C# Essentials - Klassen

Inhoudsopgave

1. Introductie

1.1. Wat is een klasse?

Een klasse is een soort blauwdruk of sjabloon waarmee je objecten kan maken. De klasse beschrijft hoe iets eruitziet en wat het kan doen, maar zelf doet het nog niets. Je kan het vergelijken met een bouwplan: het plan zelf is geen huis, maar het vertelt wel hoe je een huis moet bouwen.

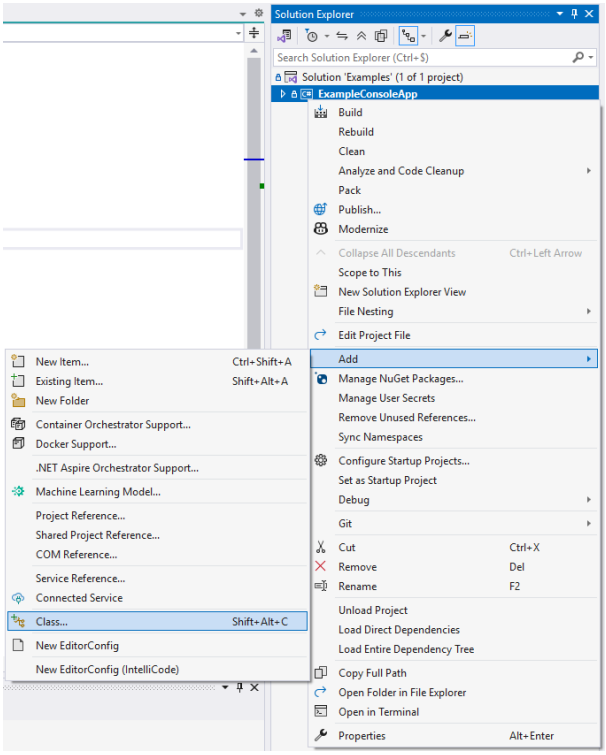
1.2. Waarom gebruiken we klassen?

Wanneer je een iets complexer programma schrijft, wordt het al snel onoverzichtelijk als je alle data en functies zomaar door elkaar zet. Klassen helpen je om:

- code logisch te structureren,
- data en gedrag te groeperen,
- grote programma's in kleinere stukjes op te delen.

Door met klassen te werken, maak je je programma overzichtelijker, beter onderhoudbaar en gemakkelijker uitbreidbaar.

1.3. Klasse toevoegen



In Visual Studio kan je eenvoudig een nieuwe klasse toevoegen aan je project:

1. Rechtsklik in **Solution Explorer** op je project.
2. Kies **Add** → **Class…**.
3. Geef een duidelijke naam in **PascalCase**, bv. `Student.cs`.
4. Klik **Add**. Visual Studio maakt nu een leeg klassenbestand aan.

In grotere projecten worden klassen in aparte mappen geplaatst. Voeg hiervoor eerst een map **Models** toe (**Add** → **New Folder**) en plaats je klasse daar.

1.4. Basisvorm

De basisstructuur van een klasse ziet er als volgt uit:

```
public class Student
{
    // Hier komen velden, eigenschappen en methodes
}
```

- `public` is de **access modifier**, die bepaalt wie deze klasse mag gebruiken.
- `class` geeft aan dat je een klasse definieert.
- `Student` is de naam van de klasse.

1.4.1. Toegankelijkheid

Voor een klasse geef je een **access modifier** op, zoals `public` of `internal`. Dit bepaalt **wie de klasse mag gebruiken** in je project of andere projecten. Zo voorkom je dat klassen per ongeluk op plaatsen gebruikt worden waar ze niet voor bedoeld zijn. Dit maakt je code overzichtelijker en beter beheersbaar.

Access modifier	Toegang
internal	De klasse is **enkel** toegankelijk **binnen hetzelfde project** (standaard).
public	De klasse is **overal** toegankelijk, ook buiten het project.

Als je geen access modifier opgeeft wordt er standaard de `internal` modifier gebruikt!

1.4.2. Naming conventions

- Gebruik **PascalCase** voor de naam van de klasse
- Gebruik een **betekenisvolle naam** die beschrijft wat de klasse voorstelt.
- Gebruik een **Engelse** benaming

Zie PXL Coding Conventions

Voorbeelden:

```
public class Student { }
```

```
public class TemperatureSensor { }

public class InvoiceCalculator { }
```

Vermijd namen zoals `student1`, `mijnKlasse`, `TestClass` of `doeIets`, want die zijn ofwel niet beschrijvend genoeg, of volgen de conventies niet.

