C# OO Programmeren

LES 5

JOHAN DONNÉ

Tips: snippets

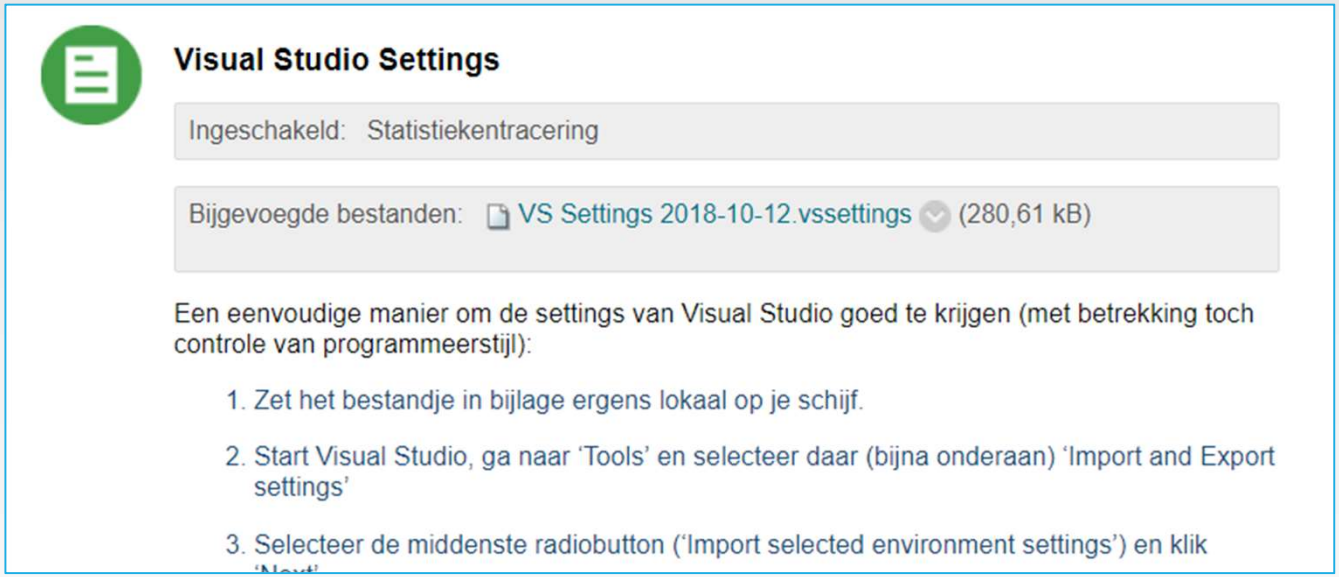
‘Snippets’ = autocomplete van typische stukjes code,
Activeren door 2x de ‘Tab’ toets in te drukken:

- ctor
- for
- fore
- prop, propg, propf,
- switch
- Bij event-handlers: +=
- …

==> Demo

Tips: settings

Item op Toledo over het automatisch goed zetten van de settings van Visual Studio



Visual Studio Settings

Ingeschakeld: Statistiekentracering

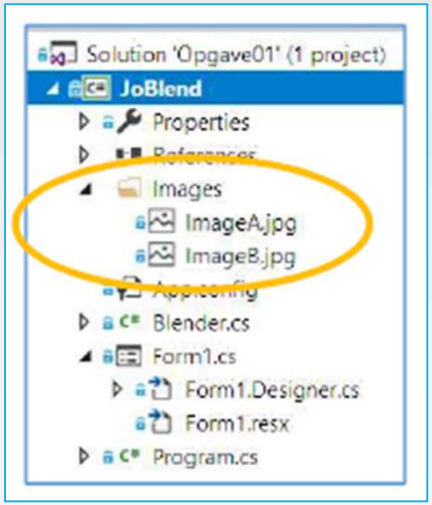
Bijgevoegde bestanden: VS Settings 2018-10-12.vssettings (280,61 kB)

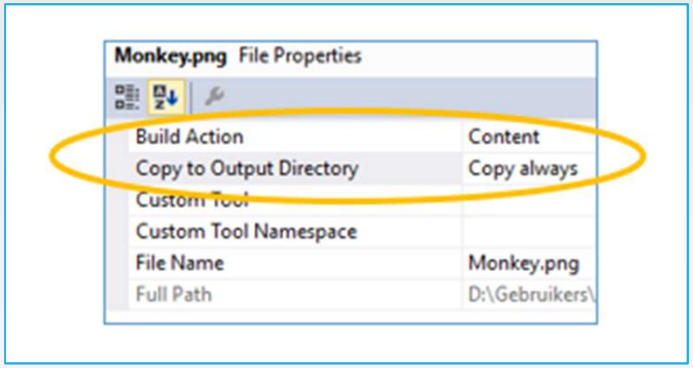
Een eenvoudige manier om de settings van Visual Studio goed te krijgen (met betrekking toch controle van programmeerstijl):

1. Zet het bestandje in bijlage ergens lokaal op je schijf.
2. Start Visual Studio, ga naar 'Tools' en selecteer daar (bijna onderaan) 'Import and Export settings'
3. Selecteer de middenste radiobutton ('Import selected environment settings') en klik

