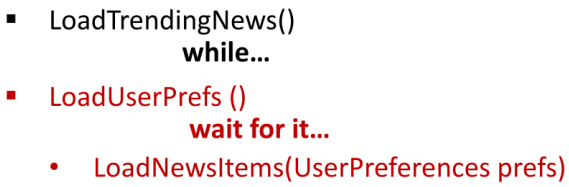


Figure 28

6.2.1 Problem

```
public async Task LoadNewsItemsAsync()
{
      LoadTrendingNewsAsync();
      LoadUserPrefsAsync();
      LoadNewsItemsAsync(prefs);
}
```

This function needs the user preferences that are being loaded in LoadUserPrefsAsync()
however since this function is async this will **not** wait for it!

Figure 29: LoadNewsItemsAsync(prefs) needs to wait

6.2.2 Solution

```
public async Task LoadNewsItemsAsync()
{
        LoadTrendingNewsAsync();
        var prefs = await LoadUserPrefsAsync();
        LoadNewsItemsAsync(prefs);
}
```

Figure 30: Solution: use the 'await' keyword

```
public async Task LoadNewsItemsAsync()
{
        LoadTrendingNewsAsync();
        var prefs =
            await LoadUserPrefsAsync();
        LoadNewsItemsAsync(prefs);
}
```

Trending items are being loaded, **while**

… user preferences are being loaded, **while**

… UI does not freeze!

However: news items are only being loaded **after** the user preferences

Figure 31

6.3 Creating an asynchronous function

6.3.1 With void-functions: use Task

```
public static void SavePreferences()
{
    //save preferences synchronously
    throw new NotImplementedException();
}
```

```
public static async Task SavePreferencesAsync()
{
    //save preferences asynchronously
    await Task.Yield();
    throw new NotImplementedException();
}
```

Figure 32: Turning a void function asynchronous

6.3.2 With a function that returns a type T

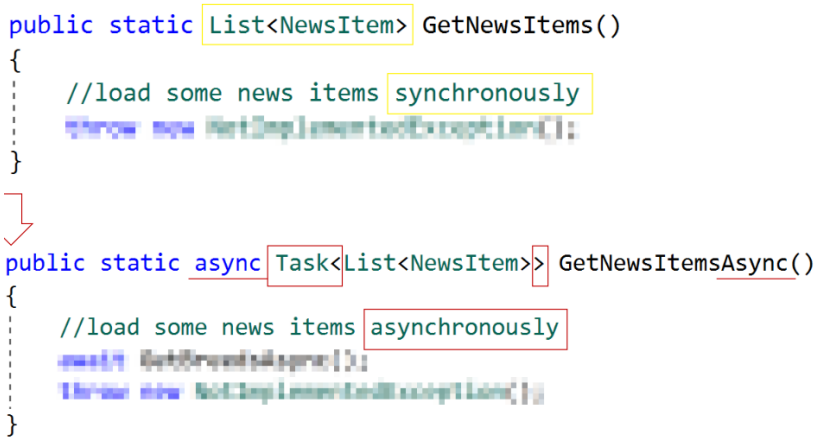


Figure 33: Turning a type function asynchronous

6.4 Calling an asynchronous function

- 'async' in the headers of:
  - the function you call it in (FillNewsItemsAsync)
  - the function you are calling (GetNewsItemsAsync)
- await when calling the function

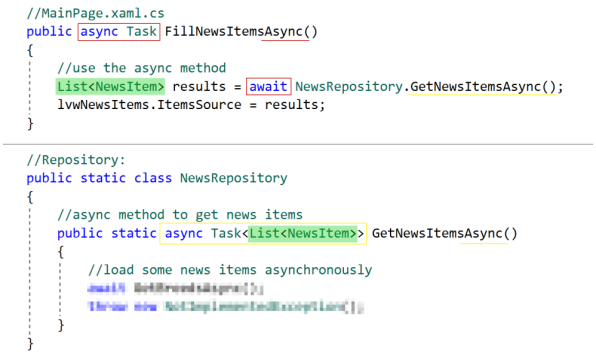


Figure 34: The await keyword

6.5 Summary

- The difference between **synchronous** and **asynchronous** programming
- The **what** and the **why** of asynchronous programming
- The meaning and usage of the **async** keyword
- The meaning and usage of the **await** keyword
- The **return type** of async functions

