Pijlers van OO - Verdieping

Ludwig Stroobant

28/01/2025

Inhoudstafel

1. Doelstellingen

Na het studeren en maken van de oefeningen van dit hoofdstuk ben je in staat om volgende zaken te herkennen, toe te lichten, te definiëren, toe te passen en te implementeren:

- Overerving en polymorfisme
- Klasse Object, met zijn methodes ToString, Equals en GetType
- Het verschil tussen de methode `Equals` en de `==` operator
- De initialisatie van een object binnen zijn klassenhiërarchie
- Uitbreiding en specialisatie van een klasse
- Klasse en instantie variabelen en klasse en instantie methoden
- Keywords `abstract`, `static` en `readonly`
- Visibiliteit

2. Inleiding

Dit hoofdstuk is een verdieping van 'Pijlers van OO', waarin een introductie werd gegeven van overerving.

Overerving is een mechanisme waarbij software opnieuw wordt gebruikt: nieuwe klassen worden gecreëerd vertrekkende van bestaande klassen, waarbij eigenschappen en gedrag worden geërfd van de superklasse en uitgebreid met nieuwe mogelijkheden, noodzakelijk voor de nieuwe klasse die men de subklasse noemt.

Bij overerving is het zeer belangrijk om onderscheid te maken tussen de verschillende types enerzijds en hun onderlinge relaties anderzijds.

Laat ons een aantal zaken opfrissen.

3. Wat is een 'type'?

In C# is een **type** een fundamenteel concept dat bepaalt welke soort data een variabele kan bevatten, welke operaties op die data kunnen worden uitgevoerd en hoe de data in het geheugen wordt opgeslagen.

Types zijn essentieel voor de werking van C#, omdat het een sterk getypeerde taal is. Dit betekent dat elk stuk data (variabelen, constanten, expressies, enz.) een specifiek type heeft, en de compiler controleert of alle operaties die je uitvoert, geldig zijn voor dat type.

3.1. Wat is een type in C#?

Een type in C# definieert:

1. **De soort data** die een variabele kan bevatten (bijvoorbeeld een getal, tekst, een object, enz.).

2. **De grootte van de data** (bijvoorbeeld een `int` neemt 4 bytes in beslag, een `long` 8 bytes).
3. **De operaties die kunnen worden uitgevoerd** op de data (bijvoorbeeld optellen, vergelijken, methoden aanroepen).
4. **De structuur van de data** (bijvoorbeeld een klasse heeft velden en methoden, een array bevat een reeks elementen).

3.2. Soorten types in C#

In C# zijn er twee hoofdsoorten types:

1. **Waarde-types (Value Types)**
2. **Referentie-types (Reference Types)**

3.2.1. Waarde-types (Value Types)

Waarde-types slaan de **werkelijke waarde** van de data op. Ze worden direct in het geheugen opgeslagen (meestal op de stack, tenzij ze deel uitmaken van een object op de heap).

Voorbeelden van waarde-types zijn:

- **Primitieve types**: `int`, `float`, `double`, `bool`, `char`, enz.
- **Structs**: Zelfgedefinieerde waarde-types, zoals `DateTime` of een eigen `struct`.

```
int number = 42; // 'number' is een waarde-type en bevat de waarde 42
```

Kenmerken van waarde-types:

- Ze worden gekopieerd wanneer ze worden toegewezen aan een andere variabele of doorgegeven als argument.
- Ze zijn meestal lichtgewicht en snel.
- Ze kunnen niet `null` zijn, tenzij je `Nullable<T>` gebruikt (bijvoorbeeld `int?`).

3.2.2. Referentie-types (Reference Types)

Referentie-types slaan een **referentie** (een adres in het geheugen) op naar de locatie waar de data zich bevindt. De data zelf wordt opgeslagen op de heap.

Voorbeelden van referentie-types zijn:

- **Klassen**: `string`, `object`, en zelfgedefinieerde klassen.
- **Arrays**: `int[]`, `string[]`, enz.
- **Interfaces**: `IEnumerable`, `IDisposable`, enz.
- **Delegates**: `Action`, `Func`, enz.

```
string name = "Alice"; // 'name' is een referentie-type en verwijst naar een string op de heap
```

Kenmerken van referentie-types:

- De referentie wordt gekopieerd wanneer ze worden toegewezen aan een andere variabele, niet het object waarnaar ze verwijzen. Beide variabelen bevatten nu een referentie naar hetzelfde object.
- Ze kunnen `null` zijn, wat betekent dat ze naar niets verwijzen.
- Ze zijn geschikt voor complexe data en objecten.

3.3. Voorbeelden van types in C#

Type	Categorie	Voorbeeld	Beschrijving
`int`	Waarde-type	`int age = 30;`	Een 32-bit geheel getal.
`double`	Waarde-type	`double pi = 3.14;`	Een 64-bit floating-point getal.
`bool`	Waarde-type	`bool isActive = true;`	Een Booleaanse waarde (`true` of `false`).
`char`	Waarde-type	`char letter = 'A';`	Een enkel Unicode-teken.
`string`	Referentie-type	`string name = "Bob";`	Een reeks Unicode-tekens (een klasse in C#).
`int[]`	Referentie-type	`int[] numbers = { 1, 2, 3 };`	Een array van gehele getallen.
`List<int>`	Referentie-type	`List<int> list = new List<int>();`	Een lijst van gehele getallen (een klasse in C#).
`DateTime`	Waarde-type (struct)	`DateTime now = DateTime.Now;`	Een datum- en tijdwaarde.

3.4. Waarom zijn types belangrijk?

1. **Typeveiligheid**: De compiler controleert of je alleen geldige operaties uitvoert op een variabele. Bijvoorbeeld, je kunt geen tekst toevoegen aan een getal zonder expliciete conversie.

```
int number = 10;
string text = "20";
// int result = number + text; // Dit geeft een compileerfout!
```

---

1. **Geheugenbeheer**: Types bepalen hoe data in het geheugen wordt opgeslagen (stack voor waarde-types, heap voor referentie-types).
2. **Code-organisatie**: Types helpen bij het modelleren van de echte wereld in code. Bijvoorbeeld, een klasse `Person` kan eigenschappen zoals `Name` en `Age` hebben.
3. **Herbruikbaarheid**: Door types te definiëren (bijvoorbeeld via klassen of interfaces), kun je code herbruiken en uitbreiden.

3.5. Zelf een type definiëren

Je kunt je eigen types definiëren in C# met behulp van:

- **Klassen**: Voor referentie-types.

```
public class Person {
    public string Name { get; set; }
    public int Age { get; set; }
}
```

- **Structs**: Voor waarde-types.

```
public struct Point {
    public int X { get; set; }
    public int Y { get; set; }
}
```

- **Enums**: Voor een vaste set benoemde waarden.

```
public enum DayOfWeek {
    Monday, Tuesday, Wednesday, Thursday, Friday, Saturday, Sunday
}
```

3.6. Samenvatting

Een type in C# is een definitie van de soort data die een variabele kan bevatten, hoe die data wordt opgeslagen en welke operaties erop kunnen worden uitgevoerd.

Er zijn twee hoofdsoorten types: **waarde-types** (directe opslag van data) en **referentie-types** (opslag van een verwijzing naar data).

> In C#, there are two main categories of types: value types and reference types. Value types are stored directly in memory, while reference types store a reference (or pointer) to the memory location where the data is stored.

> — DeepSeek

Types zijn essentieel voor typeveiligheid, geheugenbeheer en het modelleren van complexe systemen in code.

4. Wat is een klasse?

Een **klasse** in C# is een blauwdruk of sjabloon voor het maken van objecten. Het definieert de structuur en het gedrag van een object door middel van **velden** (data) en **methoden** (functionaliteit).

Klassen zijn een fundamenteel onderdeel van objectgeoriënteerd programmeren (OOP) en worden gebruikt om objecten of entiteiten uit de echte wereld of abstracte concepten te modelleren.

4.1. Wat is een klasse in C#?

Een klasse is een **referentietype** in C#. Dit betekent dat wanneer je een object van een klasse maakt, het object op de heap wordt opgeslagen en je werkt met een referentie (een adres in het geheugen) naar dat object.

Een klasse kan de volgende elementen bevatten:

1. **Attributen of velden (Fields)**: Variabelen die de staat (data) van een object bijhouden.
2. **Eigenschappen (Properties)**: Een manier om gecontroleerde toegang tot de velden te bieden.
3. **Methoden (Methods)**: Functies die het gedrag van een object definiëren.
4. **Constructors**: Speciale methoden die worden gebruikt om een object te initialiseren.
5. **Events**: Gebeurtenissen die een object kan activeren.
6. **Indexers**: Een manier om een object te gebruiken als een array.
7. **Operators**: Een manier om aangepaste gedragingen voor operatoren (zoals `+`, `-`, enz.) te definiëren.

