Anatomie van een C# klasse

Default constructor

```
class Vierkant
{
    public Vierkant() : this(1)
    {
    }
```

Roept de overloaded constructor aan die een `int` als parameter aanvaardt en geeft de waarde `1` mee.

Overloaded constructor

```
    public Vierkant(int zijdeIn)
    {
        Zijde = zijdeIn;
    }
```

Instantievariabele

```
    private int zijde = 0;
```

Full property

```
    public int Zijde
    {
        get { return zijde; }
        set
        {
            if (value >= 0)
                zijde = value;
            else
            {
                string fout = "Zijde mag niet negatief zijn";
                throw new Exception(fout);
            }
        }
    }
```

Uitzondering wordt opgeworpen (zie sectie “Exception handling”).

Read-only property die transformeert

```
    public double Oppervlakte
    {
        get
        {
            return Math.Pow(Zijde, 2);
        }
    }
```

Methode

```
    public void ToonOppervlakte()
    {
        Console.WriteLine(Oppervlakte);
    }
}
```

Gebruik klasse

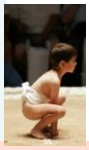
O.L. constr.
```
Vierkant oudeKader = new Vierkant(9);
```
Def. constr.
```
Vierkant nieuweKader = new Vierkant();
Console.WriteLine("Hoe groot is dit kader?");
int input = int.Parse(Console.ReadLine());
```
Setter Zijde
```
nieuweKader.Zijde = input;

Console.WriteLine("Grootste heeft opp:");
```
Getter Zijde
```
if (nieuweKader.Oppervlakte <= oudeKader.Oppervlakte)
```
Methode
```
        oudeKader.ToonOppervlakte();
else
```
Methode
```
        nieuweKader.ToonOppervlakte();
```

Access modifiers

`public`	Overal zichtbaar.
`protected`	Enkel klasse & child-klassen.
`private` of [niets]	Enkel in klasse zelf.

stack heap



By value
`int`, `char`, etc.
`enum`
`struct`

By reference
Arrays
Objecten

`static` vs non-static

static autoproperty met private set

Autoproperty

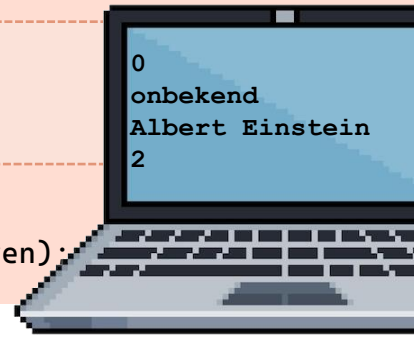
```
class Student
{
    public static int AantalIngeschreven { get; private set; }
    public Student()
    {
        AantalIngeschreven++;
    }
    public string Naam { get; set;} = "onbekend";
}
```

`static` properties en methoden worden op de klasse aangeroepen én horen niet bij de instanties van de klasse.

Voorbeeld gebruik:

Object initializer syntax

```
Console.WriteLine(Student.AantalIngeschreven);
Student deEerste = new Student();
Student deTweede = new Student()
{
    Naam = "Albert Einstein"
};
Console.WriteLine(deEerste.Naam);
Console.WriteLine(deTweede.Naam);
Console.WriteLine(Student.AantalIngeschreven);
```



Exception handling

```
try
{
    //code die uitzonderingen kan geven
}
catch (Exception)
{
    //uitzondering verwerken
}
finally
{
    //wordt uitgevoerd,
    //ongeacht resultaat
}
```

Fully Qualified Type Name

[Naam Namespace].[Naam klasse]
Bijv. `MijnProject.Student`

Veel gemaakte fouten

```
private int prijs;
public int Prijs
{
    get { return prijs; }
    set { Prijs = value; }
}
```

`StackOverFlowException` omdat `set` zichzelf aanroept (`Prijs` moet `prijs` zijn).

```
Student mijnStudent;
//...
mijnStudent.Naam = "Freddy";
```

`NullReferenceException` omdat object nog `null` is. Oplossen met `new` of controle op `null` doen.

Arrays van objecten

```
Vierkant[] veelKaders = new Vierkant[10];
for (int i = 0; i < veelKaders.Length; i++)
{
    veelKaders[i] = new Vierkant();
}
//Zijde van 4e vierkant aanpassen
veelKaders[3].Zijde = 12;
```

Enkel de array wordt geinstantieerd in de heap. **NIET de individuele objecten** die in de array moeten komen. Grootte is vereist!

