

Device Programming

Tuur Vanhoutte

October 12, 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#	4
2.2	Datatypes	4
2.3	Collections	4
2.3.1	Arrays	4
2.3.2	Dictionary <TKey, TValue>	4
2.3.3	List<T>	5
2.4	Selections	5
2.5	Loops	6
2.6	Classes	6
2.7	Instantiate objects	6
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	8
3.4	Embedded files	8
3.4.1	Read an embedded file in Xamarin	9
3.4.2	Processing the file's content	9
3.5	Summary	9
4	Navigation	9
4.1	Modal vs Modeless	9
4.2	Navigate forward	10
4.3	Navigate back	10
4.3.1	Go back - Modeless page	10
4.3.2	Go back - Modal page	11
4.4	Navigation stack	11
4.5	Page types	12
4.6	Exchanging data	12
4.7	Summary	13

5	Object Orientation	13
5.1	Inheritance	13
5.1.1	Constructor	14
5.1.2	Access modifiers	14
5.1.3	Properties/methods: VIRTUAL and OVERRIDE	14
5.1.4	Properties/methods: ABSTRACT and OVERRIDE	15
5.2	Polymorphisme	16
5.2.1	Disadvantage	17
5.3	Interfaces	17
5.3.1	Summary	17
5.3.2	Example	18
5.4	Composition	18
5.4.1	Example	19
5.4.2	Interfaces in Xamarin(.Forms)	19
5.5	Summary	19

1 .NET

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

* languages + libraries

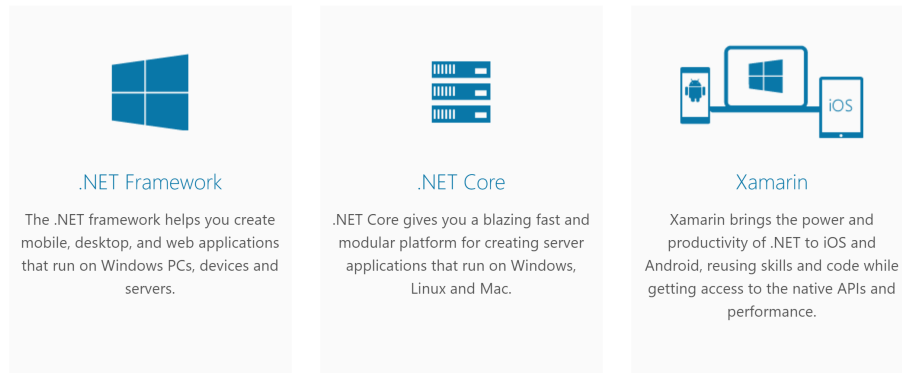


Figure 1: .NET ecosystem

1.1 Languages

- Syntax very similar to C, C++, Java & JavaScript
- Functional programming language, cross-platform, open source
- Approachable English-like language for 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

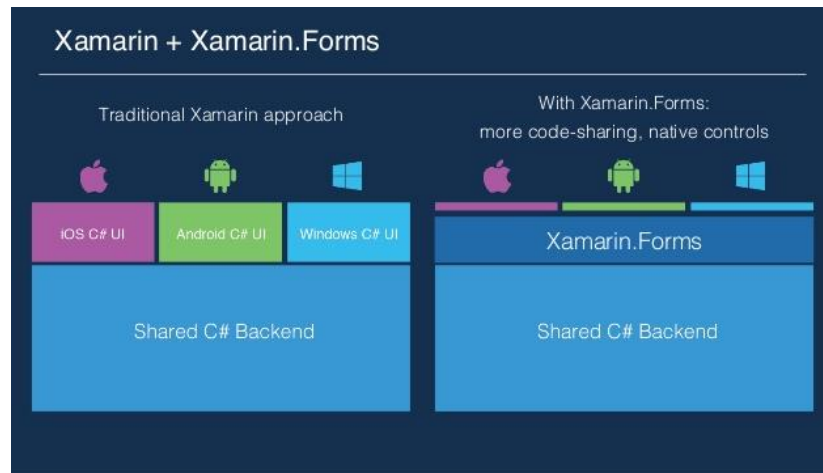


Figure 3: Native vs Xamarin.Forms

1.3.2 Xamarin - Code Sharing strategy



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

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#

- curly brackets { } in plaats van indenting

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

```
//collections of type Person:
Person[] teacherArr = new Person[10];
List<Person> teacherList = new List<Person>();

//You can only add Person objects to these collections!
```

2.3.1 Arrays

= meerdere variabelen van hetzelfde type

```
//initialize int array with 10 positions:
int[] numbers = new int[10];
//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
- Auto-implemented properties have a hidden field
- Use properties to control field access
- Enhance input/output control using get & set
- Calculated properties build 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
- Embedded Files
- System.IO

3.1 Namespaces