1.5. Wat is een instantie?

Een object dat je maakt op basis van een klasse, noemen we ook een **instantie** van die klasse. In C# gebruik je hiervoor het `new` sleutelwoord. Zo geef je aan dat je een nieuw object wil maken.

Bijvoorbeeld:

```
Student student = new Student();

InvoiceCalculator calculator = new InvoiceCalculator();
```

- `student` is een instantie van de klasse `Student`.
- `calculator` is een instantie van de klasse `InvoiceCalculator`

1.6. Klassen vs objecten

Een klasse is dus de definitie van hoe iets moet werken. Maar pas als je een **object** maakt op basis van die klasse, kan je ermee aan de slag gaan.

- Een klasse = het ontwerp
- Een object = de uitvoering

Voorbeeld uit het echte leven:

Klasse	Object
Huisdier	Een kat met de naam Luna
Recept voor spaghetti	Een bord spaghetti op tafel

1.7. Klassen als parameter = byref!

Wanneer je een klasse gebruikt als parameter voor een methode, geef je eigenlijk de **verwijzing naar het object** door. Dat betekent dat de methode rechtstreeks aanpassingen kan maken aan het object dat je meegeeft.

Dit gedrag is anders dan bij eenvoudige types zoals `int` of `bool`, die standaard gekopieerd worden (`by value`).

1.7.1. Voorbeeld

Student.cs

```
class Student
{
    public string FirstName;
    public string LastName;
}
```

Program.cs

```
static void Main(string[] args)
{
    Student student = new Student();
    student.FirstName = "Bob";
    student.LastName = "Smith";

    UpdateStudent(student);

    Console.WriteLine(student.FirstName); // Alice
}

static void UpdateStudent(Student parameter)
{
    parameter.FirstName = "Alice";
}
```

Resultaat:

```
Alice
```

Zoals je ziet: ook al gebruik je **geen** `ref`, wordt het object wél aangepast binnen de methode.

1.8. Klassen als onderdeel van OOP

C# is een objectgeoriënteerde taal (**Object-Oriented Programming**, kortweg OOP). Klassen spelen hierin een centrale rol. De drie belangrijkste principes van OOP zijn:

- **Encapsulatie** – data verbergen en beschermen
- **Overerving** – gedrag hergebruiken
- **Polymorfisme** – verschillende objecten op dezelfde manier gebruiken

In de cursus C# Advanced gaan we dieper in op deze principes.

1.9. Samenvatting

- Een klasse is een blauwdruk waarmee je objecten maakt.
- Een object is een concreet exemplaar van een klasse: dit noem je een **instantie**.
- Klassen helpen je om je programma logisch op te bouwen en herbruikbare onderdelen te maken.
- In C# maak je een object met `new`.

> Een klasse is als een recept, en een object is het gerecht dat je ermee klaarmaakt.

2. Variabelen & Properties

2.1. Velden (instantievariabelen)

Een veld is een variabele die je in een klasse definieert om gegevens op te slaan. Deze hoort bij een object en bevat de toestand van dat object.

Voorbeeld:

```
public class Student
{
    private string _firstName;
    private string _lastName;
    private int _age;
}
```

De variabele `_firstName` is een veld dat gebruikt wordt om een voornaam op te slaan. Deze variabele is echter `private` waardoor je deze enkel kan gebruiken binnen de klasse zelf.

2.1.1. Naming Conventions

- De naam van een instantievariabele start met een underscore `_`, gevolgd door camelCase.

Zie PXL Coding Conventions

2.2. Properties

2.2.1. Wat is een property?

Een property lijkt op een veld, maar biedt extra controle. Je kan zelf bepalen wat er gebeurt wanneer iemand de waarde wil opvragen (**get**) of instellen (**set**).

2.2.2. Syntax

De algemene structuur van een property ziet er zo uit:

```
<access modifier> <type> <Naam>
{
    get { ... }
    set { ... }
}
```

Bijvoorbeeld:

Student.cs

```
public class Student
{
    private string _firstName;
    private int _age;
```

```
    public string FirstName
    {
        get { return _firstName; }
        set { _firstName = value; }
    }

    public int Age
    {
        get { return _age; }
        set { _age = value; }
    }
}
```

- De **property** `FirstName` maakt het mogelijk om `_firstName` te gebruiken **buiten de klasse**.
- De **property** `Age` maakt het mogelijk om `_age` te gebruiken **buiten de klasse**.
- De naam van de property gebruikt **PascalCase**.