Tips: Toevoegen van databestanden aan toepassing

==> Installatie & gebruik Visual studio, p19 onderaan



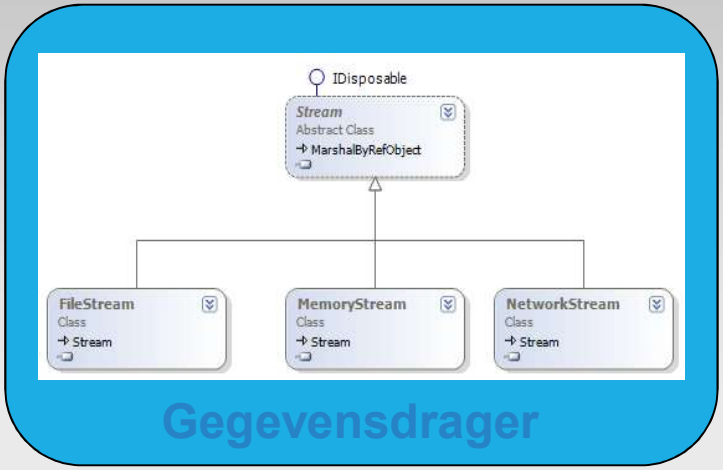


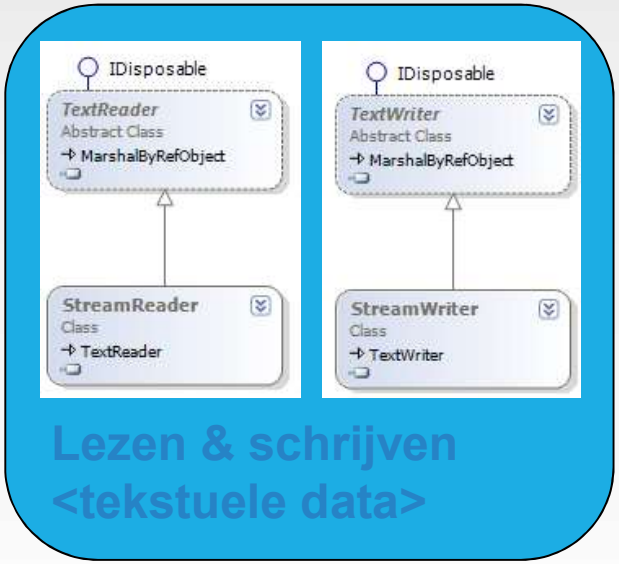
Overzicht

- File I/O (eigen literatuur)
- Dispose, using
- Serialisatie
- value/reference types
- Cloning

File I/O

I/O-gerelateerde klassen





Bestanden en mappen beheren

Namespace System.IO

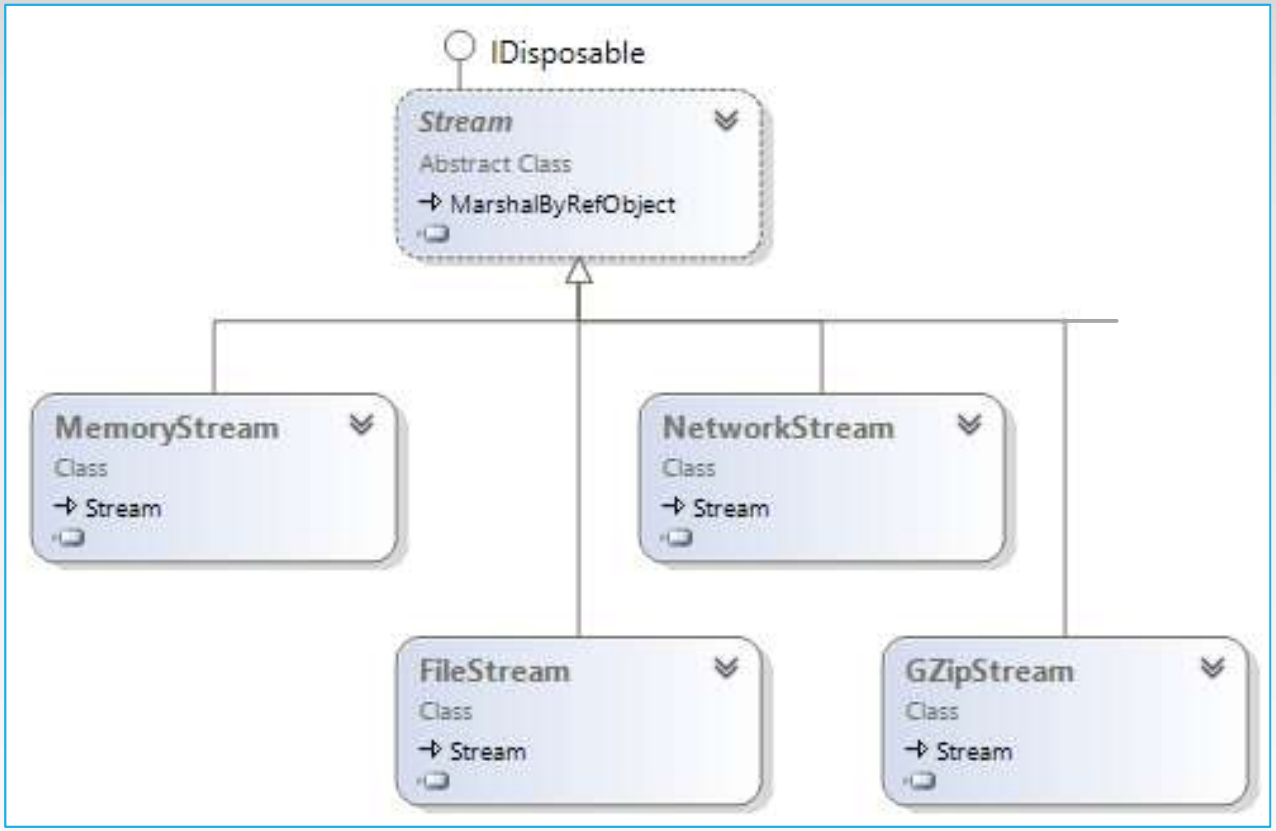
- DriveInfo
- DirectoryInfo
- FileInfo
- Demo

- 2 statische klassen: 'Directory' en 'File'

bv.: `var lines = System.IO.File.ReadAllLines(filePath);`

Streams

Concept voor I/O: Streams



Abstracte klasse 'Stream'