4.2. Syntax van een klasse

Dit is de basisstructuur van een klasse in C#:

```
public class ClassName
{
    // Velden (Fields)
    private string _name;

    // Eigenschappen (Properties)
    public string Name
    {
        get { return _name; }
        set { _name = value; }
```

```
    }

    // Constructor
    public ClassName(string name)
    {
        _name = name;
    }

    // Methoden (Methods)
    public void DisplayName()
    {
        Console.WriteLine("Name: " + _name);
    }
}
```

4.3. Voorbeeld van een klasse

Laten we een eenvoudige klasse `Person` maken die een persoon modelleert:

```
public class Person
{
    // Velden
    private string _name;
    private int _age;

    // Eigenschappen
    public string Name
    {
        get { return _name; }
        set { _name = value; }
    }

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

    // Constructor
    public Person(string name, int age)
    {
        _name = name;
        _age = age;
    }

    // Methode
    public void Introduce()
    {
        Console.WriteLine($"Hello, my name is {_name} and I am {_age} years old.");
```

```
    }
}
```

4.4. Een object maken van een klasse

Om een klasse te gebruiken, moet je een **object** (ook wel een **instantie** genoemd) van die klasse maken. Dit doe je met het `new`-keyword: zo krijg je een een referentie terug naar het pas aangemaakte object.

```
// Maak een object van de klasse Person
Person person1 = new Person("Alice", 30);

// Gebruik de methoden en eigenschappen van het object
person1.Introduce(); // Output: Hello, my name is Alice and I am 30 years old.
person1.Age = 31;    // Wijzig de leeftijd
person1.Introduce(); // Output: Hello, my name is Alice and I am 31 years old.
```

> There may be many references to the same object. Most objects have state, stored in the fields of objects that are instances of classes or in the variables that are the components of an array object. If two variables contain references to the same object, the state of the object can be modified using one variable's reference to the object, and then the altered state can be observed through the reference in the other variable.
>
> — The Java Language Specification

Je kan één, twee, drie of meer referentievariabelen hebben die allen een referentie bevatten naar hetzelfde object.

Wat als er geen enkele referentievariabele meer is die naar een bepaald object verwijst? Op dit moment mag het object uit het geheugen verwijderd worden. Dit is de taak van de `Garbage Collector`: een process dat in de achtergrond draait met specifiek de taak van het opruimen van objecten. Moest dit niet gebeuren dan zou het geheugen van het systeem vol raken waardoor het onbruikbaar wordt.

4.5. Belangrijke concepten bij klassen

1. **Encapsulatie (Inkapseling)**:
    - Het verbergen van de interne details van een klasse en het blootstellen van een gecontroleerde interface (via eigenschappen en methoden).
    - Bijvoorbeeld: Het veld `_name` is privé, maar toegankelijk via de eigenschap `Name`.
2. **Constructors**:
    - Worden aangeroepen wanneer een object wordt gemaakt.

- Ze worden gebruikt om de beginwaarden van de velden in te stellen.
- Als je geen constructor definieert, wordt een standaardconstructor (zonder parameters) automatisch gegenereerd.

3. **Eigenschappen (Properties)**:
   - Een manier om gecontroleerde toegang tot de velden van een klasse te bieden.
   - Ze kunnen `get`- en `set`-accessors hebben om waarden op te halen en in te stellen.
4. **Methoden**:
   - Functies die het gedrag van een klasse definiëren.
   - Ze kunnen parameters accepteren en waarden retourneren.
5. **Static Members**:
   - Velden, eigenschappen of methoden die bij de klasse zelf horen in plaats van bij een specifiek object.
   - Je kunt ze aanroepen zonder een object te maken.

```
public class MathUtility
{
    public static int Add(int a, int b)
    {
        return a + b;
    }
}

int result = MathUtility.Add(5, 3); // Roep de statische methode aan
```

4.6. Waarom klassen gebruiken?

1. **Modulariteit**: Klassen helpen om code te organiseren in logische eenheden.
2. **Herbruikbaarheid**: Je kunt een klasse hergebruiken in verschillende delen van je programma.
3. **Uitbreidbaarheid**: Je kunt bestaande klassen uitbreiden via overerving.
4. **Abstraheren van complexiteit**: Klassen verbergen de interne details en bieden een eenvoudige interface aan de gebruiker.

4.7. Verschil tussen een klasse en een struct

- **Klasse**: Een referentietype dat op de heap wordt opgeslagen. Geschikt voor complexe objecten.
- **Struct**: Een waardetype dat op de stack wordt opgeslagen. Geschikt voor lichtgewicht objecten.

```
public struct Point
{
    public int X { get; set; }
    public int Y { get; set; }
```

```
}
```

4.8. Samenvatting

Een klasse in C# is een blauwdruk voor het maken van objecten. Het definieert de data (velden) en functionaliteit (methoden) die een object heeft. Klassen zijn een krachtig hulpmiddel voor het modelleren van objecten of entiteiten uit de echte wereld en het organiseren van code in een objectgeoriënteerde stijl.

5. Pijlers van objectgeoriënteerd programmeren (OOP)

De vier pijlers van OOP zijn: **Abstractie**, **Inkapseling**, **Overerving** en **Polymorfisme**.

5.1. Abstractie

Abstractie betekent dat je de essentiële details van een object gaat modelleren en zo de complexiteit verbergt. Het gaat om het creëren van een eenvoudige weergave van iets complex.

5.1.1. Voorbeeld:

Stel je voor dat je een auto modelleert. Je hoeft niet te weten hoe de motor werkt om de auto te besturen. Je hebt alleen toegang tot de essentie: het stuur, de pedalen en de versnellingspook.

```
public class Car
{
    public void Start() => Console.WriteLine("Car started.");
    public void Stop() => Console.WriteLine("Car stopped.");
}

// Gebruik van de klasse
Car myCar = new Car();
myCar.Start(); // Je hoeft niet te weten hoe de motor werkt, alleen hoe je de auto
start.
```

5.2. Inkapseling

Inkapseling betekent het verbergen van de interne details van een object en het blootstellen van een gecontroleerde interface. Dit wordt vaak gedaan met behulp van `private` velden en `public` eigenschappen of methoden.

Voorbeeld

> Een bankrekening heeft een saldo, maar je wilt niet dat iedereen direct het saldo kan wijzigen. In plaats daarvan bied je methoden aan om geld op te nemen of te storten.

```
public class BankAccount
{
    private double _balance; // Privé veld

    public void Deposit(double amount)
    {
        if (amount > 0)
            _balance += amount;
    }

    public void Withdraw(double amount)
    {
        if (amount > 0 && amount <= _balance)
            _balance -= amount;
    }

    public double GetBalance() => _balance;
}

// Gebruik van de klasse
BankAccount account = new BankAccount();
account.Deposit(100);
account.Withdraw(50);
Console.WriteLine(account.GetBalance()); // Output: 50
```

5.3. Overerving

Overerving betekent dat een klasse eigenschappen en gedrag kan overnemen van een andere klasse. Dit bevordert codehergebruik en hiërarchische organisatie.

Voorbeeld

Een `Huisdier` klasse heeft algemene eigenschappen en gedrag, en een `Hond` klasse erft deze over en voegt specifieke functionaliteit toe.

```
public class Huisdier
{
    public string Naam { get; set; }

    public void StelJeVoor() => Console.WriteLine($"Ik heet {Naam}.")
}

public class Hond : Huisdier
{
    public void Kwispel() => Console.WriteLine($"{Naam} kwispelt.");
}

// Gebruik van de klasse
Hond hond = new();
```

```
hond.Naam = "Buddy";
hond.StelJeVoor();  // Overgenomen van Huisdier
hond.Kwispel(); // Specifiek voor Hond
```

5.4. Polymorfisme

Polymorfisme betekent dat een object zich op verschillende manieren kan gedragen, afhankelijk van de context. Dit kan worden bereikt via methoden die worden overschreven in afgeleide klassen of via interfaces.

Voorbeeld

Een `Shape` klasse heeft een methode `Draw()`, maar elke specifieke vorm (zoals `Circle` of `Rectangle`) implementeert deze methode op zijn eigen manier.

```
public class Shape
{
    public virtual void Draw() => Console.WriteLine("Drawing a shape.");
}

public class Circle : Shape
{
    public override void Draw() => Console.WriteLine("Drawing a circle.");
}

public class Rectangle : Shape
{
    public override void Draw() => Console.WriteLine("Drawing a rectangle.");
}

// Gebruik van polymorfisme
Shape shape1 = new Circle();
Shape shape2 = new Rectangle();

shape1.Draw(); // Output: Drawing a circle.
shape2.Draw(); // Output: Drawing a rectangle.
```

5.5. Samenvatting van de vier pijlers

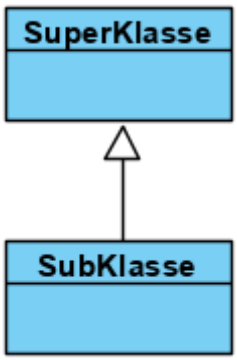
Abstractie	**Verberg complexiteit en toon alleen essentiële details.**	**Een `Car` klasse met methoden `Start()` en `Stop()`.**
Inkapseling	Verberg interne details en bied een gecontroleerde interface.	Een `BankAccount` klasse met privé saldo en publieke methoden voor transacties.
Overerving	Een klasse erft eigenschappen en gedrag van een andere klasse.	Een `Dog` klasse die overerft van `Animal` en specifiek gedrag toevoegt.

Abstractie	**Verberg complexiteit en toon alleen essentiële details.**	**Een `Car` klasse met methoden `Start()` en `Stop()`.**
Polymorfisme	Een object kan zich op verschillende manieren gedragen, afhankelijk van de context.	Een `Shape` klasse met een `Draw()` methode die wordt overschreven in `Circle`.

6. Overerving in detail

Overerving betekent dat we bij de declaratie van een klasse eigenschappen en gedrag kunnen laten overnemen van een andere klasse. Dit overgeërfde deel kunnen we uitbreiden of specialiseren.

De klasse waarvan eigenschappen en gedrag wordt overgeërfd is de ***superklasse***, de klasse die de eigenschappen en het gedrag overneemt is de ***subklasse***.