2.2.3. Hoe properties gebruiken?

Eens je een property hebt gedefinieerd in een klasse, kan je de `set` gebruiken om een eigenschap van een object een bepaalde waarde te geven:

Program.cs

```
Student student = new Student();     // Maak een nieuwe instantie van de Student
klasse
student.FirstName = "Alice";         // Gebruik de property FirstName om "Alice"
als waarde in te stellen voor de variabele _firstName
student.Age = 19;
```

De `get` kan je gebruiken om de waarde van een eigenschap weer uit te lezen:

Program.cs

```
Console.WriteLine($"{student.FirstName} is {student.Age} jaar"); // Toont: Alice
is 19 jaar
```

In een `set`-blok van een property wordt het sleutelwoord `value` gebruikt om te verwijzen naar de nieuwe waarde die aan de property wordt toegekend.

Als je dit doet:

Program.cs

```
student.FirstName = "Alice";
```

Dan is `"Alice"` de `value` in de set-blok. Je kan ook extra logica toevoegen vóór je de

waarde toekent.

2.2.4. Waarom properties gebruiken?

Eigenschappen geven je meer controle dan gewone velden. Je kan:

- extra logica toevoegen bij lezen of schrijven
- data controleren of beperken
- gegevens afschermen van buitenaf

Bijvoorbeeld:

Student.cs

```
private int _age;

public int Age
{
    get { return _age; }
    set
    {
        if (value >= 0)
        {
            _age = value;
        }
    }
}
```

Hier voorkom je dat iemand een negatieve leeftijd instelt.

2.2.5. Automatische properties

Als je geen extra logica nodig hebt, kan je een **automatische property** gebruiken. De compiler maakt dan zelf het interne veld aan.

```
public string LastName { get; set; }
```

Dit is hetzelfde als:

```
private string _lastName;

public string LastName
{
    get { return _lastName; }
    set { _lastName = value; }
}
```

Gebruik automatische properties alleen als je geen logica nodig hebt in `get` of `set`.

2.2.6. Read-only

Soms hebben we properties die je enkel kan uitlezen (enkel `get`):

```
public string FullName
{
    get { return $"{FirstName} {LastName}"; }
}
```

Verkorte notatie

Als je property enkel een `get` heeft, en die gewoon een waarde of berekening teruggeeft, dan mag je de verkorte `=>`-notatie gebruiken. Dit heet een **expression-bodied property**.

Voorbeeld:

```
public string FullName => $"{FirstName} {LastName}";
```

Je mag dit enkel gebruiken als er **geen `set`-blok** nodig is en je `get`-code uit **exact één regel** bestaat.

2.2.7. Write-only

Omgekeerd komt het ook voor dat je een property enkel kan instellen (enkel `set`), al komt dit in de praktijk zelden voor:

```
private string _secret;

public string Secret
{
    set { _secret = value; }
}
```

2.3. Snippets

In Visual Studio kan je gebruikmaken van **snippets** om sneller properties te schrijven.

- `propfull` + Tab + Tab

  Maakt een **volledige property** met een backing field:

  ```
  private int _myField;

  public int MyProperty
  {
      get { return _myField; }
      set { _myField = value; }
  }
  ```

- `prop` + Tab + Tab

  Maakt een **automatische property** aan:

  ```
  public string MyProperty { get; set; }
  ```

Deze snippets besparen je tijd en zorgen ervoor dat je code altijd de juiste structuur volgt.

2.4. Samenvatting

- Gebruik **velden** om data intern op te slaan.
- Gebruik **properties** om gecontroleerde toegang te geven tot die velden.
- Een property bestaat uit een `get`- en/of `set`-blok.
- Kies voor een **automatische property** als je geen extra logica nodig hebt.
- Naming conventions:
  - `_camelCase` voor private velden
  - `PascalCase` voor properties

3. Methods

3.1. Wat is een method?

In een klasse definieer je methods om aan te geven **wat een object kan doen**.