Gebruik steeds `.Length` om totale lengte van array (of `List<>`) te kennen.

Nu maken we de individuele objecten aan in de heap en plaatsen ze in de array.

List<T>

Object Initializer Syntax

```
List<int> alleGetallen = new List<int>();
List<bool> binaryList = new List<bool>()
{
    true, false, true
};

var listOfStringarrays = new List<string[]>();

List<Vierkant> winkel = new List<Vierkant>();
```

Element toevoegen

```
winkel.Add(new Vierkant(5));
```

`var` in C# heeft geen nut, enkel dat je datatype niet moet hertypen (dus niet zoals in JS).

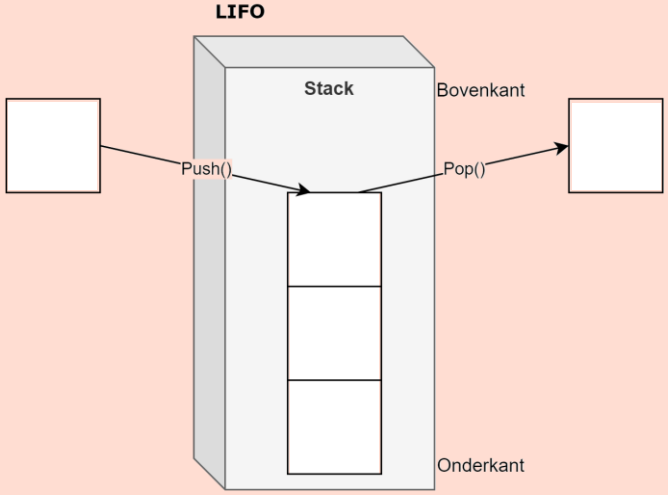
Andere nuttige List<T> methoden: `.Clear()`, `.Insert(…)`, `.IndexOf(…)`, `.RemoveAt(…)`, etc.

foreach

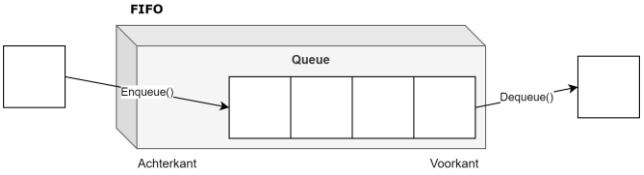
Iteration variable

```
foreach (var enkelItem in winkel)
{
    enkelItem.Zijde+=5;
}
```

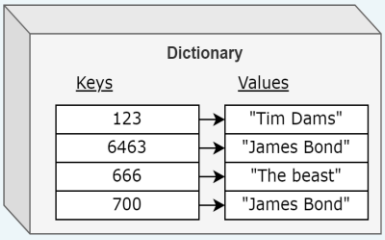
Stack<T>



Queue<T>



Dictionary<S,T>



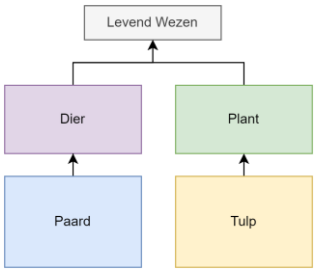
Key type **Value type**

```
var klanten = new Dictionary<int, string>();
klanten.Add(123, "Tim Dams");
//...
Console.WriteLine(klanten[123]);
foreach (var item in klanten)
{
    Console.WriteLine(item.Key + "\t:" + item.Value);
}
```

`key` is de unieke index van ieder element.

Overerving

```
class LevendWezen {}
class Dier:LevendWezen {}
class Paard:Dier {}
class Plant:LevendWezen {}
class Tulp:Plant {}
```



`:` duidt overerving aan. **"Is een"**-relatie.

Transitief	Child klasse erft alles over van parentklasse.
Multiple inheritance	Is niet mogelijk in C#.
`sealed`	Van deze klasse mag niet overgeërfd worden.
`protected`	Private maar ook zichtbaar in child klassen.

abstract

```
abstract class LevendWezen
{

}
```

- Abstracte klasse kan niet geïnstantieerd worden.
- Abstracte methoden en properties **moeten** *override* worden.
- Van zodra je 1 abstracte methode of property hebt, moet de klasse ook abstract zijn.