Klassen in C# ondersteunen **enkelvoudige overerving**, waarbij elke klasse slechts één enkele superklasse kan hebben.

Elke klasse erft de eigenschappen en het gedrag van zijn superklasse en van andere klassen hogerop in de klassenhiërarchie.

6.1. Doel van overerving

Overerving is het opzetten van een gestructureerde hiërarchie met als doel het halen van een **betere maintenance factor**:

- Meer/eenvoudiger overzicht in structuur en gebruik
- Generalisatie en specialisatie
- Hergebruik van code

(Abstracte) klassen bovenaan de hiërarchie leggen eigenschappen en gedrag vast dat wordt overgeërfd door klassen lager in de hiërarchie.

- Een subklasse erft van zijn superklasse alle eigenschappen en gedrag.

- Een subklasse kan maar één superklasse hebben in C#.
- Een subklasse kan op zijn beurt een superklasse zijn.
- Een superklasse kan meerdere directe subklassen hebben.
- De hoogste klasse in elke klassenhiërarchie in C# is de klasse Object.

6.2. Wat erft de subklasse?

Een klasse erft ***eigenschappen*** en ***gedrag*** van zijn superklasse, en de klassen erboven in de hiërarchie tot uiteindelijk de klasse Object.

Een constructor is geen methode en wordt niet overgeërfd door de subklasse!!!

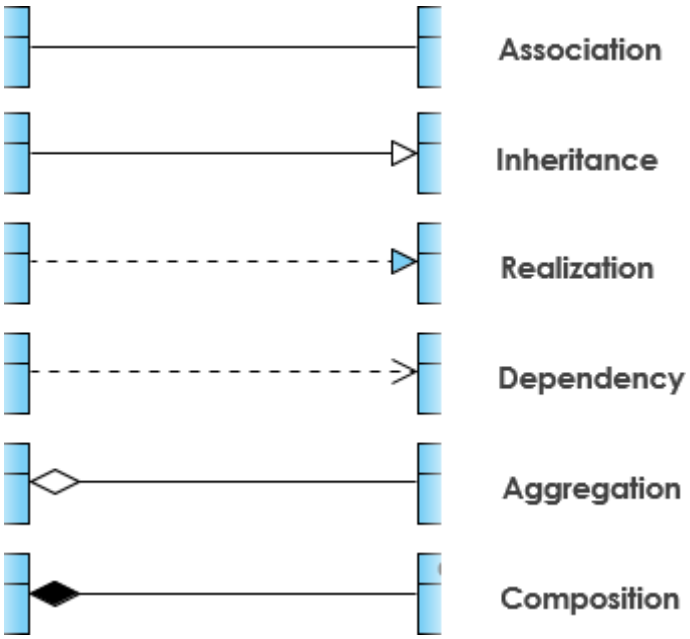
Een constructor heeft een rechtstreekse link met het gedeclareerde klasse-referentie-type. De naam van de constructor is de naam van de klasse en wordt gebruikt om te differentiëren tussen verschillende referentie types.

> Constructors are similar to methods, but cannot be invoked directly by a method call; they are used to initialize new class instances. Like methods, they may be overloaded.
>
> — The Java Language Specification

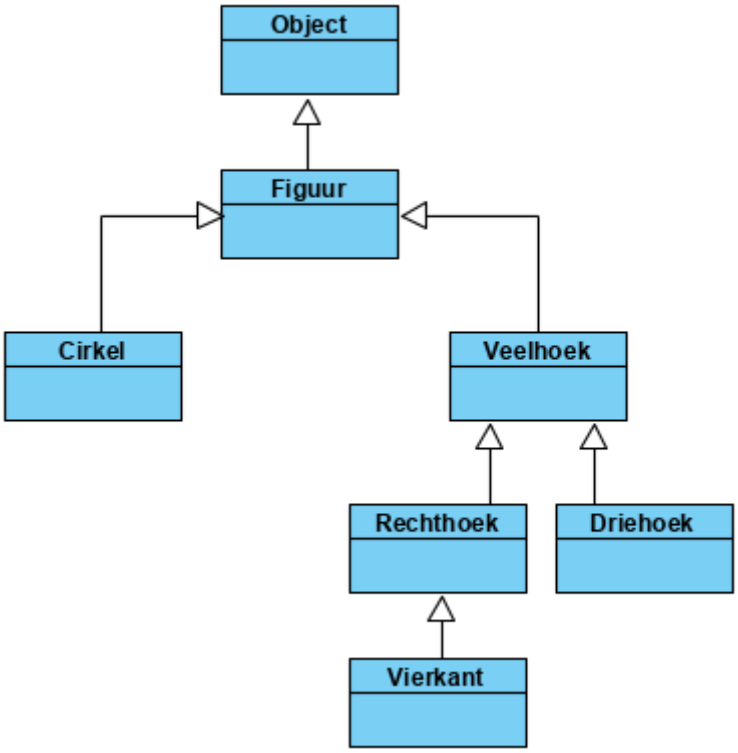
6.3. "IS EEN" relatie

Overerving (= Inheritance in het Engels) resulteert in een **'IS EEN'** relatie

Bij ontwerp, in een UML, wordt deze 'IS EEN' relatie weergegeven met een specifieke pijl, wat helpt om de relatie te visualiseren. Hieronder een overzicht van de mogelijke relaties tussen verschillende types. De "IS EEN" relatie duidt op een overerving ('Inheritance' in het Engels).



6.4. Voorbeeld van een hiërarchie



6.4.1. Bottom to top

- Vierkant *IS EEN* Rechthoek: Let op de richting van de pijl. Een Vierkant is 100% zeker een Rechthoek.
- Rechthoek *IS EEN* Veelhoek → Vierkant *IS EEN* Veelhoek
- Veelhoek *IS EEN* Figuur → Rechthoek *IS EEN* Figuur → …
- Cirkel *IS EEN* Figuur
- Figuur *IS EEN* Object → Veelhoek *IS EEN* Object → …

6.4.2. Top to bottom

- Object *KAN EEN* Figuur zijn
- Figuur *KAN EEN* Cirkel zijn
- Figuur *KAN EEN* Veelhoek zijn: Let op de richting van de pijl. Een Figuur kan een Cirkel zijn, maar ook een Veelhoek…
- Veelhoek *KAN EEN* Rechthoek zijn
- Rechthoek *KAN EEN* Vierkant zijn

Volg je de pijl van de *IS EEN* relatie in de tegengestelde richting, dan kan je dit verwoorden met een *KAN EEN* verband. Dit geeft rechtstreeks aanleiding tot ***polymorfisme***.

Een variabele met als referentie type Figuur kan verschillende vormen aannemen. M.a.w. een

referentie variabele van het type Figuur kan refereren naar een object van het type Cirkel maar ook naar een object van het type Veelhoek. Van zowel Cirkel als Veelhoek ben je zeker dat ze alle eigenschappen en gedrag van Figuur overnemen want: een Cirkel *IS EEN* Figuur en ook een Veelhoek *IS EEN* Figuur.

Polymorfisme laat toe dat variabelen van een klasse type een referentie kunnen bevatten naar een instantie van die klasse of naar een instantie van eender welke subklasse van die klasse.

Polymorfisme laat zo het declareren en gebruik van nieuwe types toe die beschikken over reeds bestaande methodes.

6.5. Type verifiëren: de keywords `is` en `as`

Referentie variabelen kunnen verwijzen naar objecten van verschillende types (polymorfisme). In een bepaalde context kan het nodig zijn om te verifiëren of een object van een bepaald referentie type is alvorens het te gebruiken. Dit kan je doen door gebruik te maken van het keyword `is` of `as`.

Het `is` keyword gaat na of het object wel degelijk een instantie is van een bepaald type en geeft een `bool` terug.

Het `is` keyword dien je altijd te gebruiken alvorens je een `cast` uitvoert naar een specifiek type. Het `as` keyword combineert het `is` keyword en een `cast`, en zal een referentie teruggeven van het gecaste type, of `null` indien de cast mislukt.

```
1    Rechthoek rechthoek = new Rechthoek();
2    bool a = rechthoek is Rechthoek; ①
3    bool b = rechthoek is Veelhoek; ②
4    bool c = rechthoek is Vierkant; ③
```

1. a zal `true` zijn (rechthoek IS EEN Rechthoek)
2. b zal `true` zijn (rechthoek IS EEN Veelhoek)
3. c zal `false` zijn (rechthoek KAN EEN Vierkant zijn, maar is daarom nog geen Vierkant. In deze context is rechthoek geen Vierkant)

7. De moederklasse *Object*

In de C# programmeertaal is de `Object`-klasse een speciale klasse die de basis vormt voor alle andere klassen. In eenvoudige woorden:

Basis van alles

Stel je voor dat je een huis wilt bouwen. Voordat je begint met het ontwerpen van de kamers, deuren en ramen, moet je een stevige fundering hebben. De Object-klasse is die stevige fundering in C#. Alle andere klassen, zoals Auto, Bankrekening of zelfs jouw eigen aangepaste klassen, erven stiekem van deze `Object`-klasse.

Standaardfuncties

De `Object`-klasse bevat enkele standaardfuncties die elk object in C# kan gebruiken. Bijvoorbeeld:

- ToString(): Hiermee kun je een tekstuele representatie van een object krijgen (bijvoorbeeld een beschrijving van een auto).
- Equals(): Hiermee kun je controleren of twee objecten gelijk zijn aan elkaar.
- GetHashCode(): Dit is een speciale code die elk object uniek identificeert.