Een method hoort bij een object van de klasse. Je noemt dit ook een **instantiemethode**. Je kan die method dus enkel gebruiken als je eerst een object gemaakt hebt van die klasse.

3.2. Waarom gebruik je methods?

Je gebruikt methods om:

- logica toe te voegen aan een klasse
- acties uit te voeren die horen bij dat object
- herbruikbare stukjes code te maken

Voorbeelden van method-namen: `Greet()`, `CalculateScore()`, `Reset()`…

3.3. Syntax en opbouw

Een method bestaat uit:

- een toegangsmodifier (`public`, `private`, …)
- een returntype (`void` of een ander type)
- een naam
- optionele parameters tussen haakjes
- een method body met de uit te voeren instructies

```
public class Student
{
    public string StudentNumber { get; set; }
    public string FirstName { get; set; }
    public string LastName { get; set; }
    public string FullName => $"{FirstName} {LastName}";
    public int Age { get; set; }

    public void PrintStudentCard()
    {
        Console.WriteLine($"+------------------------------------------------+")
        Console.WriteLine($"| {StudentNumber, 46} |");
        Console.WriteLine($"| {FullName, 46} |");
        Console.WriteLine($"+------------------------------------------------+")
    }
}
```

3.4. Parameters en returnwaarden

Een method kan parameters ontvangen en/of een waarde teruggeven.

Voorbeeld met parameters:

Student.cs

```
public void AddGrade(string olod, double grade)
{
    _olod = olod;
    _grade = grade;
}
```

Voorbeeld met returnwaarde:

Student.cs

```
public bool IsAdult()
{
    return _age >= 18;
}

public bool HasPassed(double minimumScore)
{
    return _grade >= minimumScore;
}
```

Als een method **geen waarde teruggeeft**, gebruik je `void` als returntype.

3.5. Access modifiers

Met een access modifier bepaal je wie de method mag oproepen. De meest gebruikte modifiers in een klasse zijn:

Modifier	Beschrijving
public	Overal toegankelijk
private	Enkel binnen de klasse zelf oproepbaar

Voorbeeld:

```
public class Student
{
    private void LogMessage(string text)
    {
        Console.WriteLine($"[LOG] {text}");
    }
```

```
    public void Introduce()
    {
        LogMessage("Introduce wordt uitgevoerd");
        Console.WriteLine("Ik ben een student.");
    }
}
```

3.6. Een method gebruiken

Om een method te gebruiken, moet je eerst een object van de klasse aanmaken met `new`. Daarna kan je de method oproepen via dat object.

```
Student student = new Student();
student.AddGrade("C# Essentials", 15);
if(student.HasPassed(12))
{
    Console.WriteLine("Student is ready for C# Advanced!");
}
```

Je kan een method zo vaak oproepen als je wil, telkens met andere parameters.

3.7. Samenvatting

- Een method in een klasse beschrijft een actie die een object kan uitvoeren.
- Methods bestaan uit een naam, parameters, een returntype en een blokje instructies.
- Gebruik `void` als de method geen waarde teruggeeft.
- Je roept een method op via een object van de klasse.
- Met een access modifier bepaal je of een method publiek of privé is.

4. Constructor

4.1. Wat is een constructor?

Een constructor is een speciale methode die automatisch wordt uitgevoerd wanneer je een object maakt met `new`. Je gebruikt een constructor om een object **juist te initialiseren** bij de start.

```
Student student = new Student();
```

Zodra je dit doet, wordt de constructor van `Student` uitgevoerd.

4.2. Waarom gebruiken we een constructor?

Met een constructor kan je:

- verplichte waarden meegeven bij het maken van een object
- velden of properties correct initialiseren
- vermijden dat je object in een ongeldige toestand terechtkomt

4.3. Hoe schrijf je een constructor?

Een constructor:

- heeft altijd dezelfde naam als de klasse
- heeft **geen returntype**
- kan parameters hebben (maar dat is niet verplicht)

Voorbeeld:

```
public class Student
{
    private string _firstName;

    // Constructor
    public Student(string firstName)
    {
        _firstName = firstName;
    }
}
```

Gebruik:

```
Student student = new Student("Alice");
```

4.4. Default constructor

Als je geen constructor zelf schrijft, maakt C# automatisch een **default constructor** aan:

```
public class Student
{
    // Geen constructor gedefinieerd
}

Student student = new Student(); // OK
```

Zodra je zelf een constructor toevoegt, verdwijnt die automatische default constructor. Als je nog een parameterloze constructor wil, moet je die expliciet toevoegen.