7 XAML in Xamarin.Forms

- XAML was created by Microsoft specifically to describe UI
- Xamarin Forms + XAML = complete mobile app

7.1 Benefits

- Seperation of UI from behaviour
- Easy designing of a UI, designer friendly

7.2 Pages

Xamarin.Forms Pages represent **cross-platform** mobile application screens

Figure 35: Types of pages in Xamarin.Forms

7.2.1 Adding a XAML Page

There are two Item Templates available to add XAML Content

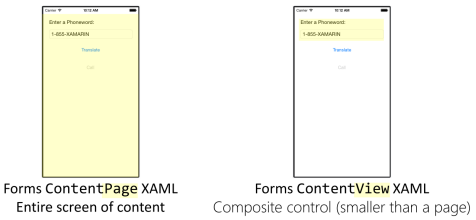


Figure 36: ContentPage vs ContentView

7.3 Steps to build a UI with XAML

7.3.1 Describing a screen

- XAML is used to construct object graphs, in this case a visual Page
- XML based: case sensitive, open tags must be closed, etc . . .

```
<?xml version="1.0" encoding="UTF-8" ?>
<ContentPage ...>
   <StackLayout Padding="20" Spacing="10">
      <Label Text="Enter a Phoneword:" />
      <Entry Placeholder="Number" />
      <Button Text="Translate" />
      <Button Text="Call" IsEnabled="False" />
   </StackLayout>
</ContentPage>
```

Figure 37: Example

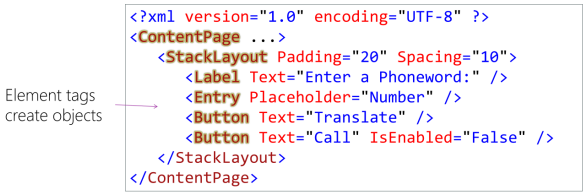


Figure 38: Element tags

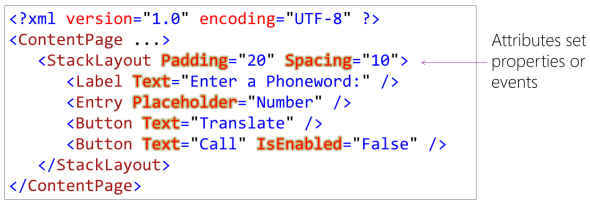


Figure 39: Attributes

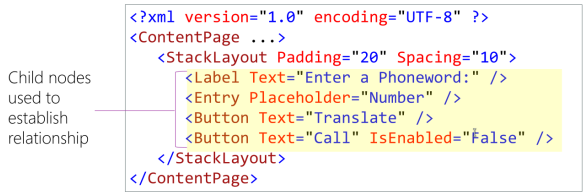


Figure 40: Child nodes in a StackLayout

7.3.2 C# code with XAML

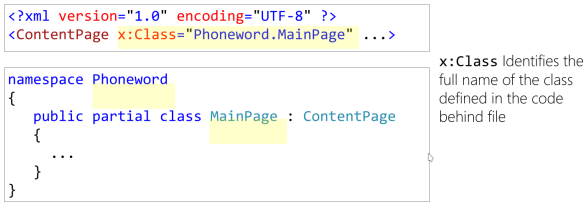


Figure 41: x:Class

7.3.3 XAML Initialization

The code behind the constructor has a call to 'InitializeComponent' which is responsible for loading the XAML and creating the objects. This method needs to run before doing anything else.

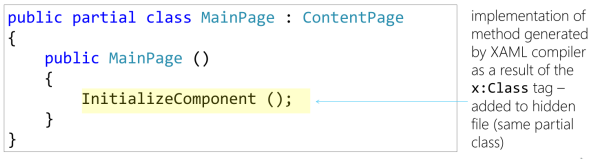


Figure 42: InitializeComponent

7.3.4 Naming elements in XAML

- Use **x:Name** to assign field name ⇒ allows you to reference element in XAML and C#
- This adds a private field to the XAML-generated partial class (file ending in **.g.cs**)
- Name must conform to C# naming conventions
- Name must be unique