➔ generieke gegevensdrager: *"stroom van bytes"*

- omvat methoden **Close()** & **Flush()** & **Dispose()**

- Eigenschappen:

public abstract long **Length** { get; }
Member of System.IO.Stream

- Methoden om ***binaire*** data te lezen:

public virtual int **ReadByte**()
Member of System.IO.Stream

public abstract int **Read**(byte[] *buffer*, int *offset*, int *count*)
Member of System.IO.Stream

- Methoden om ***binaire*** data weg te schrijven:

public virtual **void WriteByte**(byte *value*)
Member of System.IO.Stream

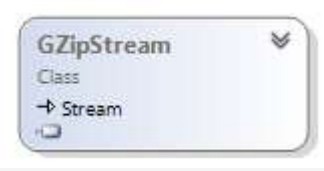
public abstract **void Write**(byte[] *buffer*, int *offset*, int *count*)
Member of System.IO.Stream

Klassen afgeleid van ‘Stream’

I/O naar bestanden

I/O over netwerkconnecties (TCP-sockets)

Stream-interface naar objecten in het geheugen



Voor LZW-compressie van een stream

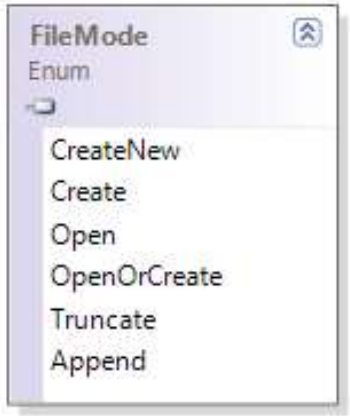
-
-
-

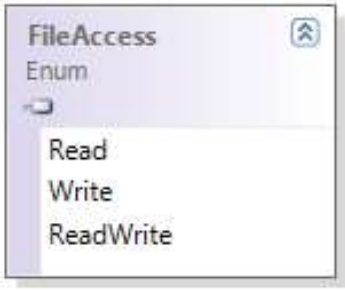
FileStream

constructoren

public **FileStream**(string *path*, System.IO.FileMode *mode*)
Member of System.IO.FileStream

public **FileStream**(string *path*, System.IO.FileMode *mode*, System.IO.FileAccess *access*)
Member of System.IO.FileStream





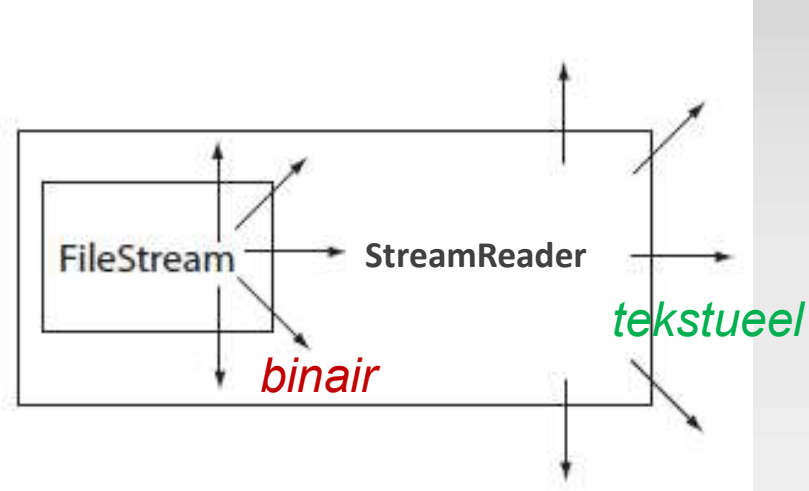
path:

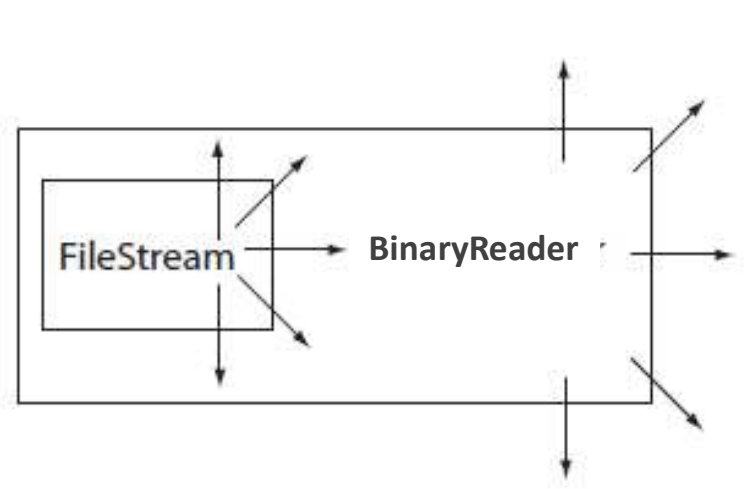
```
"bestandsnaam"
"C:\\doc\\bestandsnaam"
@"C:\doc\bestandsnaam"
```

Low level bestandstoegang, (bijna) atijd gebruikt via 'Wrapper' klassen:

Voor tekst: StreamReader & StreamWriter
Voor binair: BinaryReader & BinaryWriter

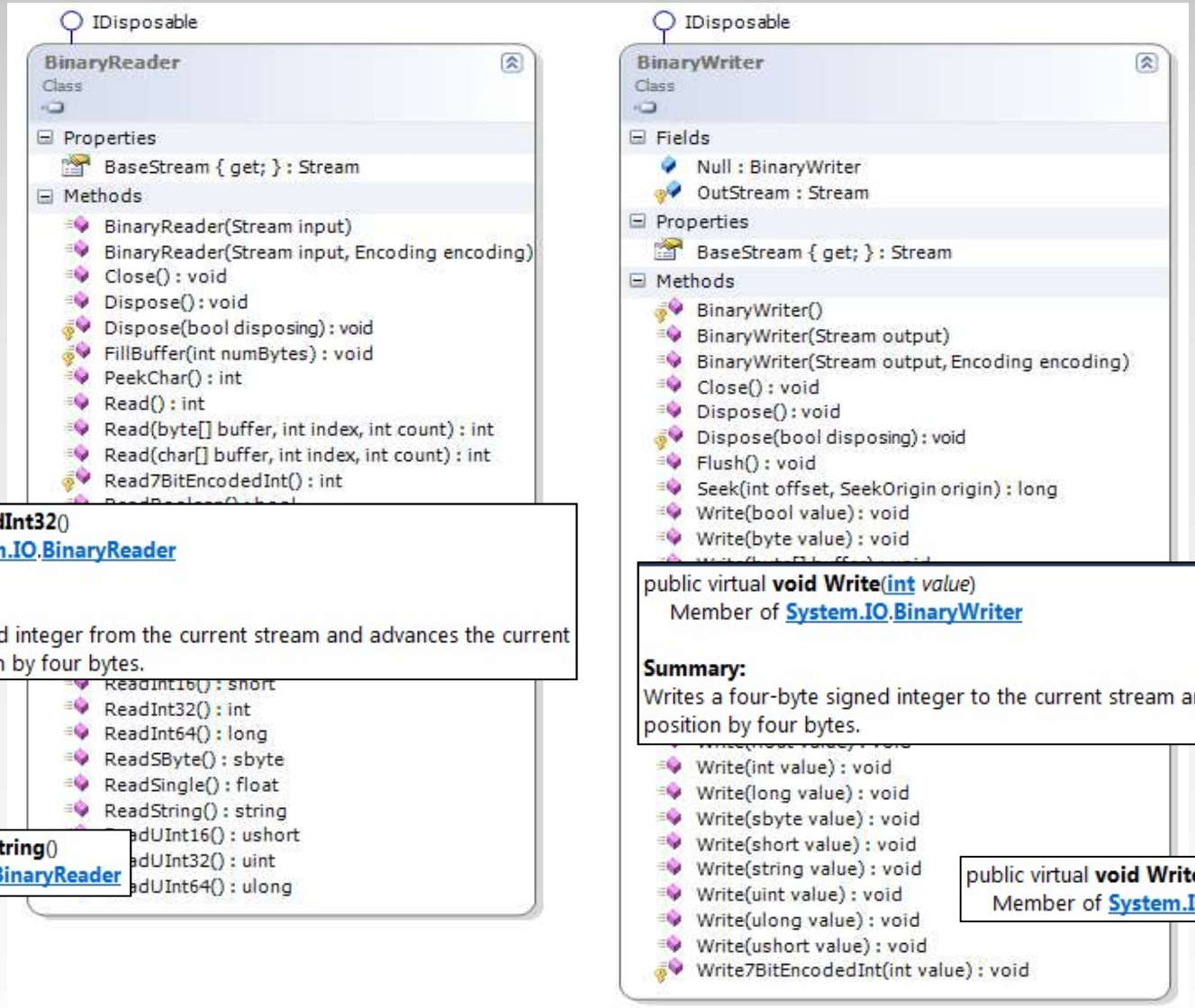
‘Wrapper’ klassen voor Streams





Analoog voor alle soorten streams: FileStream, MemoryStream, NetworkStream

BinaryReader/BinaryWriter



StreamReader

```
public void StreamReaderDemo1()
{
    StreamReader sr = null;
    try
    {
        sr = new StreamReader("TestFile.txt");
        string line;
        while (!sr.EndOfStream)
        {
           line = sr.ReadLine();
           Console.WriteLine(line);
        }
     }
     catch (Exception e)
     {
        Console.WriteLine($"The file could not be read:{e.Message}");
     }
     finally
     {
        if (sr != null) sr.Dispose();
     }
}
```

'Dispose()' is nodig als stream werd aangemaakt!

Using

Streams gebruiken unmanaged resource

⇨ Dispose() is nodig (maar: exceptions!)

Beter:

```
using (StreamWriter file = new StreamWriter(@"D:\test.txt"))
{
    file.WriteLine("Demo");
}
```

⇨ Automatische Dispose

StreamReader (beter)

```
public void StreamReaderDemo2()
{
    try
    {
        // The using statement also disposes the StreamReader.
        using (StreamReader sr = new StreamReader("TestFile.txt"))
        {
            string line;
            while (!sr.EndOfStream)
            {
                line = sr.ReadLine();
                Console.WriteLine(line);
            }
        }
    }
    catch (Exception e)
    {
        Console.WriteLine($"The file could not be read:{e.Message}");
    }
}
```

Using

Steeds gebruiken wanneer een klasse ‘Dispose’ methode bevat:

⇨ Streams

⇨ Bitmap, Graphics, Brush, Pen

⇨ Process

⇨ …

Als ‘Disposable’ object niet binnen één blok gehouden kan worden: Dispose() oproepen als het niet meer gebruikt wordt (anders memory leaks !).

Using

Vanaf .Net Core 3.0 (C# 8.0)

```
using (var file = new StreamWriter(@"D:\test.txt");
file.WriteLine("Demo");
```

Dus: geen block meer nodig na ‘using’.

Object Serialisatie

Object serialisatie

Doel:

Objecten doorgeven via streams.

- ‘persisteren’ naar bestand
- uitwisselen over bv. netwerk, tussen processen

Probleem:

Object is samenstelling van waarden die op zich ook andere objecten kunnen zijn.
Moet omgevormd worden naar een sequentie van waarden.