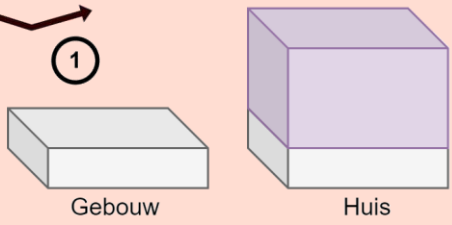
Volgorde van constructor calls

```
class Gebouw
{
    public int AantalVerdiepingen { get; private set; }
    public Gebouw(int verdiepingenIn)
    {
        AantalVerdiepingen = verdiepingenIn;
    }
}
class Huis: Gebouw
{
    public bool HeeftTuintje { get; private set; };
    public Huis(bool tuintjeIn, int verdiepingenIn): base(verdiepingenIn)
    {
        HeeftTuintje = tuintjeIn;
    }
}
```

START → 4 → 1 → 2 → 3 → 5 → EINDE

`:base()` roept versie (van methode of constructor) van parentklasse op.
`:this()` roept versie van klasse zelf op.

```
new Huis(true, 5);
```



virtual & override

```
class Vliegtuig
{
    public virtual void Vlieg()
    {
        Console.WriteLine("Ik vlieg");
    }
}
class Raket : Vliegtuig
{
    public override void Vlieg()
    {
        base.Vlieg();
        Console.WriteLine("\t hoger");
    }
}
```

Child klasse mag deze methode of property *override'n*.

Virtual versie van parent klasse aanpassen in childklasse. `base()` aanroep is optioneel.

Voorbeeld gebruik:

```
var boeing = new Vliegtuig();
var spaceX = new Raket();
boeing.Vlieg();
spaceX.Vlieg();
```

System.Object

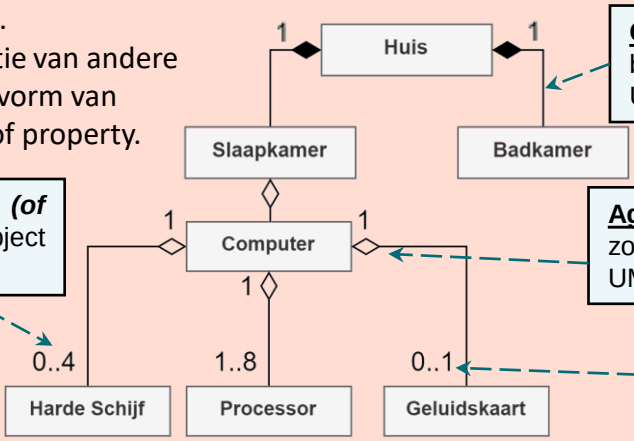
Parent-klasse van ALLE klassen.

Heeft volgende methoden als `virtual`:

`.Equals()`	Gebruikt om te ontdekken of twee instanties gelijk zijn.
`.GetHashCode()`	Geeft een unieke *hash* terug van het object; nuttig om o.a. te sorteren.
`.GetType()`	Geeft het datatype (de klasse) van het object terug.
`.ToString()`	Geeft een string terug die het object voorstelt.

Compositie en aggregatie

"Heeft een"-relatie.
Klasse heeft instantie van andere klasse in zich in de vorm van instantievariabele of property.



Compositie: interne object kan niet bestaan zonder omliggende object.
UML-notatie: volle ruit.

"Heeft meerdere (of geen)": interne object is een array of lijst.

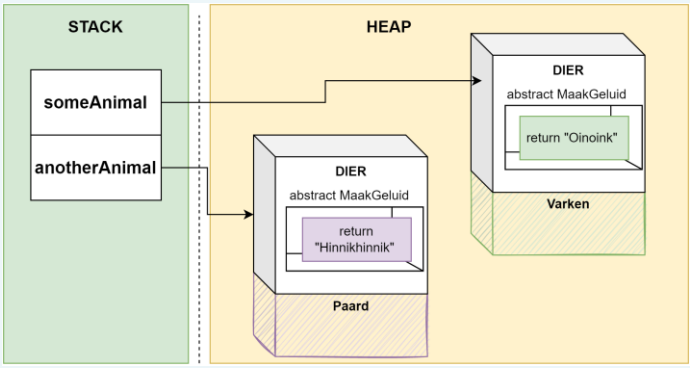
Aggregatie: interne object kan ook zonder omliggende object bestaan.
UML-notatie: lege ruit.

"Heeft 1 of geen": Interne object kan ook `null` zijn. Testen!