4.5. Constructor vs methode

Constructor	**Methode**
Wordt automatisch uitgevoerd bij `new`	Moet je zelf oproepen
Geen returntype	Heeft meestal een returntype (soms `void`)
Zelfde naam als de klasse	Zelfgekozen naam

4.6. Constructor overloading

Je kan meerdere constructors maken met verschillende parameters. Dit heet **overloading**.

```
public class Student
{
    private string _firstName;
    private int _age;

    public Student(string firstName)
    {
        _firstName = firstName;
    }

    public Student(string firstName, int age)
    {
        _firstName = firstName;
        _age = age;
    }
}
```

Gebruik:

```
Student s1 = new Student("Alice");
Student s2 = new Student("Bob", 22);
```

C# kiest automatisch de juiste constructor op basis van de meegegeven parameters.

4.7. Primary constructor

Sinds versie 12 van C# bestaat er een kortere manier om een constructor te schrijven, vooral handig bij eenvoudige klassen. Dit noemen we een **primary constructor**.

In plaats van eerst velden en dan een constructor te schrijven, kan je dit meteen bovenaan de klasse doen.

Voorbeeld:

```
public class Student(string name, int age)
{
    public void PrintInfo()
    {
        Console.WriteLine($"Naam: {name}, Leeftijd: {age}");
    }
}
```

Deze syntax doet exact hetzelfde als:

```
public class Student
{
    private string _name;
    private int _age;

    public Student(string name, int age)
    {
        _name = name;
        _age = age;
    }

    public void PrintInfo()
    {
        Console.WriteLine($"Naam: {name}, Leeftijd: {age}");
    }
}
```

4.7.1. Belangrijk om te weten

- Je kan deze syntax enkel gebruiken bij gewone klassen, niet bij structs of bij overerving met complexe basisconstructors.
- Het is puur syntactische vereenvoudiging: de werking blijft identiek.
- Voor beginners is de klassieke vorm meestal duidelijker. Maar je zal in moderne voorbeelden op het internet vaak deze primary constructors tegenkomen.

Gebruik dit dus alleen als het je code eenvoudiger maakt.

4.8. Constructor chaining

Wil je dat één constructor de andere aanroept? Dat kan met `: this(…)`.

Voorbeeld:

```
public class Student
{
    private string _firstName;
    private int _age;

    public Student(string firstName)
    {
        _firstName = firstName;
    }

    public Student(string firstName, int age) : this(firstName)
    {
        _age = age;
    }
}
```

Hier wordt bij het aanroepen van `new Student("Bob", 22)` automatisch de eerste constructor uitgevoerd, daarna krijgt het `_age`-veld een initiële waarde.

4.9. Object initializers

In C# kan je een object aanmaken met de `new` keyword én tegelijkertijd bepaalde waarden instellen. Dit heet een **object initializer**. Je gebruikt hierbij accolades `{}` om meteen de gewenste properties of velden toe te wijzen.

```
Student student = new Student
{
    FirstName = "Bob",
    LastName = "Smith"
};
```

Dat is een kortere versie van:

```
Student a = new Student();
a.FirstName = "Bob";
a.LastName = "Smith";
```

Je kan dit enkel doen bij public velden of properties die je apart kan instellen. Het werkt ook in combinatie met een constructor:

```
public class Student
```

```
{
    private string FirstName { get; set; }
    private string LastName { get; set; }
    public int Age { get; set; }

    public Student(string firstName, string lastName)
    {
        FirstName = firstName;
        LastName = lastName
    }
}
```

```
Student student = new Student("Bob", "Smith")
{
    Age = 19
};
```

Let op: de constructor wordt eerst uitgevoerd, daarna worden de properties één voor één ingevuld.

4.10. Snippets & Quick actions

In Visual Studio kan je gebruikmaken van **snippets** om sneller constructors te schrijven.