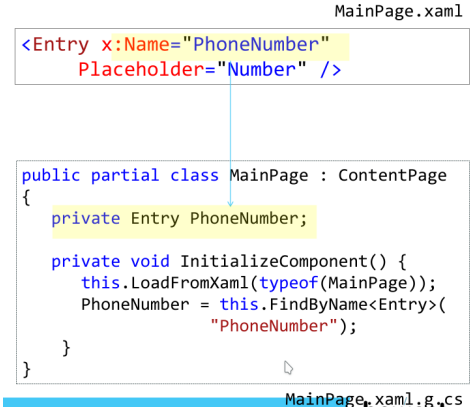


Figure 43: Naming elements

7.3.5 Handling events in XAML

- You can wire up events in XAML
- The event handler must be defined in the C# file and have proper signature

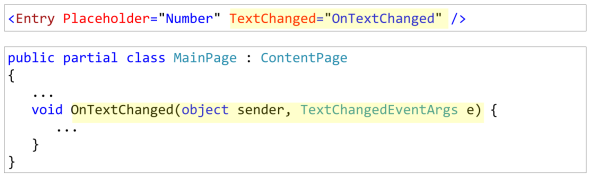


Figure 44: Event handling with the TextChanged Attribute

7.3.6 Handling events in the code behind

Can work with named elements as long as you define them in code, but keep in mind the field is not set until **After** 'InitializeComponent' is called

```
public partial class MainPage : ContentPage
{
    public MainPage () {
        InitializeComponent ();
        PhoneNumber.TextChanged += OnTextChanged;
    }

    void OnTextChanged(object sender, TextChangedEventArgs e) {
       ...
    }
}
```

Figure 45: Adding event handler directly in code instead of in the XAML file

- Many developers prefer to wire up all events in code behind by naming the XAML elements and adding event handlers in code
  - Keeps the UI layer 'pure' by pushing all behavior + management into the code behind
  - Names are validated at compile time, but event handlers are not
  - Easier to see how logic is wired up
- Pick the approach that works for your team / preferences

7.4 Layout in Xamarin.Forms

7.4.1 Motivation

- using layout containers to calculate view size and position helps your UI adapt to varied screen dimensions and resolutions
- Sizes/positions are recalculated automatically when device rotates

7.4.2 Layouts

A layout is a Xamarin.Forms container that determines the size and position for a collection of children

Figure 46: Layouts

7.4.3 Views

Views are the building blocks of cross-platform mobile user interfaces

- User-interface objects such as labels, buttons, sliders, . . .
- Commonly known as controls or widgets in other graphical programming environments

7.4.4 Working with sizes

The rendered size of a view is a collaboration between the view itself and its layout container

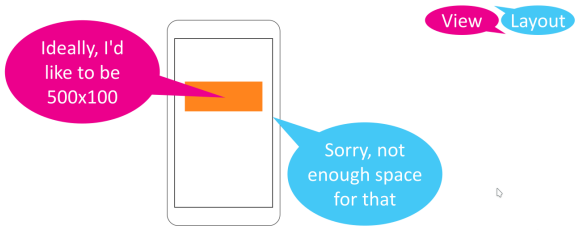


Figure 47: XAML view vs sizing

7.4.5 Layout algorithm

1. Layout panel asks each child how much room it would like
2. Then tells each child how much it gets

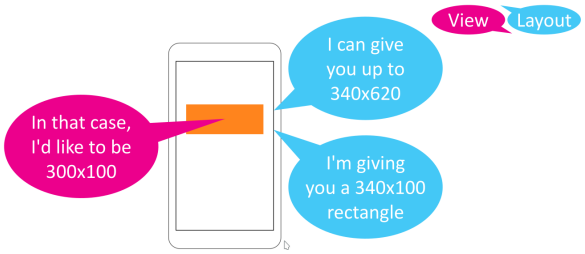


Figure 48

7.4.6 Default view sizing

By default, most views try to size themselves just large enough to hold their content

Figure 49: Default sizing

7.4.7 View preferences

- A view has four properties that influence its rendered size
- they are all requests and may be overruled by the layout container

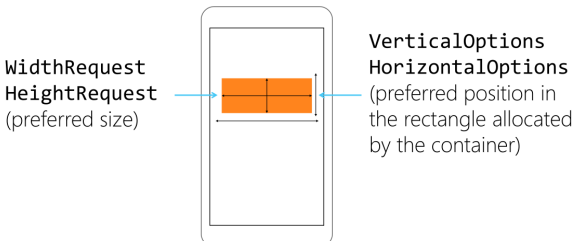


Figure 50: View preferences using WidthRequest and HeightRequest, VerticalOptions and HorizontalOptions