Onzichtbaar, maar overal aanwezig: je hoeft de Object-klasse niet expliciet te noemen in je code. Het is als de stille helper die altijd aanwezig is. Wanneer je een nieuwe klasse maakt, erft deze automatisch de functies en eigenschappen van de Object-klasse.

De `Object`-klasse is als de verborgen basis waarop alle andere klassen in C# zijn gebouwd. Deze klasse is de *ultieme superklasse* van elke andere klasse.

`Object` is de klasse aan de top van elke klassenhiërarchie. Als een klasse niet expliciet een subklasse is van een andere klasse, dan wordt ze impliciet een subklasse van de klasse Object.

Klassen ondersteunen **enkelvoudige overerving** in C#, waarbij elke klasse slechts één superklasse kan hebben, behalve de klasse `Object`, die geen superklasse heeft.

> Class Object is the root of the class hierarchy. Every class has Object as a superclass. All objects implement the methods of this class.
>
> — Java API

> Supports all classes in the .NET class hierarchy and provides low-level services to derived classes. This is the ultimate base class of all .NET classes; it is the root of the type hierarchy.
>
> — C# API

7.1. Object methoden

Elk klasse erft de eigenschappen en het gedrag van hun verre superklasse Object. Dat gedrag omvat o.a.:

- **Equals**: definieert de ***notie van gelijkheid gebaseerd op de toestand van een object***, los van de referentie.
- **ToString**: geeft een *String* representatie terug van het object.
- **GetType**: geeft een referentie terug naar het *Type* object, een representatie van de klasse van dat object.

7.1.1. `Equals` methode

De methode `Equals` definieert de ***notie van gelijkheid gebaseerd op de toestand van een object***, los van de referentie. `Equals` vergelijkt twee objecten en retourneert `true` als ze 'gelijk' zijn, anders `false`.

Zowel de `Equals` methode als de operator "==" vergelijken 'iets' met elkaar. Er zijn echter belangrijke verschillen tussen beide:

- Het belangrijkste verschil is dat `Equals` een methode is, daar waar "==" een operator is.
- De operator "==" wordt gebruikt om waarden te vergelijken. In geval van referentiewaarden gaat het om adressen in het geheugen. De methode `Equals` gaat vergelijken op basis van inhoud.

De operator "==" vergelijkt of twee waarden, deze links en rechts van de "==" operator, wiskundig gelijk zijn aan elkaar. In geval waarde variabelen gebruikt worden, worden de waarden van de variabelen vergeleken. Gaat het om referentie variabelen, dan verwijzen deze waarden naar een locatie in het geheugen. Verwijzen twee referenties naar dezelfde locatie in het geheugen, dan verwijzen ze naar hetzelfde object. Dit is de default implementatie van de methode `Equals` in de klasse `Object`.

De operator "==" vergelijkt dus de referentiewaarden van de objecten met elkaar in tegenstelling tot de `.Equals()` methode die twee objecten vergelijkt op basis van hun toestand.

→ Indien een klasse de methode `Equals` niet overschrijft, dan zal de overgeërfde default implementatie van de Object klasse gebruikt worden of de implementatie uit de dichtstbijzijnde super klasse. De default implementatie van de Object klasse valt terug op de operator "==". Dit is meestal niet het gewenste gedrag!

```
1    public class Student
2    {
3        public string Naam { get; set; }
4
5        public Student(string naam)
6        {
7            Naam = naam;
8        }
9
10       public override bool Equals(object obj)
11       {
12           return obj is Student student &&
13                  Naam == student.Naam;
14       }
15
16       public override int GetHashCode()
17       {
18           return HashCode.Combine(Naam);
```

```
        }
    }

    public class EqualsTest
    {
        public static void Main(string[] args)
        {

            Student s1 = new Student("Jan");
            Student s2 = new Student("Jan");

            Console.WriteLine(s1 == s2); ①
            Console.WriteLine(s1.Equals(s2)); ②
        }
    }
```

① False

② True

In het voorbeeld worden twee objecten geïnstantieerd: s1 en s2. Beide referenties verwijzen naar verschillende objecten waardoor ze volgens de "==" operator verschillend zijn: twee verschillende objecten kunnen nooit op dezelfde locatie in het geheugen staan.

Via de `Equals` methode worden de objecten echter vergeleken op inhoud: de klasse Student specialiseert de methode `Equals` en vergelijkt twee String objecten (Naam) op basis van de inhoud van de bijgehouden tekst.

De twee student objecten worden als gelijk beschouwd indien ze dezelfde naam hebben (zie implementatie van de Equals methode in de klasse Student).

Nog een voorbeeld:

```
    public class Robot
    {
        private readonly string _type;
        private readonly int _serieNummer; // Uniek serienummer

        public Robot(string type, int serieNummer) : base()
        {
            this._type = type;
            this._serieNummer = serieNummer;

            const int prime = 31;
            int result = 1;
            result = prime * result + _serieNummer;
            return result;
        }

        public override bool Equals(object obj)
        {
```

```
19              return obj is Robot robot &&
20                     _serieNummer == robot._serieNummer; ①
21          }
22
23          public override int GetHashCode()
24          {
25              return HashCode.Combine(_serieNummer); ②
26          }
27      }
```

In bovenstaan voorbeeld heeft een Robot een bepaald serieNummer dat uniek is:

① Twee Robot objecten kunnen met elkaar vergeleken worden (@Override van `Equals`) en zijn gelijk aan elkaar als hun serieNummer hetzelfde is.

② De methode `Equals` gaat altijd samen met de methode `GetHashCode'. Indien je de ene specialiseert moet je vaak ook de ander specialiseren.

De methodes `Equals` en `GetHashCode` horen samen:

- als twee objecten gelijk zijn aan elkaar (`o1.Equals(o2)` is `true`), dan moet de gegenereerde hashcode dezelfde zijn (`o1.GetHashCode() == o2.GetHashCode()` moet altijd `true` zijn in dit geval).
- als twee objecten een gelijke hashcode hebben (`o1.GetHashCode() == o2.GetHashCode()` is `true`), dan betekent dit **niet** dat de twee objecten gelijk zijn aan elkaar (`o1.Equals(o2)` hoeft niet `true` te zijn).

Je mag `GetHashCode` en `Equals` zo implementeren dat de `GetHashcode` voor twee objecten een gelijke waarde oplevert terwijl de `Equals` methode voor diezelfde twee objecten false retourneert. Retourneert de `Equals` methode true, dan moet de `GetHashcode` dezelfde waarde opleveren.

De compiler kan dit gedrag niet afdwingen, het is aan de software ontwikkelaar om dit te garanderen.

> If two objects are equal according to the Equals(Object) method, then calling the GetHashCode method on each of the two objects must produce the same integer result.
>
> — API

Visual Studio biedt de mogelijkheid om de methodes `Equals` en `GetHashCode` samen te genereren d.m.v. een snippet.

7.1.2. `ToString` methode

De methode `ToString` is één van de methoden die elke klasse direct of indirect overerft van de klasse Object.

De `ToString` methode geeft een tekstuele voorstelling weer, meestal met een in

'mensentaal' begrijpbare beschrijving van het object.

Deze methode wordt ook impliciet aangeroepen wanneer een object moet geconverteerd worden naar een string.

```
1    Kat mijnKat = new Kat("Garfield");
2    Console.WriteLine(mijnKat.ToString()); ①
3    Console.WriteLine(mijnKat); ②
```

① De uitvoer is gelijk aan <2>

② De uitvoer is gelijk aan <1>

Zowel <1> als <2> roepen de `ToString` methode aan van het object myString.

8. De klasse Type

In C# is de `Type` **klasse** een belangrijk onderdeel van de .NET-bibliotheek. Het vertegenwoordigt **type-informatie** over een klasse, struct, interface, array, enum, of ander type tijdens de uitvoering van een programma (runtime).

Met de `Type` klasse kun je informatie over een type opvragen, zoals de naam, methoden, eigenschappen, velden, basisklassen, interfaces, en meer. Dit wordt vaak gebruikt in **reflection**, een krachtige functie van .NET waarmee je metadata over types kunt inspecteren en dynamisch code kunt uitvoeren.

8.1. Wat is de Type klasse?

De `Type` klasse bevindt zich in de `System`-namespace en biedt de volgende functionaliteiten:

1. **Type-informatie opvragen**: Je kunt details over een type verkrijgen, zoals de naam, assembly, basisklasse, interfaces, enz.
2. **Reflection**: Je kunt dynamisch methoden aanroepen, eigenschappen instellen, of velden uitlezen van een object.
3. **Type-vergelijking**: Je kunt controleren of een object van een bepaald type is of of het een specifieke interface implementeert. Beter is om dit te doen met de keywords `is` en `as`.

8.2. Hoe krijg je een Type object?

Er zijn verschillende manieren om een `Type` object te verkrijgen:

1. **Via `typeof`**: Gebruik de `typeof` operator om het `Type` object van een type te krijgen.

```
Type type = typeof(int);
Console.WriteLine(type.Name); // Output: Int32
```

2. **Via `GetType()`**: Elk object in C# heeft een `GetType()` methode die het `Type` object van dat object retourneert.

```
string text = "Hello";
Type type = text.GetType();
Console.WriteLine(type.Name); // Output: String
```

3. **Via `Type.GetType()`**: Je kunt de volledig gekwalificeerde naam van een type gebruiken om het `Type` object op te halen.

```
Type type = Type.GetType("System.String");
Console.WriteLine(type.Name); // Output: String
```

8.3. Wat kun je doen met de `Type` klasse?

Met een `Type` object kun je allerlei informatie over een type opvragen en dynamisch code uitvoeren.