```
• using Xamarin.Forms;  
    namespace  
  
• System.Diagnostics.Debug.WriteLine("DEVPROG");  
    namespace    class
```

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."

3.3 System.IO

= Input/Output <https://developer.xamarin.com/api/namespace/System.IO/>

- StreamReader <https://developer.xamarin.com/api/type/System.IO.StreamReader/>
- StreamWriter <https://developer.xamarin.com/api/type/System.IO.StreamWriter/>

3.4 Embedded files

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

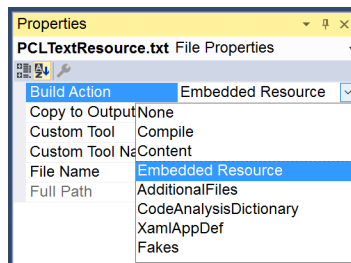


Figure 7: Embedded files inladen in een Visual Studio project: rechtermuisknop op 1 of meerdere files ⇒ properties ⇒ build action = Embedded resources

3.4.1 Read an embedded file in Xamarin

```
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

•

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

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

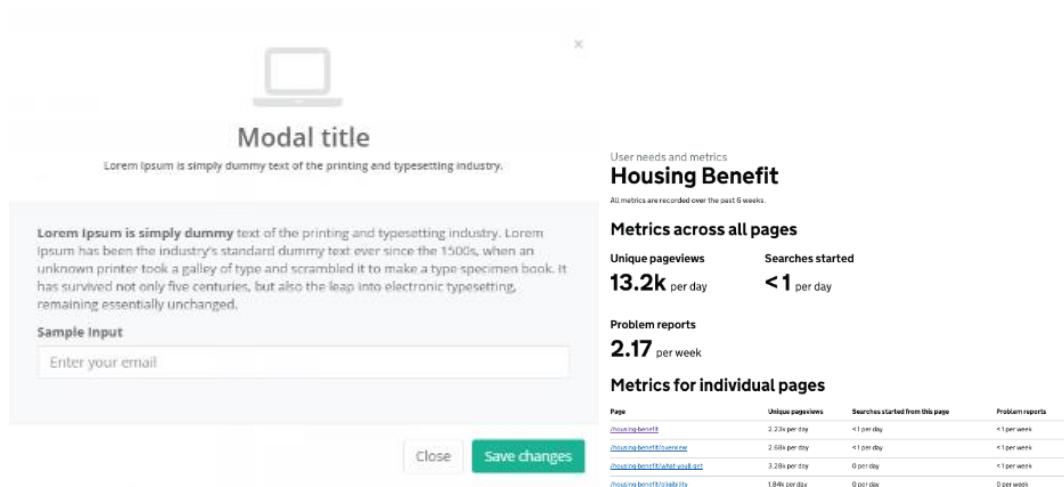


Figure 10: Modal page vs Modeless page

4.2 Navigate forward

```
Navigation.PushAsync(new FooPage());
Navigation.PushModalAsync(new FooPage());

// 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

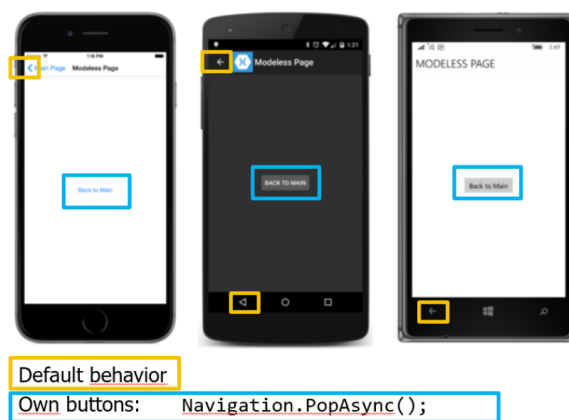


Figure 11

4.3.2 Go back - Modal page

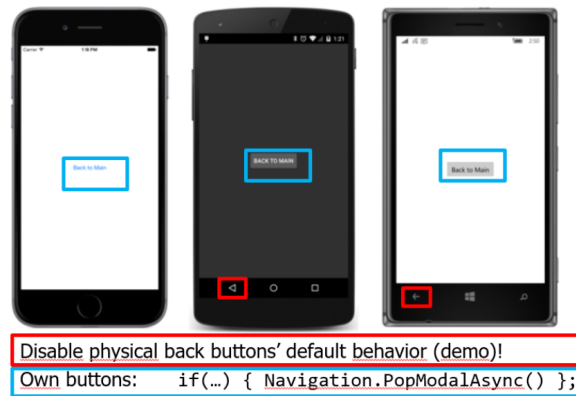


Figure 12

4.4 Navigation stack



Figure 13: Pushen op de stack

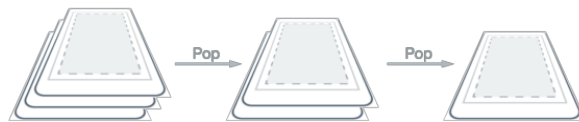


Figure 14: Pop'en van de stack



Figure 15: InsertPageBefore

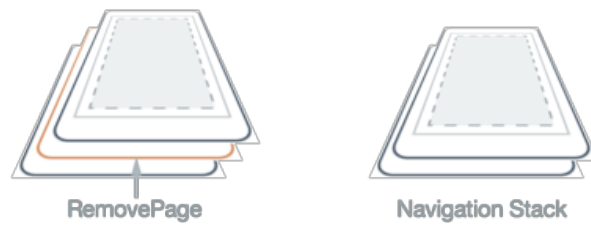


Figure 16: RemovePage

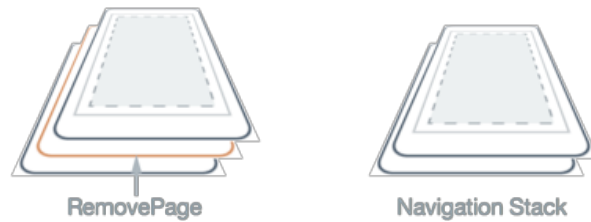


Figure 17: PopToRoot

4.5 Page types

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

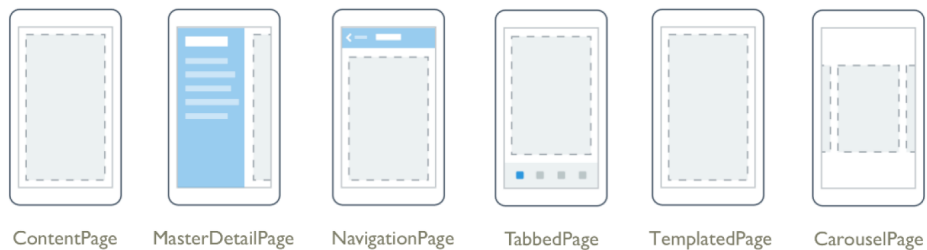


Figure 18

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

= klassen 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 : Advisor {  