7.4.8 Sizing requests

A view can request a desired with and height

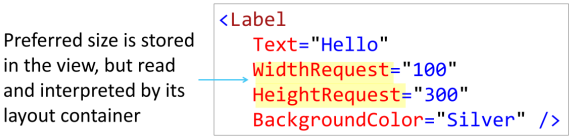


Figure 51: Preferred size

7.4.9 Size units

- Explicit sizes in Xamarin.Forms have no intrinsic units
- The values are interpreted by each platform according to that platform's rules
  - UWP: 'Effective pixels'
  - iOS: 'Points'
  - Android: 'Density-independant pixels'

7.4.10 Platform rendering

- Sizes set in Xamarin.Forms are passed to the underlying platform
- The platform will scale the values based on screen size and resolution

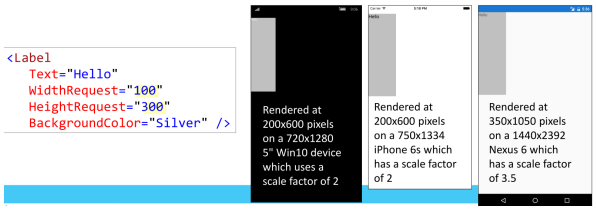


Figure 52: Platform rendering

7.4.11 Alignment

A view's preferred alignment determines its position and size within the rectangle allocated for it by its container

```
<StackLayout>
   <Label Text="Start"  HorizontalOptions="Start"  BackgroundColor="Silver" />
   <Label Text="Center" HorizontalOptions="Center" BackgroundColor="Silver" />
   <Label Text="End"    HorizontalOptions="End"    BackgroundColor="Silver" />
   <Label Text="Fill"   HorizontalOptions="Fill"   BackgroundColor="Silver" />
</StackLayout>
```

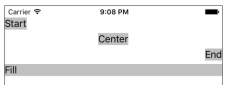
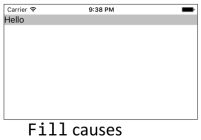


Figure 53: Alignment using HorizontalOptions

7.4.12 Fill

- The 'Fill' layout option generally overrides size preferences
- Horizontal and vertical alignment options generally default to Fill

```
<StackLayout>
   <Label Text="Hello"
      WidthRequest="100"
      HorizontalOptions="Fill"
      BackgroundColor="Silver" />
</StackLayout>
```



`Fill` causes `WidthRequest` to be ignored here

Figure 54: Fill overrides WidthRequest

7.5 Margin & Padding

7.5.1 Margin

- = extra space around the outside of a view
- available in all views, including containers

```
<StackLayout>
   <BoxView Color="Silver" Margin="10,20,30,40"/>
   <BoxView Color="Blue"   Margin="50,60"/>
   <BoxView Color="Gray"   Margin="70"/>
</StackLayout>
```

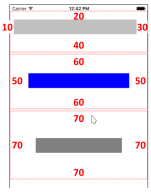


Figure 55: Margins

7.5.2 Padding

- = extra space on the inside of a layout that creates a gap between the children and the layout itself
- applicable only to layouts

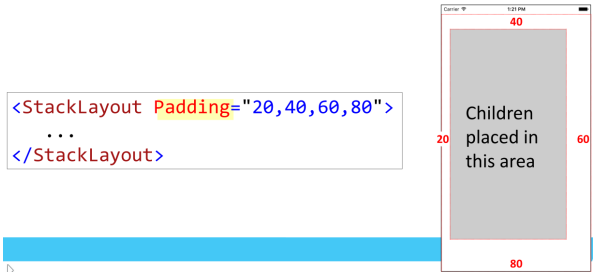


Figure 56

7.6 Arrange views with StackLayout

StackLayout arranges its children in a single column or a single row

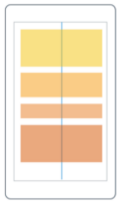


Figure 57: StackLayout

7.6.1 Adding children

Figure 58: Children in XAML

7.6.2 Child ordering

- Child layout order is determined by the order they were added to the Children collection
- Applies to both code and XAML