8.3.1. Type-informatie opvragen

Je kunt de naam, assembly, basisklasse, interfaces, en meer opvragen.

```
Type type = typeof(string);

Console.WriteLine("Name: " + type.Name); // Output: String
Console.WriteLine("Full Name: " + type.FullName); // Output: System.String
Console.WriteLine("Namespace: " + type.Namespace); // Output: System
Console.WriteLine("Is Class: " + type.IsClass); // Output: True
Console.WriteLine("Base Type: " + type.BaseType); // Output: System.Object
```

8.3.2. Methoden en eigenschappen inspecteren

Je kunt alle methoden, eigenschappen, velden, en constructors van een type opvragen.

```
Type type = typeof(string);

// Alle publieke methoden opvragen
foreach (var method in type.GetMethods())
{
    Console.WriteLine(method.Name);
}

// Alle publieke eigenschappen opvragen
foreach (var property in type.GetProperties())
{
    Console.WriteLine(property.Name);
```

```
}
```

8.3.3. Dynamisch een object maken

Je kunt een object van een type maken met behulp van de `Activator.CreateInstance` methode.

```
Type type = typeof(string);
object instance = Activator.CreateInstance(type); // Maakt een nieuwe string
Console.WriteLine(instance); // Output: (lege string)
```

8.3.4. Dynamisch methoden aanroepen

Je kunt methoden van een object dynamisch aanroepen met behulp van reflection.

```
string text = "Hello, World!";
Type type = text.GetType();

// Zoek de Substring methode
var method = type.GetMethod("Substring", new[] { typeof(int), typeof(int) });

// Roep de Substring methode aan
var result = method.Invoke(text, new object[] { 0, 5 });
Console.WriteLine(result); // Output: Hello
```

8.3.5. Controleren of een type een interface implementeert

Je kunt controleren of een type een specifieke interface implementeert.

```
Type type = typeof(string);
bool implementsIEnumerable =
typeof(System.Collections.IEnumerable).IsAssignableFrom(type);
Console.WriteLine(implementsIEnumerable); // Output: True
```

8.4. Samenvatting

De `Type` klasse in C# is een krachtig hulpmiddel om type-informatie tijdens runtime te verkrijgen en te gebruiken. Het wordt vaak gebruikt in combinatie met **reflection** om dynamisch code uit te voeren, zoals het inspecteren van methoden, het maken van objecten, of het aanroepen van methoden. Dit is vooral handig in geavanceerde scenario's zoals serialisatie, dependency injection, en plug-in systemen.

9. Initialisatie van een overerving hiërarchie

9.1. Constructor

Constructors worden nooit overgeërfd: ze kunnen dus niet overschreven worden.

Constructors worden aangeroepen vanuit expressies die een object van een klasse instantiëren, vanuit conversies en concatenaties t.g.v. de string concatenatie operator en door het expliciet aanroepen van een constructor vanuit een andere constructor.

Constructoren kunnen nooit aangeroepen worden door een methode expressie. Een constructor is geen methode! En wordt dus nooit overgeërfd.

9.1.1. Soorten constructors

Er bestaan verschillende soorten constructors in de C# programmeertaal:

Default Constructor (Standaardconstructor)

Een default constructor is een speciale constructor die geen parameters accepteert.

```
class Auto
{
    public Auto()
    {
        // Dit is de default constructor
    }
}
```

Als een klasse geen declaratie voorziet van een constructor, dan wordt er impliciet een default constructor gedeclareerd. Eens een andere constructor is gedeclareerd krijg je de default constructor er niet meer zomaar bij.

Gewone Constructor

Een gewone constructor is een aangepaste constructor die je zelf definieert in je klasse. Je voegt parameters toe om specifieke waarden in te stellen bij het maken van een object.

```
class Bankrekening
{
    public Bankrekening(string eigenaar, double saldo)
    {
        Eigenaar = eigenaar;
        Saldo = saldo;
    }

    public string Eigenaar { get; set; }
    public double Saldo { get; set; }
}
```

Primaire Constructor

De primaire constructor (= `Primary constructor`) is een speciale constructor die je kunt definiëren in een klasse.

een primaire constructor wordt gebruikt om verplichte parameters te specifiëren bij het maken van een nieuw object van die klasse. Met andere woorden, je kunt aangeven welke gegevens **altijd** moeten worden ingevuld wanneer je een object maakt. Elke andere constructor in de klasse zal expliciet deze primaire constructor moeten aanroepen!

De belangrijkste kenmerken van primaire constructors zijn:

1. De parameters worden direct na de klassenaam gedeclareerd tussen haakjes
2. De parameters zijn beschikbaar in de hele klasse
3. Je kunt ze combineren met automatisch geïmplementeerde properties

```
class Persoon(String naam, int leeftijd) ①
{
    public string Naam { get; } =
        !String.IsNullOrWhiteSpace(naam) ? naam : throw new
ArgumentOutOfRangeException(nameof(naam), "De naam mag niet leeg zijn."); ②


    public int Leeftijd { get; } = leeftijd >= 0 ? leeftijd : throw new
ArgumentOutOfRangeException(nameof(leeftijd), "De leeftijd moet positief zijn.");
}

class Student(String naam, int leeftijd, string studentId) : Persoon(naam, leeftijd)
④
{
    public string StudentId { get; } = studentId;

    public Student(string naam, int leeftijd) : this(naam, leeftijd, String.Empty) ③
    {
    }
}
```

1. Stel je voor dat je een klasse `Persoon` hebt. Elke persoon moet een naam en een leeftijd hebben. Met een primaire constructor kun je deze verplichte gegevens direct bij het maken van een object instellen, zonder extra stappen.
2. Deze parameters zijn overal in de klasse beschikbaar. Je kunt ze gebruiken om eigenschappen of velden te initialiseren.
3. Andere constructoren zijn verplicht om de primaire constructor aan te roepen, wat logisch is omdat we willen afdwingen dat die parameters verplicht zijn.
4. Vanuit de primaire constructor kan ook de superklasse geïnitialiseerd worden.

9.1.2. Constructor body

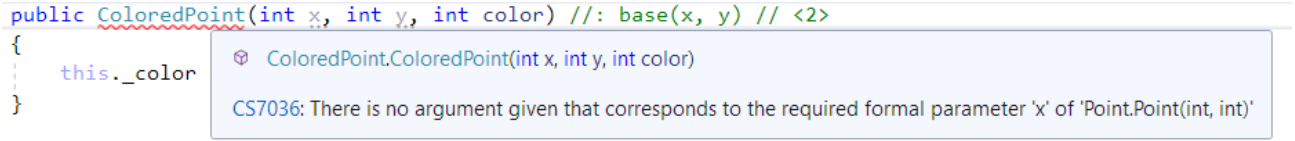
Bij de declaratie van een constructor mag een expliciete aanroep gebeuren naar een constructor van dezelfde klasse, of de directe superklasse. Als er een primaire constructor is moet deze aangeroepen worden.

Als binnen de declaratie van een constructor geen expliciete aanroep gebeurt van een andere constructor (en het niet gaat om de klasse Object) dan zal de constructor **impliciet** beginnen met het aanroepen van de superklasse constructor **base()**, een aanroep van diens **default** constructor. Bestaat deze default constructor in de superklasse niet, dan krijg je een foutmelding.

```
1   public class Point
2   {
3       private int _x, _y;
4
5       public Point(int x, int y)
6       {
7           this._x = x;
8           this._y = y;
9       }
10  }
11
12  public class ColoredPoint : Point
13  {
14      private const int WHITE = 0, BLACK = 1;
15      private int _color;
16
17      public ColoredPoint(int x, int y) : this(x, y, WHITE) ①
18      {
19      }
20
21      public ColoredPoint(int x, int y, int color) : base(x, y) ②
22      {
23          this._color = color;
24      }
25  }
```

① de eerste constructor van ColoredPoint roept de tweede constructor aan met een extra argument

② De tweede constructor van ColoredPoint roept als eerste de constructor aan van zijn superklasse Point, en geeft de nodige parameters door. Doe je dit niet, dan zal impliciet de default constructor aangeroepen worden. Als deze niet bestaat, krijg je een compiler error.



9.2. Sequentie van initialisatie: Top to bottom!

In een klassenhiërarchie zal eerste de toestand van de hoogste klasse in de hiërarchie geïnitialiseerd worden (= Object), daarna de 2de hoogste, dan de 3de etc.