- `ctor` + Tab

  Maakt een nieuwe constructor zonder parameters

  [ctor snippet] | *ctor.gif*

Een andere manier om snel en efficiënt een constructor aan te maken is via **Quick actions**

- `Ctrl + ;` + `Generate constructor from members…`

  Maakt een nieuwe constructor op basis van de bestaande properties

  [quick actions] | *generate-from-members.gif*

4.11. Samenvatting

- Een constructor wordt automatisch uitgevoerd bij `new`.
- De naam is exact gelijk aan die van de klasse.
- Constructors hebben geen returntype.
- Je kan meerdere constructors definiëren (overloading).
- Met `: this(…)` kan je constructor chaining doen.
- Als je geen constructor schrijft, maakt C# er automatisch één aan (default constructor).

5. De object-klasse

5.1. Wat is de object-klasse?

Elke klasse die je maakt in C# *erft* automatisch van `object`. Daardoor heb je altijd een aantal methodes ter beschikking:

- `ToString()`: geeft een tekstuele beschrijving van een object.
- `Equals(object obj)`: vergelijkt of twee objecten gelijk zijn.
- `GetHashCode()`: genereert een unieke waarde (bv. voor gebruik in een `Dictionary`).
- `GetType()`: geeft informatie over het type van het object.

In dit hoofdstuk focussen we op `ToString()` en `Equals()`.

5.2. Voorbeeld: standaard gedrag

Als je geen eigen `ToString()` of `Equals()` schrijft, krijg je standaard het gedrag van `object`.

Student.cs

```
class Student
{
    public string FirstName { get; set; }
    public string LastName { get; set; }
}
```

Program.cs

```
Student student = new Student();
Console.WriteLine(student.ToString());     // ExampleConsoleApp.Student
Console.WriteLine(student.Equals("test")); // False
```

5.3. ToString()

Stel dat je een student met voor- en achternaam hebt. Het standaard `ToString()`-resultaat is niet erg zinvol. Je kan het overschrijven:

```
class Student
{
    public string FirstName { get; set; }
    public string LastName { get; set; }

    public override string ToString()
    {
        return $"Student: {FirstName} {LastName}";
    }
}
```

Gebruik:

Program.cs

```
Student student = new Student
{
    FirstName = "Bob",
    LastName = "Smith"
};
Console.WriteLine(student); // Toont: "Student: Bob Smith"
```

5.4. Equals()

De standaardimplementatie vergelijkt of twee variabelen naar exact hetzelfde object verwijzen:

```
Student a = new Student();
Student b = new Student();
Student c = a;
Console.WriteLine(a.Equals(b)); // False
Console.WriteLine(a.Equals(c)); // True
```

Wil je logischer gedrag (bv. twee studenten met dezelfde naam beschouwen als "gelijk"), dan kan je `Equals()` overschrijven:

```
class Student
{
    public string FirstName { get; set; }
    public string LastName { get; set; }

    public override bool Equals(object obj)
    {
        if (obj is Student other) // Test of obj verschillend is van null en van het type Student is, daarna wordt de waarde "gecast" naar other
        {
            return FirstName == other.FirstName && LastName == other.LastName; // Return true als zowel voor- als achternaam gelijk zijn
        }
        return false;
    }
}
```

Gebruik:

```
Student a = new Student { FirstName = "Bob", LastName = "Smith" };
Student b = new Student { FirstName = "Bob", LastName = "Smith" };
Console.WriteLine(a.Equals(b)); // True
```

5.5. Waarom moet je override gebruiken?

De methodes `ToString()` en `Equals()` bestaan al in de `object`-klasse. Als je deze wil aanpassen voor je eigen klasse, dan moet je dit expliciet aangeven met `override`. Zo weet C# dat je de originele implementatie wil vervangen door je eigen versie.

5.6. Quick actions

Je kan opnieuw gebruik maken van de **Quick actions** in Visual Studio om eenvoudig de `ToString()`- en `Equals()`-methode aan te maken:

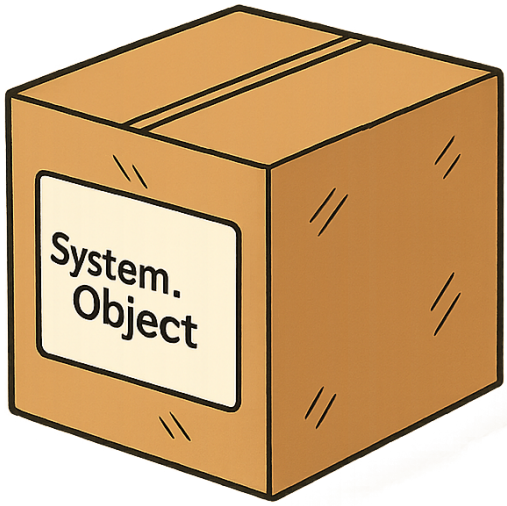
[quick actions] | *generate-overrides.gif*