7.6.3 Child spacing

- StackLayout's Spacing separates the children
- The default is 6

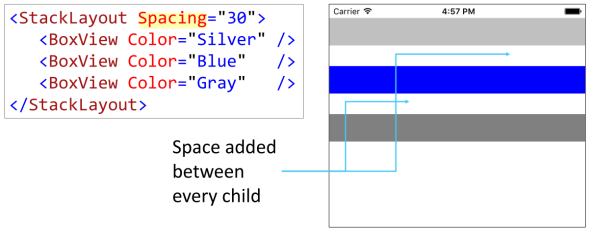


Figure 59: Add space between children

7.6.4 Orientation

- StackLayout's Orientation property lets you choose a vertical column or a horizontal row
- Vertical is the default

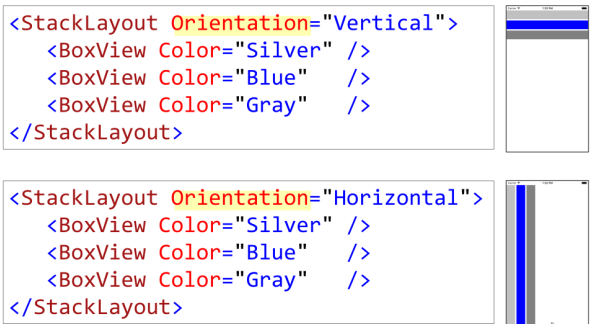


Figure 60

LayoutOptions with and against orientation

- In the direction opposite of its orientation, StackLayout uses the **Start**, **Center**, **End** and **Fill** layout options
- In the direction of its orientation, StackLayout ignores the Start, Center, End, and Fill layout options

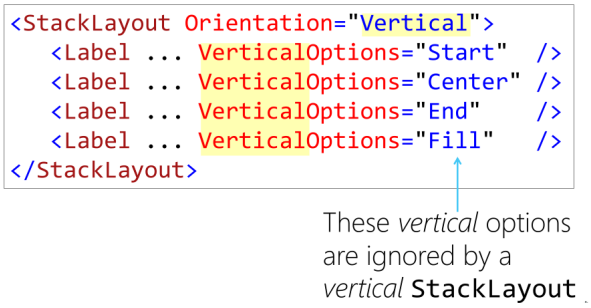


Figure 61: With orientation

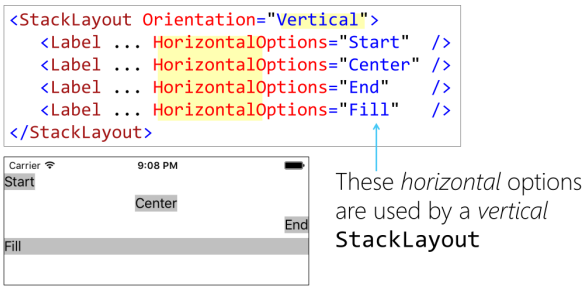


Figure 62: Against orientation

7.7 Expansion

- A view's **expansion setting** determines whether it would like the StackLayout to allocate available extra space to its rectangle

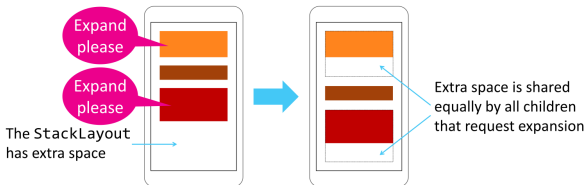


Figure 63: Expansion

7.7.1 Expansion direction

- StackLayout expands children only in the direction of its orientation
- For example: a vertical StackLayout expands vertically

7.7.2 How much extra space?

StackLayout determines the amount of extra space using its standard layout calculation as if there were no expansion

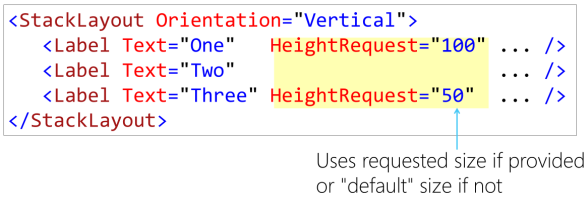


Figure 64: Amount of extra space

7.7.3 How to specify expansion?

To request expansion, use the "...AndExpand" version of the layout options in the direction of the StackLayout's orientation

```
<StackLayout Orientation="Vertical">
   <Label ... VerticalOptions="StartAndExpand"  />
   <Label ... VerticalOptions="CenterAndExpand" />
   <Label ... VerticalOptions="EndAndExpand"    />
   <Label ... VerticalOptions="FillAndExpand"   />
</StackLayout>
```