Stel: Rechthoek IS EEN Veelhoek IS EEN Figuur IS EEN Object

```
1    public class Figuur
2    {
3        private string _naam;
4
5        public Figuur(string naam)
6        {
7            Console.WriteLine("Constructor Figuur");
8
9            this._naam = naam;
10       }
11   }
```

```
1    public class Veelhoek : Figuur
2    {
3        public Veelhoek(string naam) : base(naam)
4        {
5            Console.WriteLine("Constructor Veelhoek");
6        }
7    }
```

```
1    public class Rechthoek : Veelhoek
2    {
3        public Rechthoek(string naam) : base(naam)
4        {
5            Console.WriteLine("Constructor Rechthoek");
6        }
7    }
```

```
1    public class Vierkant : Rechthoek
2    {
3        public Vierkant(string naam) : base(naam)
4        {
5
6            Console.WriteLine("Constructor Vierkant");
7        }
8    }
```

```
1    public class InitialisatieVanObjecten
```

```
2     {
3         public static void Main(string[] args)
4         {
5             new Vierkant("Mijn vierkant");
6         }
7     }
```

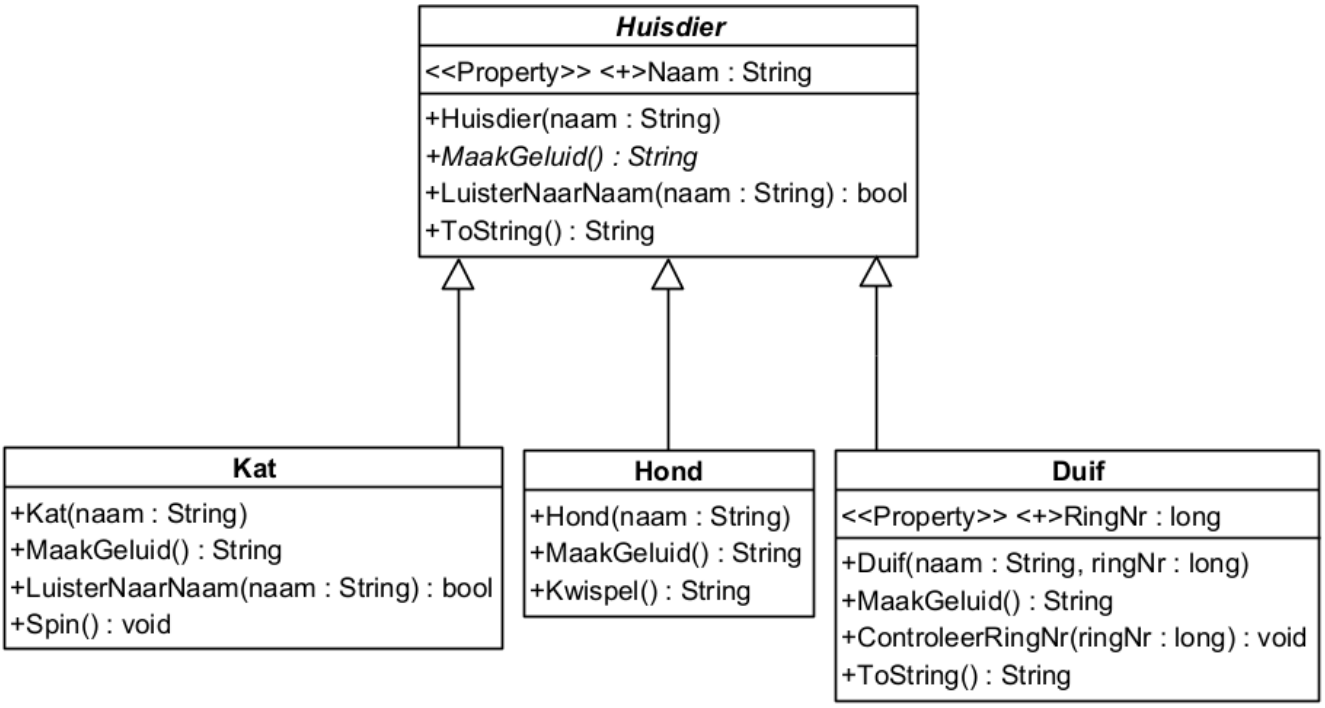
Bekijk de uitvoer van dit voorbeeld!

Vergeet geen noodzakelijke parameters door te geven vanuit de subklasse naar de superklasse, zodat de overgeërfde eigenschappen correct kunnen geïnitialiseerd worden.

Als de superklasse een constructor declareert en niet beschikt over een default constructor, dan moet deze constructor expliciet aangeroepen worden. Anders krijg je een compile error.

10. Uitbreiding van een klasse

Een klasse erft eigenschappen en gedrag van zijn superklasse, kan nieuwe eigenschappen en gedrag toevoegen of bestaand gedrag specialiseren.



In voorgaande hiërarchie breiden de subklassen Kat, Hond en Duif de superklasse Huisdier uit:

- Kat voegt het gedrag `spin` toe
- Hond voegt het gedrag `kwispel` toe
- Duif voegt de eigenschap `ringnr` samen met het bijhorende gedrag

11. Specialisatie van een klasse

In voorgaande hiërarchie merk je ook op dat de subklassen Kat, Hond en Duif elk hun eigen versie implementeren van de methode `MaakGeluid`. Dit gedrag werd reeds in de klasse Huisdier gedeclareerd en wordt door de subklassen overschreven. Dit noemen we specialisatie. Via het keyword `base` kan de subklasse nog steeds het gedrag uit de superklasse aanroepen.

```
1   internal class Radio
2   { // Superklasse
3       public virtual void SpeelMuziek()
4       {
5           Console.WriteLine("De radio speelt muziek");
6       }
7   }
8
9   internal class KlassiekeRadio : Radio
10  { // Subklasse
11      public override void SpeelMuziek()
12      {
13          base.SpeelMuziek(); ①
14          Console.WriteLine("De klassieke radio speelt: Mozart");
15      }
16  }
17
18  public class RadioApp
19  {
20      public static void Main(string[] args)
21      {
22          KlassiekeRadio radio = new KlassiekeRadio();
23          radio.SpeelMuziek();
24      }
25  }
```

① De gespecialiseerde methode `SpeelMuziek` roept het gedrag `SpeelMuziek` uit de superklasse aan.

12. `abstract` keyword

Een klasse of methode kan abstract gemaakt worden door het keyword `abstract` toe te voegen aan de declaratie.

12.1. `abstract` klasse

Van een abstracte klasse kan geen instantie gecreëerd worden. Probeer je dit toch, dan resulteert dit in een compile-time error.

Een abstracte klasse is een klasse die kan aanzien worden als nog niet volledig afgewerkt of het is expliciet de bedoeling dat deze klasse niet kan geïnstantieerd

worden.

Indien een klasse `abstract` is kan ze ook abstracte methodes declareren. Abstracte methodes omvatten enkel een declaratie, zonder implementatie van de methode.

Een klasse beschikt over abstracte methodes indien:

- er expliciet abstracte methodes gedeclareerd worden binnen de klasse
- enige overgeërfde methode als abstract gedeclareerd werd en nog niet geïmplementeerd werd

Een niet abstracte klasse kan je benoemen als een ***concrete*** klasse.

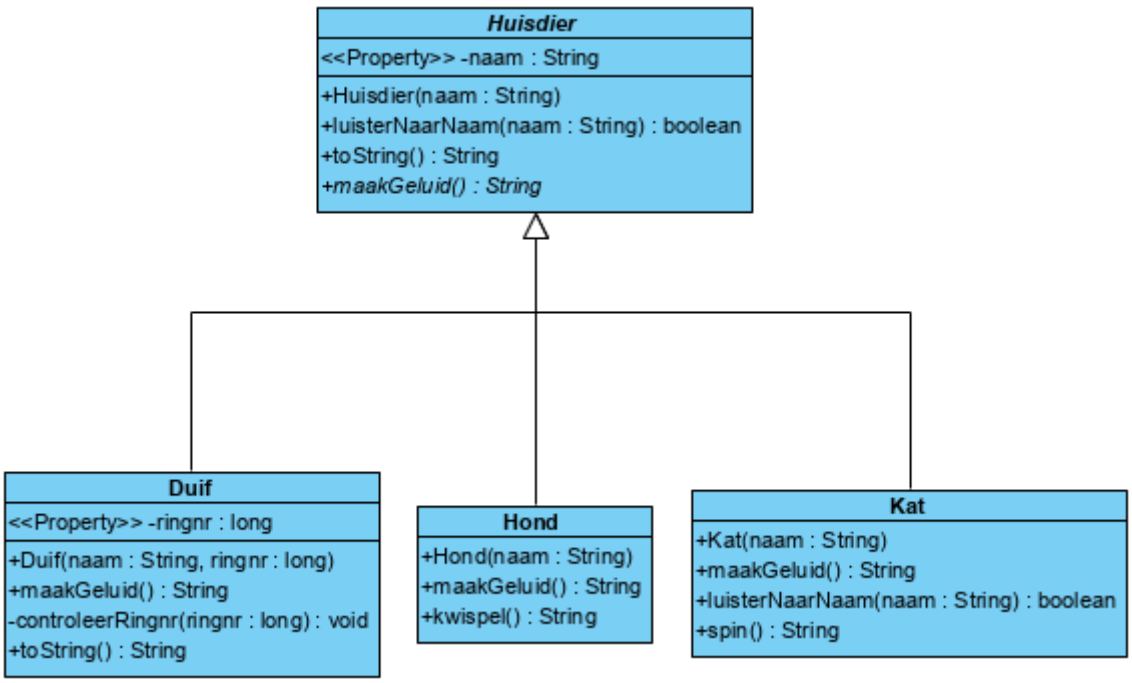
12.2. `abstract` methode

Een declaratie van een abstracte methode (keyword `abstract`) introduceert een methode als gedrag, waaronder zijn handtekening en return value , maar zonder de implementatie van de methode te voorzien.

Een niet abstracte methode kan je benoemen als een concrete methode.

12.3. 'abstract' voorbeeld

Binnen de Huisdier hiërarchie is het zinloos om een Huisdier te instantiëren: wat is precies een Huisdier en welk geluid maakt het? In deze context is het aangewezen om te verhinderen dat van Huisdier een instantie kan gemaakt worden door deze klasse abstract te maken. Ook de methode `MaakGeluid` kan op dit niveau in de hiërarchie weinig zinvol geïmplementeerd worden. Ook deze kan aangeduid worden als abstract.