Object serialisatie

Oplossing:

1. [Serializable] attribuut
2. ‘Formatter’ voor het gewenste formaat:
   - BinaryFormatter binair
   - XMLSerializer XML
   - JavaScriptSerializer JSON

Object serialisatie

Voorbeeld:

```
public enum CourseCategory { lecture, lab, project }

[Serializable]
public class Course
{
    public string Name { get; set; }
    public CourseCategory Category { get; set; }
}

[Serializable]
public class Student
{
    public string Name { get; set; }
    public List<Course> Courses { get; set; }
}
```

Object serialisatie

Voorbeeld:

```
public List<Student> Studenten { get; set; }

public void InitializeSerialisationDemo()
{
   Console.WriteLine("Generating 2 students and 3 courses...\n");
   var Courses = new Course[]
   {
      new Course() { Name = "Math",  Category = CourseCategory.lecture },
      new Course() { Name = "Programming",  Category = CourseCategory.lab },
      new Course() { Name = "Project",  Category = CourseCategory.project }
   };
   Studenten = new List<Student>()
   {
      new Student() { Name = "John", Courses= new List<Course>() {Courses[0], Courses[2]} },
      new Student() { Name = "Alice", Courses= new List<Course>() {Courses[1], Courses[2], Courses[0]} }
   };
}
```

Object serialisatie

Voorbeeld:

```
public void BinarySerializeDemo()
{
    Console.WriteLine("Serializing students to binary file...");
    var formatter = new BinaryFormatter();
    using (var bestand = File.OpenWrite("students.bin"))
    {
        formatter.Serialize(bestand, Studenten);
    }

    Console.WriteLine("Deserializing students from binary file...");
    using (var bestand = File.OpenRead("students.bin"))
    {
        var studentenKopie = (List<Student>)formatter.Deserialize(bestand);
        Console.WriteLine($"    {studentenKopie.Count} students read from file.");
    }
}
```

Object serialisatie to XML

Voorbeeld:

```
public void XMLSerializerDemo()
{
    Console.WriteLine("Serializing students to XML Stream in memory...\n");
    var formatter = new XmlSerializer(typeof(List<Student>));
    using (var stream = new MemoryStream())
    {
        formatter.Serialize(stream, Studenten);
    }
}
```

Opmerkingen:

- Constructor vraagt type dat geserialiseerd zal moeten worden.
- `using System.Xml.Serialization;`

Object serialisatie to JSON

Voorbeeld (met JSON.Net):

```
public void JSONSerializerDemo()
{
    var formatter = new JsonSerializer();
    using (var stream = new MemoryStream())
    {
        formatter.Serialize(new StreamWriter(stream), Studenten);
    }
}
```

Opmerkingen:

- Serialize verwacht `StreamWriter` ipv `Stream`.
- `using Newtonsoft.Json;`

value/reference types

Testvraagjes:

```
int a = 2;
int b = a;
b = 5;
```

Welke waarde heeft a?

Testvraagjes:

```
int a = 2;
int b = a;
b = 5;
```

Welke waarde heeft a?

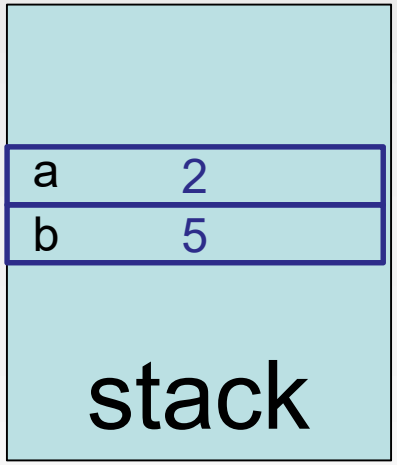
```
var x = new int[]{0,1,2};
var y = x;
y[0] = 5;
```

Welke waarde heeft x[0]?

Verklaring:

```
int a = 2;
int b = a;
b = 5;
```

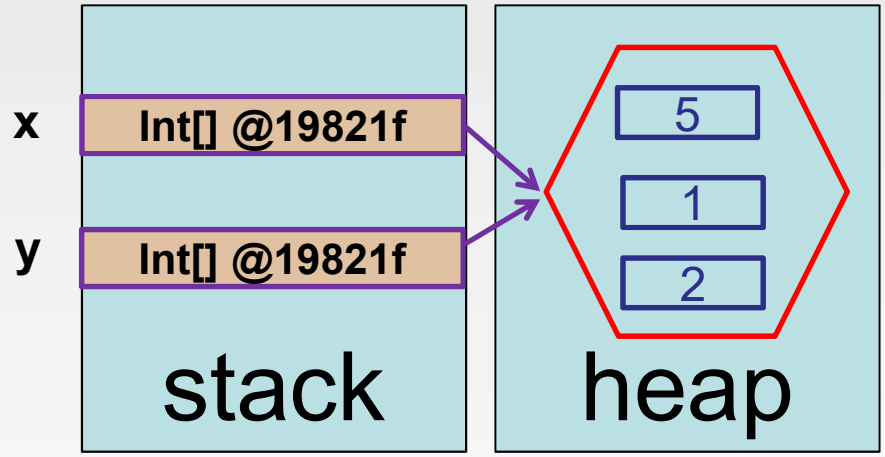
int is een ‘value’ type. `int b = a;` creëert een kopie van a in b
(beiden op de stack)



Verklaring:

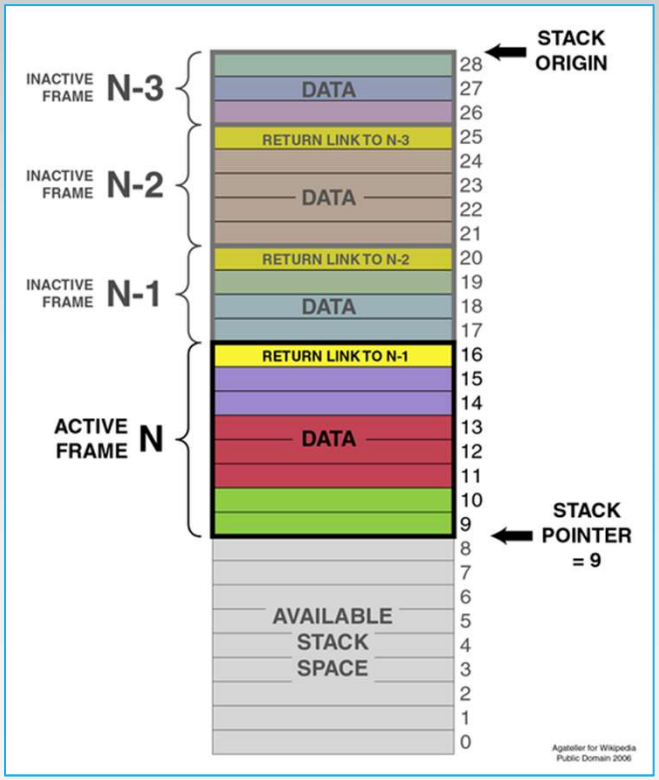
```
var x = new int[]{0,1,2};
var y = x;
y[0] = 5;
```