These settings give a `LayoutOptions` instance with `Expands` set to `true`

Figure 65

7.7.4 Expansion vs view size

- Enabling expansion can change the size of the view's layout rectangle, but doesn't change the size of the view unless it uses FillAndExpand
- In the direction **opposite** its orientation, adding "...AndExpand" tot the layout options has no effect (there is no expansion in that direction)

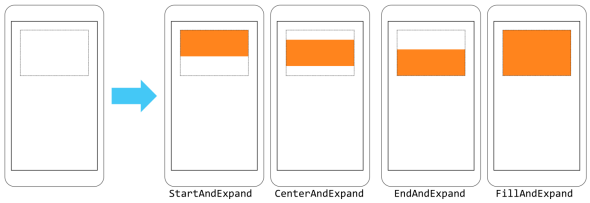
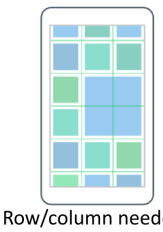


Figure 66

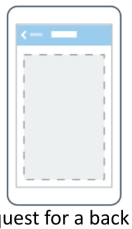
7.8 Apply attached properties

7.8.1 Motivation

Some properties are only needed in specific situations



Row/column needed when in a `Grid`



Request for a back button needed when in a `NavigationPage`

Figure 67

7.8.2 What is an attached property?

= a property that is defined in one class but set on objects of other types

- Button does not have Row/Column properties
- They are defined in Grid and attached to objects of other types as needed
  - Grid.Row="0", Grid.Row="1", . . .

7.8.3 Who consumes attached properties?

Typically, a container will look for attached properties on its children

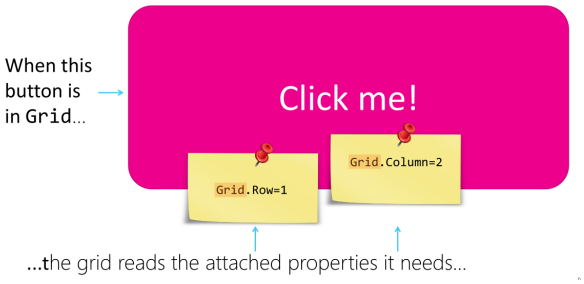


Figure 68

7.8.4 Apply an attached property in code

In code, use the static Set method to apply an attached property

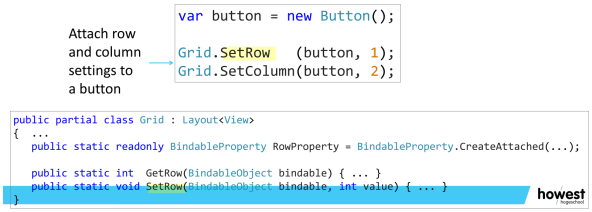


Figure 69: Attached property in code

7.8.5 Apply an attached property in XAML

In XAML, use the owning class name and the attached property name (without the Property suffix)

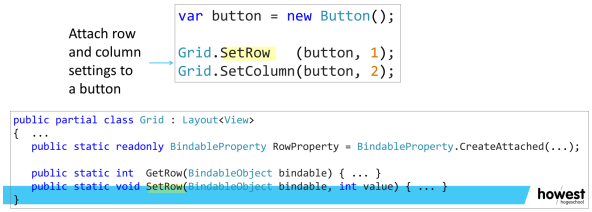


Figure 70: Attached property in XAML

7.9 Grids

Grid places its children into cells formed from rows and columns

Figure 71

7.9.1 Grid rows/columns

You specify the shape of the grid by defining each row and column individually

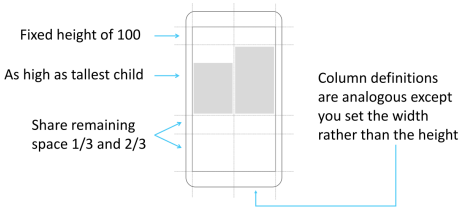


Figure 72

There are dedicated classes that define a row or a column

Specify row height

```
public sealed class RowDefinition : ...
{  ...
   public GridLength Height { get; set; }
}
```

Specify column width

```
public sealed class ColumnDefinition : ...
{  ...
   public GridLength Width { get; set; }
}
```

Figure 73: RowDefinition and ColumnDefinition

7.9.2 GridLength

GridLength encapsulates two things: unit and value

```
public struct GridLength
{  ...
   public GridUnitType GridUnitType { get; }
   public double       Value        { get; }
}
```