In een UML wordt een abstracte klasse aangeduid door zijn naam *italic* te

schrijven. Ook de naam van een abstracte methode wordt *italic* geschreven.

```
public abstract class Huisdier //extends Object
{
    private string _naam;

    public Huisdier(string naam)
    {
        Naam = naam;
    }
    public virtual string Naam
    {
        get
        {
            return _naam;
        }
        set
        {
            this._naam = value;
        }
    }
    public virtual bool LuisterNaarNaam(string naam)
    {
        return (naam.Equals(this._naam));
    }
    public override string ToString()
    {
        return string.Format("{0} met naam {1}", this.GetType().Name, _naam);
    }

    public abstract string MaakGeluid();
}
```

13. `static` keyword

13.1. static attribuut

Als een attribuut `static` wordt gedeclareerd bestaat er slechts één instantie van dat attribuut, ongeacht het aantal instanties (mogelijks geen enkele) dat van die klasse gemaakt worden.

Een `static` attribuut, ook benoemt als **klassevariabele**, wordt aangemaakt als de klasse zelf voor de eerste keer wordt geïnitialiseerd (= op het moment dat de klasse wordt ingeladen.)

Een klassevariabele wordt dus gedeeld door alle instanties van die klasse.

Een attribuut gedeclareerd zonder het keyword `static`, ook wel non-static

attribuut genoemd, noemt men een **instantievariabele**. Telkens een nieuwe instantie van de klasse wordt gecreëerd, zal een nieuwe variabele ontstaan geassocieerd met die klasse-instantie en dit voor elke instantievariabele gedeclareerd in die klasse of in elk van zijn superklassen.

13.2. `static` methode

Een `static` methode, ook benoemd als **klassemethode**, kan aangeroepen worden zonder een referentie naar een instantie van die klasse. Binnen een `static` methode resulteert het gebruik van de keywords `this` of `base` in een compile time error.

Een methode niet gedeclareerd als `static` noemt men een **instantiemethode** of een non-static methode. Een instantiemethode wordt altijd aangeroepen in relatie met een object, dat het huidige object wordt naar waar het keyword `this` verwijst tijdens de uitvoer van die methode.

14. Het keyword `sealed`

Een klasse kan gedeclareerd worden als `sealed` (= verzegeld) om te voorkomen dat ze gebruikt wordt als superklasse. Een `sealed` klasse kan dus geen subklassen hebben.

De klasse String is een voorbeeld van een `sealed` klasse.

```
public sealed class String
    : ICloneable, IComparable, IComparable<string>,
      IConvertible, IEquatable<string>, IParsable<string>,
      ISpanParsable<string>,
      System.Collections.Generic.IEnumerable<char>
```

Ook een methode kan als `sealed` gedeclareerd worden om te voorkomen dat een subklasse deze overschrijft.

15. `readonly` attribuut en init-setters

readonly-attribuut

Het readonly-attribuut is als een beschermende laag voor attributen in een klasse. Het zorgt ervoor dat een attribuut alleen kan worden toegewezen tijdens de initialisatie (bijvoorbeeld in de constructor) en daarna niet meer kan worden veranderd.

Als je een attribuut als readonly markeert, kun je het alleen een waarde geven wanneer je het object aanmaakt, maar daarna blijft het onveranderlijk.

init-setter

Een init-setter is een speciale setter om properties in te stellen. Het lijkt op een gewone setter, maar je kan hem alleen gebruiken tijdens objectinitialisatie (bijvoorbeeld in de constructor).

Een property met een init-setter kan worden ingesteld bij het maken van een object, maar daarna niet meer worden gewijzigd. Let op de gelijkenis met een readonly attribuut.

Indien een property met init-setter een achterliggend attribuut heeft is het logisch om dit attribuut `readonly` te maken.

Voorbeeld:

```
class Persoon
{
    private readonly string _naam;

    public string Naam
    {
        get => _naam;
        init => _naam = value;
    }

    public Persoon(string naam)
    {
        Naam = naam; // Hier gebruiken we de init-setter
    }
}
```

Bij het maken van een nieuw persoon-object kun je de naam instellen met de init-setter:

```
Persoon mijnPersoon = new Persoon("Alice");
// mijnPersoon.Naam = "Bob"; // Dit zou een fout geven na de initialisatie
```

Gebruik readonly voor attributen en init voor properties als je wilt dat ze alleen tijdens de initialisatie kunnen worden ingesteld.

Dit helpt om je code veiliger en robuuster te maken, omdat je zeker weet dat bepaalde waarden niet per ongeluk worden gewijzigd na de creatie van een object.

16. Andere types

16.1. Het `record` type

Een `record` is een speciaal type dat handig is voor het werken met gegevensmodellen.

Wat is een record?

Een record is als een slimme container voor gegevens. Het helpt je om informatie op een gestructureerde manier op te slaan.

Het is vergelijkbaar met een klasse, maar met enkele handige extra's.

Waarom zijn records handig?

Records bieden:

- Immutabiliteit: Gegevens in een record kunnen niet veranderen nadat je ze hebt ingesteld.
- Waardegelijkheid: Records vergelijken gebeurt inhoudelijk in plaats van op basis van geheugenadressen. Als alle properties van een record van hetzelfde type zijn en hun inhoud is gelijk, dan gaat het om dezelfde record.
- Beknopte syntax: Je kunt snel een record maken met minder code.

Voorbeeld van een record

```
public record Persoon(string Voornaam, string Achternaam); ①

Persoon mijnPersoon = new Persoon("Alice", "Johnson"); ②
```

1. Declaratie van een record
2. Aanmaken van een instantie van een record.

Records zijn vooral bedoeld voor onveranderlijke gegevensmodellen. Ze zijn handig voor situaties waarin je gegevens wilt opslaan en vergelijken, zoals bijvoorbeeld personen, punten in een ruimte, enzovoort. Records zijn als slimme dozen om je gegevens snel en netjes te organiseren!

16.1.1. Records kopiëren

In C# kun je eenvoudig een kopie van een record maken met behulp van de `with`**-expressie**. Dit is een feature van records, waarmee je een nieuw record kunt maken dat een kopie is van een bestaand record, behalve voor de eigenschappen die je wilt wijzigen.

Voorbeeld van een record en gebruik van de `with`*-expressie*

```
public record Persoon(string Voornaam, string Achternaam, int Leeftijd);

var persoon1 = new Persoon("Jan", "Janssen", 30); ①

// Maak een kopie van 'persoon1' en wijzig de leeftijd
var persoon2 = persoon1 with { Leeftijd = 31 }; ②

// Output
Console.WriteLine(persoon1); // Output: Persoon { Voornaam = Jan, Achternaam =
Janssen, Leeftijd = 30 } ③
```

```
Console.WriteLine(persoon2); // Output: Persoon { Voornaam = Jan, Achternaam =
Janssen, Leeftijd = 31 }
```

1. `persoon1` is een origineel record.
2. Met `with` maak je een nieuwe instantie van het record:
   - Alle eigenschappen van `persoon1` worden gekopieerd naar `persoon2`.
   - De eigenschap `Leeftijd` wordt overschreven met een nieuwe waarde (`31`).
3. Het originele record (`persoon1`) blijft onveranderd, want records zijn **immutable**.

16.2. Het `tuple` type

Een `tuple` is een handige manier om meerdere gegevens samen te voegen in één lichtgewicht structuur.

Stel je voor dat een tuple een soort “gegevenszakje” is waarin je snel verschillende waarden kunt stoppen, zonder hiervoor een aparte klasse of record aan te moeten maken. In een tuple kan je geen methodes declareren, maar de methods die .NET voorziet zijn wel beschikbaar (bv. ToString en de equality operatoren zoals `==` en `!=`).

Definitie

Een tuple wordt gemaakt met behulp van het tuple-keyword. Het stelt je in staat om verschillende gegevens samen te voegen zonder dat je een specifieke klasse of structuur hoeft te definiëren.

Elementen

Een tuple kan meerdere elementen bevatten, zoals getallen, tekst, of andere gegevenstypes. Deze elementen kunnen van verschillende datatypen zijn.

Toegang tot elementen

Geef je de elementen in een tuple geen naam, dan kan je de waarden benaderen met behulp van de Item1, Item2, enzovoort. Bijvoorbeeld: als je een tuple hebt met twee getallen, kun je ze ophalen met tuple.Item1 en tuple.Item2.

We raden aan om de waarden in een tuple altijd een naam te geven. Deze naam start met een hoofdletter.

```
// Een tuple met een double en een integer
(double Gewicht, int Leeftijd) persoon = (75.5, 30);

// Toegang tot de waarden in de tuple
Console.WriteLine($"Gewicht: {persoon.Gewicht} kg, Leeftijd: {persoon.Leeftijd}
jaar");
```

Dit zal “Gewicht: 75.5 kg, Leeftijd: 30 jaar” afdrukken. Tuples bieden een handige manier om

gegevens te groeperen zonder veel extra code te schrijven!