    // code here  
}
```

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

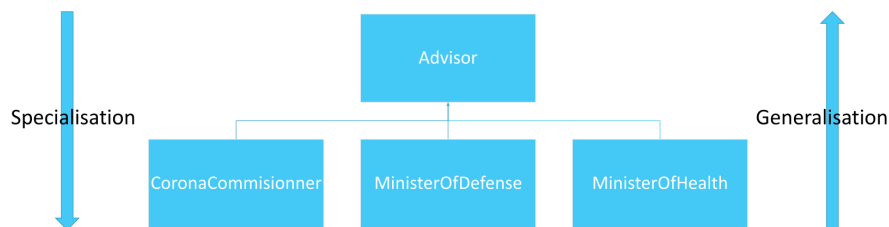
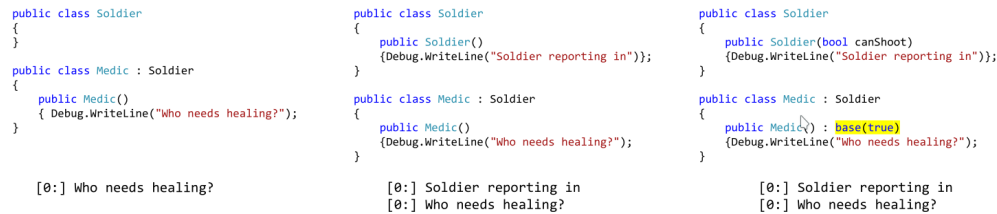


Figure 19

- **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



```
public class Soldier
{
}

public class Medic : Soldier
{
    public Medic()
    { Debug.WriteLine("Who needs healing?");
    }
}

[0:] Who needs healing?
```

```
public class Soldier
{
    public Soldier()
    { Debug.WriteLine("Soldier reporting in");
    }
}

public class Medic : Soldier
{
    public Medic()
    { Debug.WriteLine("Who needs healing?");
    }
}

[0:] Soldier reporting in
[0:] Who needs healing?
```

```
public class Soldier
{
    public Soldier(bool canShoot)
    { Debug.WriteLine("Soldier reporting in");
    }
}

public class Medic : Soldier
{
    public Medic() : base(true)
    { Debug.WriteLine("Who needs healing?");
    }
}

[0:] Soldier reporting in
[0:] Who needs healing?
```

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 method in the derived method.

Virtual properties/methods:

- Default implementation in base class
- 'virtual' keyword, to **replace** the way an object behaves
- **CAN** be overridden 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

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

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:

```

List<Animal> animals = new List<Animal>();
Animal dog = new Dog();
Animal cat = new Cat();

animals.Add(dog);
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)

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 Advice() 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

= 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 and 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, by just calling the interface’s method.

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)