Units can be: Absolute, Auto, Star

Figure 74

Absolute GridLength

Absolute GridLength specifies a fixed row height or column width

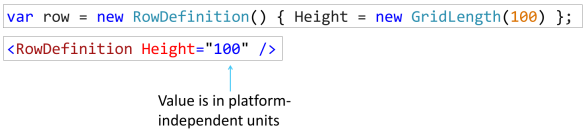


Figure 75: Absolute GridLength

Auto GridLength

Auto GridLength lets the row height or column width adapt, it automatically becomes the size of the largest child

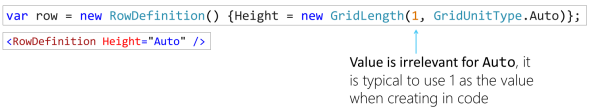


Figure 76: Auto GridLength

Star (*) GridLength

Star GridLength shares the available space proportionally among all rows/columns that use star sizing

Figure 77: Star GridLength

7.9.3 Grid example

It is common to mix different GridLength settings in the same grid

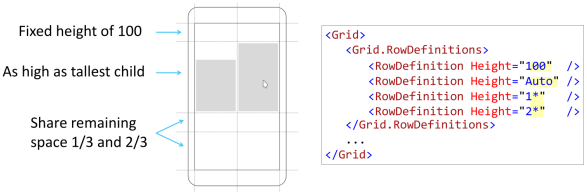


Figure 78

7.9.4 Default size

Rows and columns default to **'1*'** size

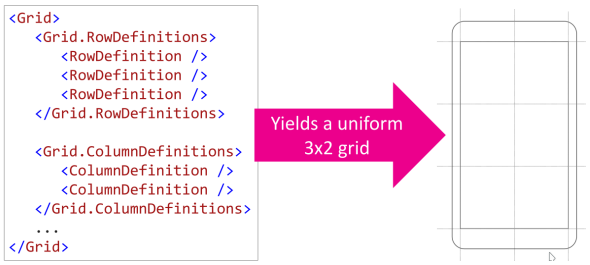


Figure 79

7.9.5 Row/Column numbering

Starts at 0:

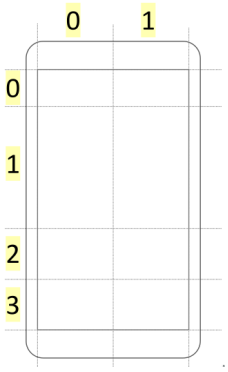


Figure 80

7.9.6 Grid positioning properties

Grid defines four attached properties used to position children

ATTACHED PROPERTY	VALUE
Column	An integer that represents the Column in which the item will appear.
ColumnSpan	An integer that represents the number of Columns that the item will span.
Row	An integer that represents the row in which the item will appear.
RowSpan	An integer that represents the number of rows that the item will span.

Figure 81

7.9.7 Cell specification

Apply the Row and Column attached properties to each child

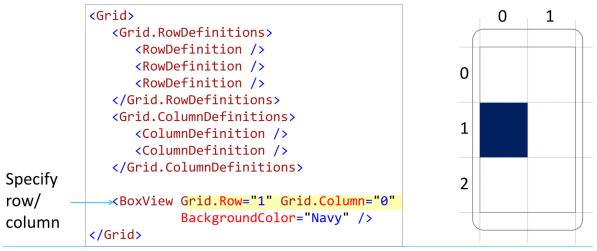


Figure 82

7.9.8 Span specification

Apply RowSpan and ColumnSpan to each child as needed

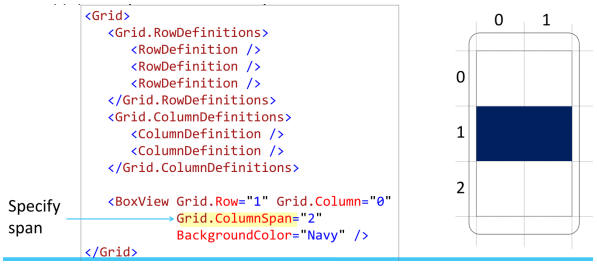


Figure 83

7.9.9 Cell and span defaults

Cell locations default to 0 and spans default to 1

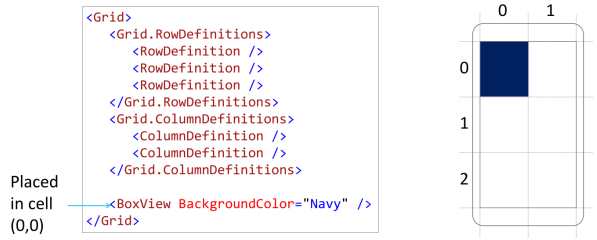


Figure 84

7.9.10 Layout options

A view's horizontal and vertical layout options control how it is sized and positioned within its grid cell (the default is Fill)