Int[] is een 'reference' type. `var y = x;` creëert een kopie van x (de referentie) in y.



x en y verwijzen dus naar dezelfde array met waarden op de heap!

Stack - Heap:



'Stack': aaneengesloten deel van het geheugen, snel toegankelijk en beheerd met hardware instructies door CPU ('PUSH', 'POP').

Oproep van een methode:

- Push terugkeeradres
- Reserveer ruimte voor lokale variabelen

⇨ 'StackFrame'

Maar: stack is beperkt in grootte
(op x86 in 64-bit mode 4MB)!

Stack - Heap:

'Heap': de rest van het geheugen dat toegankelijk is voor je toepassing, beheerd via routines van het besturingssysteem ('m_alloc', 'free').
⇨ Dus trager, maar veel meer ruimte beschikbaar.

Praktijk:

'value' types: rechtstreeks op de stack (zijn meestal klein).

'reference' types': ruimte voorzien op de 'Heap' en het adres (de 'referentie') op de stack.

!!!: 'string' is technisch een reference type, maar gedraagt zich als een 'value' type.

Tip: https://www.c-sharpcorner.com/article/C-Sharp-heaping-vs-stacking-in-net-part-i/

Object Cloning

Kopie maken van een object (reference type)

```
public class Course
{
    public string Name { get; set; }
    public CourseCategory Category { get; set; }
}

public class Student
{
    public string Name { get; set; }
    public List<Course> Courses { get; set; }
}
```

?? Kopie van een student object: wat met ‘Courses’ ?

‘Shallow copy’: enkel de referentie wordt gekopieerd.

‘Deep copy’: ook de waarden waarnaar gerefereerd wordt, worden gekopieerd

Kopie maken van een object (reference type)

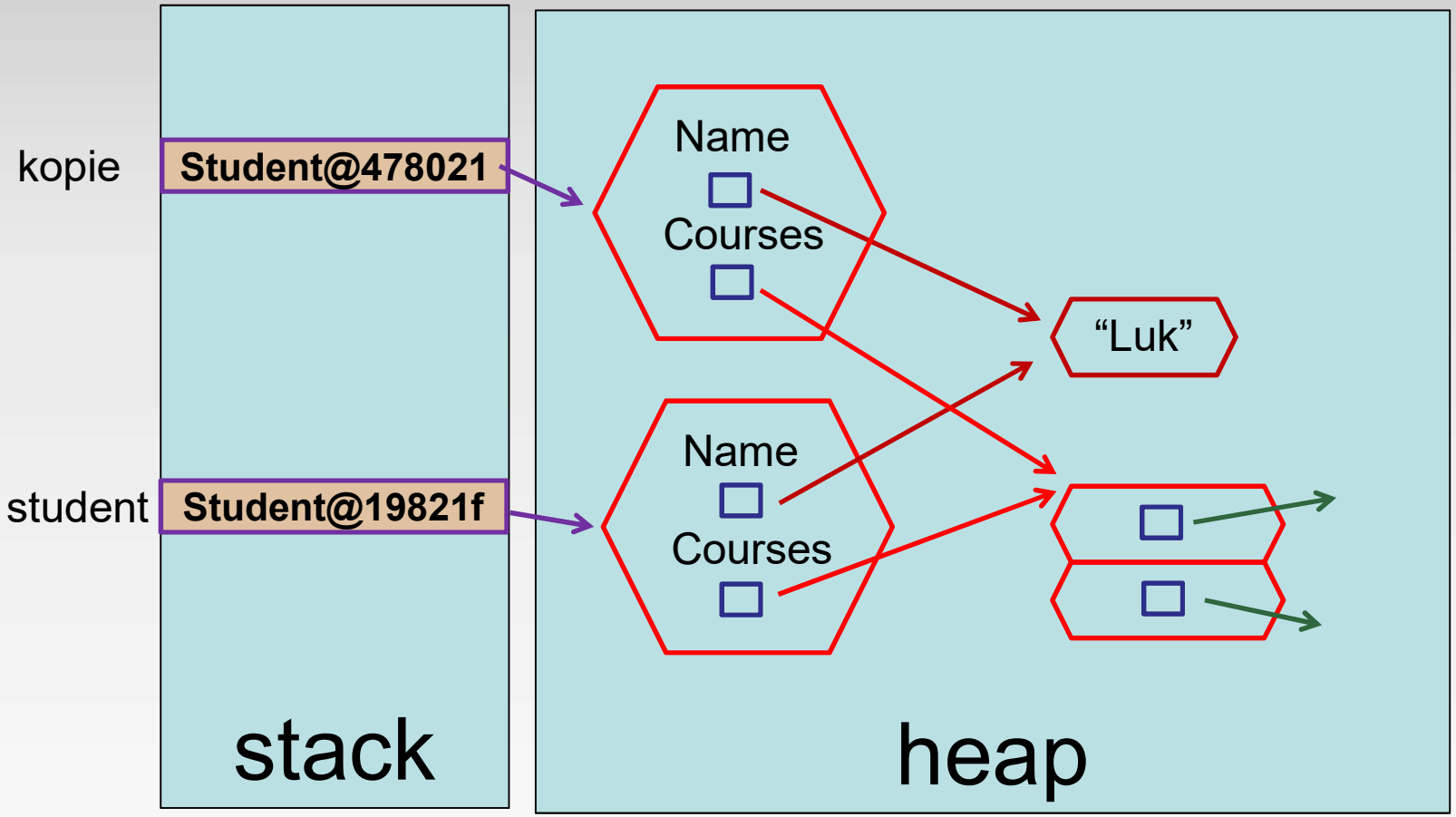
- Object.MemberwiseClone();
- Clone (Icloneable interface)
- ‘Copy’ constructor
- ‘Copy’ method zelf schrijven