Polymorfisme

1. **Objecten kunnen zich voordoen als hun parent-type.**
2. **Maar ze kunnen wel "hun" *override'd* code uitvoeren**

```
abstract class Dier
{
    public abstract string MaakGeluid();
}
class Paard : Dier
{
    public override string MaakGeluid()
    {
        return "Hinnikhinnik";
    }
}
class Varken : Dier
{
    public override string MaakGeluid()
    {
        return "Oinkoink";
    }
}
```



Voorbeeld gebruik:

Met dank aan polymorfisme!

```
Dier dier1 = new Varken();
Dier dier2 = new Paard();
Console.WriteLine(dier1.MaakGeluid());
Console.WriteLine(dier2.MaakGeluid());
```

`List<Dier>` kan dus ook gebruik maken van polymorfisme!

this

Geeft referentie van/naar object zelf terug (enkel aanroepbaar in object zelf).

is

Geeft `bool` terug om aan te geven of variabele van bepaald datatype is (**of van child-klasse van datatype!**). Werkt ook voor interfaces (zie sectie "Interfaces")

```
if(dezePersoon is Student)
    //...
```

as

Cast object naar ander datatype indien mogelijk. Zo niet geeft het `null` terug.

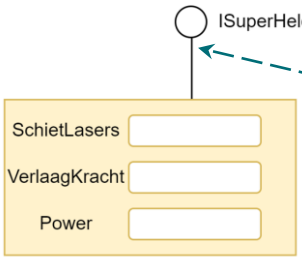
```
Student fritz = new Student();
Mens jos = fritz as Mens;
if (jos != null)
{
    //Doe Mens-zaken
}
```

Interfaces

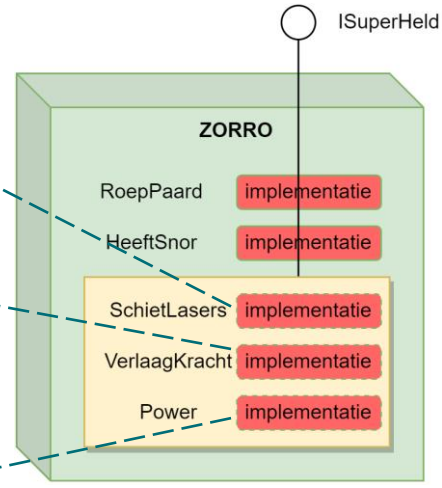
Een interface is niet meer dan een belofte: het zegt enkel welke publieke methoden en properties de klassen bezit. **Het zegt echter niets over de effectieve code-implementatie van deze methoden en properties.**

```
interface ISuperHeld
{
    void SchietLasers();
    int VerlaagKracht(bool isZwak);
    int Power { get; set; }
}

class Zorro : ISuperHeld
{
    public void RoepPaard() { ...}
    public bool HeeftSnor { get; set; }
    public void SchietLasers()
    {
        Console.WriteLine("pewpew");
    }
    public int VerlaagKracht(bool isZwak)
    {
        if (isZwak)
            return 5;
        return 10;
    }
    public int Power { get; set; }
}
```



UML aanduiding van een interface is m.b.v. een “lolly”.



Multiple inheritance mag niet bij. Maar een klasse mag wel meer dan 1 interface bevatten:

```
class DarthVader :
StarWarsCharacter,
IForceUser, Ipilot
{
```

is & as

Ideaal om te kijken of object interface heeft, en vervolgens dat deel ervan aan te roepen:

```
Zorro held = new Zorro();
if(held is ISuperHeld)
{
    ISuperHeld temp = held as ISuperHeld;
    temp.VerlaagKracht(true);
}
```

IComparable

```
interface IComparable
{
    int CompareTo(Object obj);
}
```

Ingebakken in .NET. Toegepast door Array.Sort()
(moet je dus niet manueel nog in je project toevoegen).

Voorbeeld gebruik:

```
class Land : IComparable
{
    public int CompareTo(object obj)
    {
        Land temp = obj as Land;
        if (temp != null)
        {
            return Oppervlakte.CompareTo(temp.Oppervlakte);
        }
        else
            throw new Exception("Object is not a Land");
    }
    public int Oppervlakte { get; set; }
}
```

Vanaf nu kunnen we sorteren:

```
Land[] eurolanden = new Land[3];
eurolanden[0] = new Land() { Oppervlakte = 5 };
eurolanden[1] = new Land() { Oppervlakte = 7 };
eurolanden[2] = new Land() { Oppervlakte = 6 };
Array.Sort(eurolanden);
```