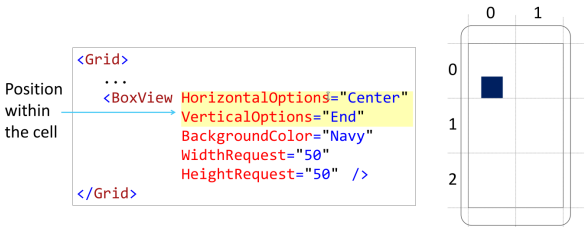


Figure 85

7.9.11 Grid child spacing

- Grid's RowSpacing and ColumnSpacing properties separate the children
- They both default to 6

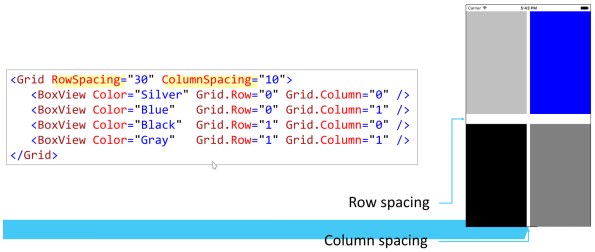


Figure 86

7.9.12 Auto-generated rows/columns

Grid will automatically generate equal-sized rows/columns based on the position of the children you add

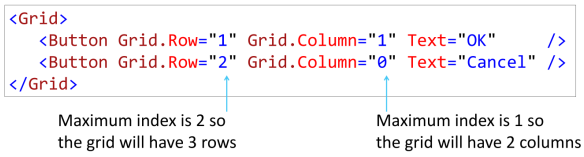


Figure 87

7.10 Scroll a layout with ScrollView

ScrollView adds scrolling to a single piece of content; the content can be an individual view or a layout container

7.10.1 How to use ScrollView

Wrap a ScrollView around a single element to add scrolling

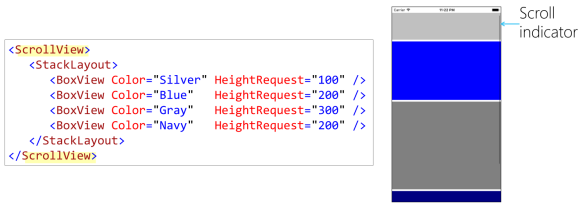


Figure 88

7.10.2 ScrollView orientation

ScrollView lets you control the scroll direction:

- Vertical (the default)
- Horizontal
- or Both

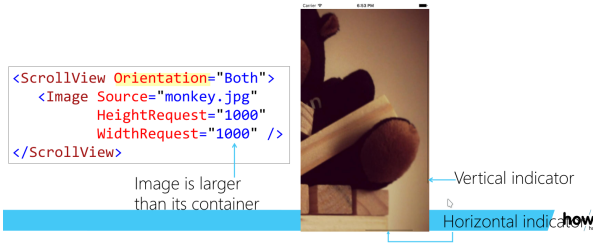


Figure 89

7.10.3 Do not nest scrolling views

Generally, do not nest ScrollViews or a ListView in a ScrollView, it often creates non-intuitive behavior

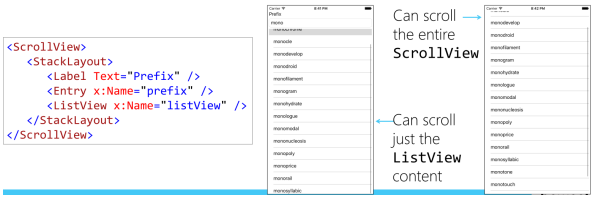


Figure 90: This is bad practice

7.11 Summary

- Examining XAML syntax
- Adding Behavior to XAML-based pages
- Exploring XAML capabilities
- Specify the size of a view
- Arrange views with StackLayout

- Apply Attached Properties
- Arrange views with Grid
- Scroll a layout with ScrollView