Object.MemberwiseClone();

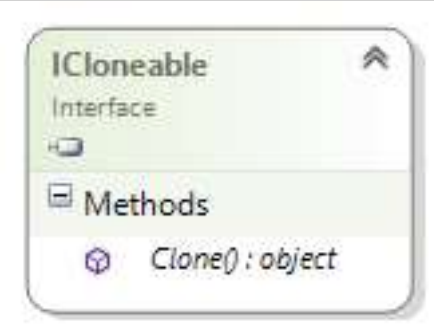
Alle klassen zijn afgeleid van ‘Object’, dus erven ook deze methode.

*The MemberwiseClone method creates **a shallow copy** by creating a new object, and then copying the nonstatic fields of the current object to the new object.*
If a field is a value type, a bit-by-bit copy of the field is performed.
If a field is a reference type, the reference is copied but the referred object is not; therefore, the original object and its clone refer to the same object.

Object.MemberwiseClone();



Icloneable Interface



Een klasse die `ICloneable` implementeert, moet volgende methode declareren:

```
public Object Clone()
```

An implementation of Clone can perform either a deep copy or a shallow copy. In a deep copy, all objects are duplicated; in a shallow copy, only the top-level objects are duplicated and the lower levels contain references.

Because callers of Clone cannot depend on the method performing a predictable cloning operation, we recommend that ICloneable not be implemented in public APIs.

‘Copy’ Constructor

Een standaard patroon is om een ‘copy’ constructor te declareren.
= constructor met als parameter een instance van dezelfde klasse

```
public class Student
{
    public Student( Student source)
    {
        this.Name = source.Name;
        this.Courses = new List<Course>(source.Courses);
    }

    public string Name { get; set; }
    public List<Course> Courses { get; set; }
}
```

Merk op: List<T> implementeert zelf ook een ‘Copy’ constructor

Alternatieve implementatie: zie ‘Deepclone’ voorbeeld op Toledo

‘Copy’ Constructor

Een ‘copy’ constructor declareer je zelf bij de klasse.

Dit kan nog steeds ‘shallow’, ‘deep’ of iets tussenin.

Daarom:

- Documenteer duidelijk welke logica je volgt (documentatie !)
- Maak logische keuzes waar je de waarden kopieert en waar niet

  bv.: ‘wrapper’ klasse voor een bestand. Ook bestand kopiëren?

Je kan ook zelf een ‘Copy’ methode schrijven.
Dit werkt verder gelijkaardig als de copy-constructor.

Hoe een Deep' copy maken

1. Manueel

   MemberwiseClone, en recursief toepassen op alle 'reference type' velden.