5.7. Voorbeeld

```
class DefaultProduct
{
    public string Name { get; set; }
    public decimal Price { get; set; }
}

class CustomProduct
{
    public string Name { get; set; }
    public decimal Price { get; set; }

    public override string ToString()
    {
        return $"Product: {Name}, Price: {Price:c}";
    }
}
```

Program.cs

```
CustomProduct laptop = new CustomProduct { Name = "Laptop", Price = 899.99M };
DefaultProduct desktop = new DefaultProduct { Name = "Desktop", Price = 599.99M
};

Console.WriteLine(laptop);  // Product: Laptop, Price: € 899,99
Console.WriteLine(desktop); // ExampleConsoleApp.DefaultProduct
```



Standaard
ToString()

Overschreven
ToString()

5.8. Samenvatting

Elke klasse in C# erft van `object`, waardoor je toegang hebt tot een aantal universele methodes. Door `ToString()` en `Equals()` zelf te overschrijven, maak je je code leesbaarder en logischer. Je past hier ook een eerste vorm van **polymorfisme** toe.

Eventueel kan je ook `GetHashCode()` overschrijven wanneer je objecten gebruikt als sleutel in een `Dictionary`, maar dat behandelen we later.

6. SOLID

6.1. Wat is SOLID?

SOLID is een afkorting die verwijst naar vijf belangrijke principes binnen objectgeoriënteerd programmeren. Deze principes helpen je om je code begrijpelijk, uitbreidbaar en onderhoudbaar te houden.

Letter	Principe
S	Single Responsibility Principle
O	Open/Closed Principle
L	Liskov Substitution Principle
I	Interface Segregation Principle
D	Dependency Inversion Principle

In deze cursus focussen we vooral op het eerste principe. De andere principes bespreken we kort zodat je ze al eens gehoord hebt.

6.2. Single Responsibility Principle (SRP)

Een klasse mag slechts verantwoordelijk zijn voor één ding. Met andere woorden: een klasse mag maar één reden hebben om te veranderen.

6.2.1. Fout voorbeeld

In dit voorbeeld is de klasse `Student` verantwoordelijk voor zowel de data van een student als het opslaan van die data naar een bestand:

```
public class Student
{
    public string Name { get; set; }
    public int Grade { get; set; }

    public void SaveToFile()
    {
        File.WriteAllText("student.txt", $"{Name} - {Grade}");
    }
}
```

Deze klasse heeft dus twee verantwoordelijkheden: gegevens beheren én opslag regelen.

6.2.2. Correct voorbeeld

We splitsen de verantwoordelijkheden op in twee aparte klassen:

```
public class Student
{
    public string Name { get; set; }
    public int Grade { get; set; }
}

public class StudentRepository
{
    public void SaveToFile(Student student)
    {
        File.WriteAllText("student.txt", $"{student.Name} - {student.Grade}");
    }
}
```

De klasse `Student` bevat nu enkel de gegevens, terwijl `StudentRepository` verantwoordelijk is voor het opslaan.

6.3. De andere SOLID-principes (kort)

O – Open/Closed Principle	Klassen moeten uitbreidbaar zijn zonder dat je ze hoeft te wijzigen.
L – Liskov Substitution Principle	Een subklasse moet perfect een superklasse kunnen vervangen zonder fouten.
I – Interface Segregation Principle	Liever meerdere kleine interfaces dan één grote. Klassen hoeven dan alleen te implementeren wat ze echt nodig hebben.
D – Dependency Inversion Principle	Werk met abstracties (bijv. interfaces), niet met concrete klassen.

6.4. Waarom zijn deze principes belangrijk?

Als je deze principes toepast, wordt je programma:

- makkelijker aanpasbaar
- minder foutgevoelig
- eenvoudiger om in team aan te werken

Je hoeft deze principes nog niet allemaal perfect toe te passen, maar hou vooral SRP al in het achterhoofd bij het ontwerpen van je klassen.

6.5. Samenvatting

- SOLID bestaat uit vijf principes die je helpen om betere klassen te schrijven.
- Zorg ervoor dat elke klasse slechts één taak heeft (SRP).