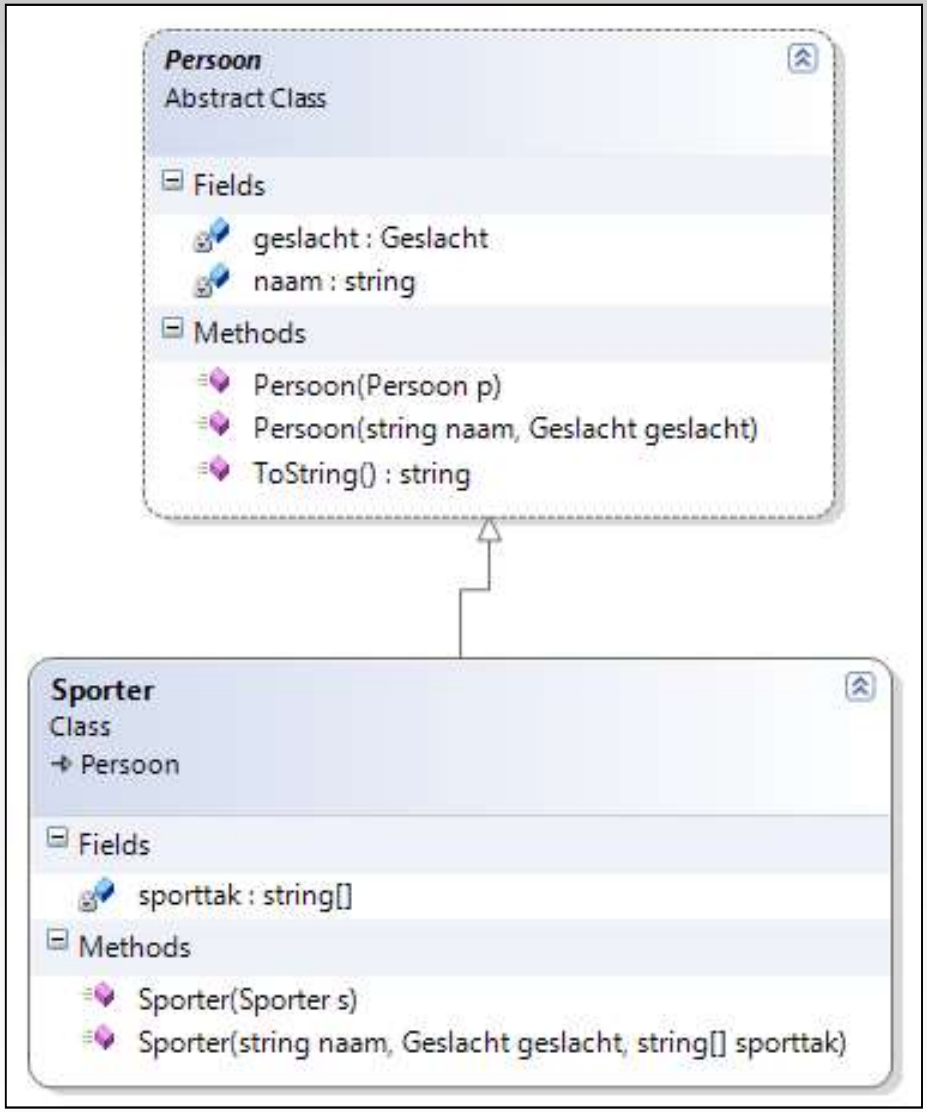
2. via serialisatie

   Het object serialiseren (bv. naar memory) en dan deserialiseren naar het nieuwe object

```
public class Student
{
      ...
      public Student Copy()
      {
          var formatter = new BinaryFormatter();
          using (var stream = new MemoryStream())
          {
              formatter.Serialize(stream, this);
              stream.Seek(0, SeekOrigin.Begin);
              return (Student)formatter.Deserialize(stream);
          }
      }
      ...
```

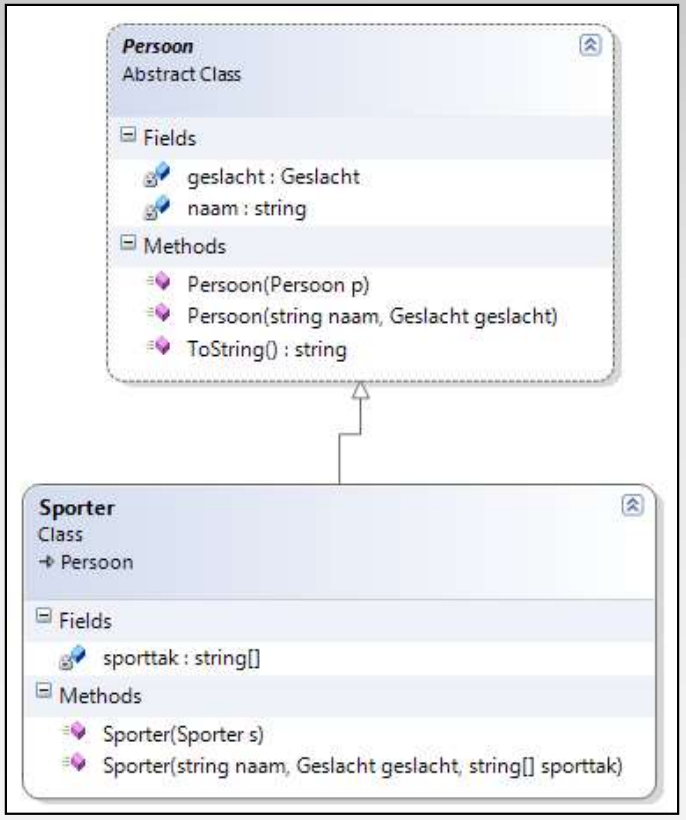
Copy constructoren en inheritance

```
public Sporter(Sporter s)
{
   this.naam = s.naam;
   this.geslacht = s.geslacht;
   this.sporttak = (string[])s.sporttak.clone();
}
```



Copy constructoren en inheritance

```
public Sporter(Sporter s)
{
   this.naam = s.naam;
   this.geslacht = s.geslacht;
   this.sporttak = (string[])s.sporttak.clone();
}
```

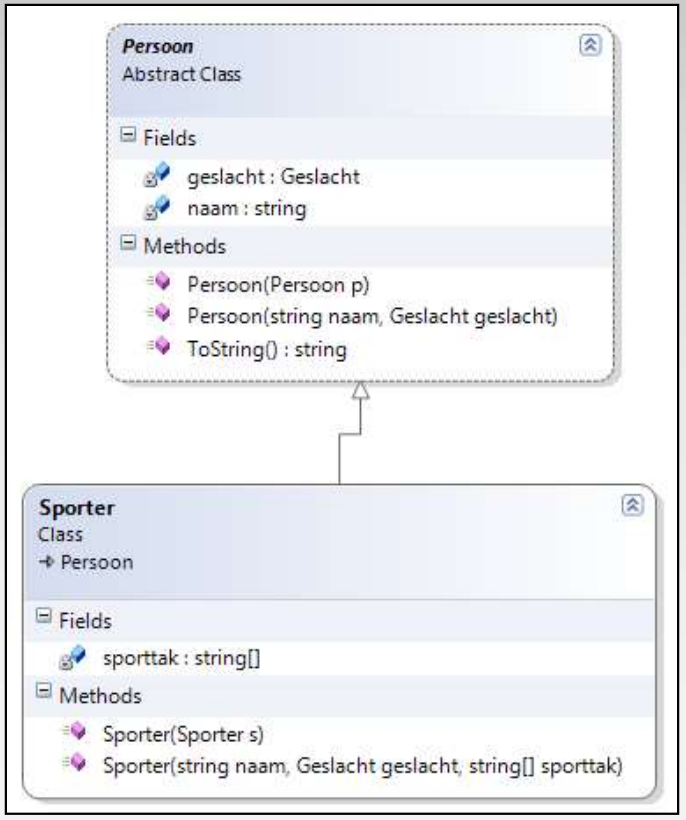


Probleem:

- geslacht en naam mogen niet private zijn
- In constructor van Sporter wordt impliciet de default constructor van Persoon opgeroepen, maar die bestaat niet.

Oplossing

```
public Sporter(Sporter s):base(s)
{
   this.sporttak = new string[s.sporttak.Length];
   this.sporttak = (string[])s.sporttak.clone();
}
```



- geslacht en naam mogen private blijven
- In copy constructor van Sporter: copy constructor van persoon opgeroepen.