16.2.1. Nuttige toepassingen van een `tuple`

Een tuple heeft verschillende nuttige toepassingen. Veelvoorkomende gebruiksscenario's zijn:

Teruggeven van meerdere waarden uit een methode

In plaats van individuele return types te definiëren, kun je methode return types groeperen in een tuple. Bijvoorbeeld de methode `FindMinMax`:

```
(int Min, int Max) FindMinMax(int[] input) ①
{
    if (input is null || input.Length == 0)
    {
        throw new ArgumentException("Cannot find minimum and maximum of a null or
empty array.");
    }

    // Initialize min to MaxValue so every value in the input
    // is less than this initial value.
    var min = int.MaxValue;

    // Initialize max to MinValue so every value in the input
    // is greater than this initial value.
    var max = int.MinValue;

    foreach (var i in input)
    {
        if (i < min)
        {
            min = i;
        }
        if (i > max)
        {
            max = i;
        }
    }

    return (min, max); ②
}
```

1. Deze methode heeft als return type een tuple met twee int waarden. Deze krijgen elk een naam: `min` en `max`.
2. De min en max waarde worden teruggegeven als een tuple: let op de haakjes!

```
int[] getallen = { 4, 7, 9 };
var (minimum, maximum) = FindMinMax(getallen);
```

```
Console.WriteLine($"Minimale en maximale waarde van [{string.Join(" ", getallen)}]
zijn {minimum} en {maximum}");

// Output: Minimale en maximale waarde van [4 7 9] zijn 4 en 9
```

De FindMinMax-methode retourneert een tuple met twee waarden: het minimum en het maximum van een gegeven array.

Representatie van een enkele set gegevens

Tuples kunnen een handige manier zijn om een enkele set gegevens te vertegenwoordigen. Bijvoorbeeld, een tuple kan een database-record voorstellen, waarbij de componenten individuele velden van het record zijn.

Eenvoudige toegang tot en manipulatie van een gegevensset

Tuples stellen je in staat om meerdere waarden samen te voegen zonder dat je een specifieke klasse of structuur hoeft te definiëren. Ze kunnen handig zijn voor tijdelijke gegevensopslag.

Samenvoegen van verschillende gegevenstypes

Tuples kunnen elementen van verschillende datatypen bevatten, zoals getallen, tekst of andere waarden. Dit maakt ze flexibel voor diverse situaties.

Tuples zijn handig wanneer je een lichtgewicht gegevensstructuur nodig hebt om meerdere waarden te groeperen zonder veel extra code te schrijven!

16.2.2. Tuple equality

Tuples ondersteunen gelijkheidsoperatoren zoals `==` en `!=`. Dit betekent dat je tuples kunt vergelijken op basis van hun waarden.

Gelijkheid van tuples

Je kunt twee tuples vergelijken met behulp van de `==` operator. Als alle elementen in beide tuples gelijk zijn, retourneert de operator true, anders retourneert deze false.

```
var tuple1 = (4, 7);
var tuple2 = (4, 7);

bool zijnGelijk = tuple1 == tuple2;

Console.WriteLine($"Zijn de tuples gelijk? {zijnGelijk}"); // Output: Zijn de tuples
gelijk? True
```

Ongelijkheid van tuples

Je kunt ook de `!=` operator gebruiken om te controleren of twee tuples ongelijk zijn.

```
var tuple3 = (1, 2);
var tuple4 = (3, 4);
```

```
bool zijnOngelijk = tuple3 != tuple4;
Console.WriteLine($"Zijn de tuples ongelijk? {zijnOngelijk}"); // Output: Zijn de
tuples ongelijk? True
```

Tuples zijn waardegebaseerd

Tuples zijn waardetypes en hun elementen zijn `public`. Dit betekent dat tuples mutabel zijn (je kunt hun waarden wijzigen). Je kunt tuples definiëren met een willekeurig groot aantal elementen.

17. Nullable value types

In C# zijn **nullable value types** een manier om waarde-types (zoals `int`, `double`, `bool`, enz.) de mogelijkheid te geven om `null` te zijn. Normaal gesproken kunnen waarde-types geen `null` bevatten, omdat ze altijd een waarde moeten hebben. Met nullable value types kun je aangeven dat een variabele van een waarde-type óf een geldige waarde heeft óf `null` is.

17.1. Waarom nullable value types?

Waarde-types zoals `int`, `bool`, en `DateTime` kunnen niet `null` zijn. Dit kan problemen veroorzaken in situaties waar een waarde ontbreekt of onbekend is. Bijvoorbeeld:

- Een databaseveld kan `NULL` bevatten, maar hoe vertaal je dat naar een `int` in C#?
- Een optionele gebruikerinvoer kan ontbreken, maar hoe geef je dat aan in een `DateTime`?

Nullable value types lossen dit op door waarde-types de mogelijkheid te geven om `null` te zijn.

17.2. Hoe werken nullable value types?

Nullable value types worden geïmplementeerd met behulp van de `Nullable<T>` struct, waarbij `T` een waarde-type is. **C# biedt ook een verkorte syntax met een vraagteken (`?`) achter het type.**

In plaats van `Nullable<int>` kun je `int?` schrijven.

Voorbeeld

```
int? nullableInt = null; // Dit is een nullable int
double? nullableDouble = 3.14; // Dit is een nullable double
bool? nullableBool = null; // Dit is een nullable bool
```

17.3. Eigenschappen van nullable value types

1. `HasValue`: Geeft aan of de nullable variabele een waarde heeft.

- `true`: De variabele heeft een waarde.
- `false`: De variabele is `null`.

2. **`Value`**: Geeft de waarde van de nullable variabele. Als `HasValue` `false` is, gooit het een `InvalidOperationException`.
3. **`GetValueOrDefault()`**: Retourneert de waarde als die bestaat, anders de standaardwaarde van het type (bijvoorbeeld `0` voor `int`).

Voorbeeld:

```
int? number = null;

if (number.HasValue)
{
    Console.WriteLine("Value: " + number.Value);
}
else
{
    Console.WriteLine("Value is null");
}

int safeValue = number.GetValueOrDefault(); // Retourneert 0 als number null is
```

17.4. Null-coalescing operator (??)

De null-coalescing operator (??) wordt vaak gebruikt om een standaardwaarde in te stellen als een nullable variabele `null` is.

```
int? number = null;
int result = number ?? 42; // Als number null is, gebruik dan 42
Console.WriteLine(result); // Output: 42
```

17.5. Null-conditional operator (?.)

De null-conditional operator (?.) wordt gebruikt om veilig toegang te krijgen tot leden van een object dat `null` kan zijn. Als het object `null` is, wordt de hele expressie `null`.

```
string? text = null;
int? length = text?.Length; // Als text null is, wordt length ook null
Console.WriteLine(length); // Output: (niets, omdat length null is)
```

17.6. Voorbeelden van nullable value types

Database-integratie

Stel dat je een database hebt waar een kolom `Age` `NULL` kan zijn. In C# kun je dit modelleren met een nullable `int`.

```
int? age = null; // Leeftijd is onbekend
```

Optionele invoer

Als een gebruiker een optioneel veld niet invult, kun je dit aangeven met een nullable type.

```
DateTime? birthDate = null; // Geboortedatum is niet ingevuld
```

Logische vlaggen

Een `bool?` kan drie statussen hebben: `true`, `false`, of `null` (onbekend).

```
bool? isActive = null; // Status is onbekend
```

17.7. Conversies tussen nullable en niet-nullable types

Je kunt een nullable type converteren naar een niet-nullable type, maar je moet rekening houden met de mogelijkheid van `null`.

Voorbeeld:

```
int? nullableInt = 10;
int normalInt = nullableInt.Value; // Werkt als nullableInt een waarde heeft

int? nullableInt2 = null;
// int normalInt2 = nullableInt2.Value; // Gooit een InvalidOperationException
```

Gebruik `GetValueOrDefault()` of de null-coalescing operator (`??`) om veilig een standaardwaarde in te stellen.

```
int safeValue = nullableInt2.GetValueOrDefault(); // Retourneert 0
int safeValue2 = nullableInt2 ?? 42; // Retourneert 42
```

17.8. Samenvatting

- **Nullable value types** stellen waarde-types in staat om `null` te zijn.
- Ze worden geïmplementeerd met `Nullable<T>` of de verkorte syntax `T?`.

- Gebruik `HasValue` en `Value` om te controleren of een waarde aanwezig is en deze op te halen.
- Gebruik `GetValueOrDefault()` of de null-coalescing operator (`??`) om veilig met `null` om te gaan.
- Nullable value types zijn handig in scenario's waar een waarde ontbreekt of onbekend is, zoals database-integratie of optionele invoer.

18. Nullable reference types

Nullable reference types zijn een functie in C# (geïntroduceerd in **C# 8.0**) waarmee je expliciet kunt aangeven of een referentietype (zoals `string`, `object`, of een klasse) `null` kan zijn. Dit helpt om `NullReferenceException`-fouten te voorkomen en maakt je code veiliger en uitdrukkelijker.

18.1. Waarom nullable reference types?

Voor C# 8.0 waren alle referentietypes standaard **nullable**, wat betekent dat ze altijd `null` konden zijn. Dit leidde vaak tot runtime-fouten zoals `NullReferenceException`, omdat de compiler niet kon controleren of een referentie `null` was voordat je er toegang toe had.

Met nullable reference types kun je:

1. **Expliciet aangeven** of een referentie `null` kan/mag zijn.
2. **Waarschuwingen of fouten** krijgen van de compiler als je mogelijk onveilige `null`-toegangen maakt.
3. **Code veiliger** maken door `null`-gerelateerde problemen tijdens het compileren te identificeren.

18.2. Hoe werken nullable reference types?

Nullable reference types worden ingeschakeld door een **compiler-vlag** of een projectinstelling. Zodra ze zijn ingeschakeld, moet je expliciet aangeven of een referentietype `null` kan zijn door een vraagteken (`?`) toe te voegen aan het type.

Voorbeeld:

```
string? nullableString = null; // Dit kan null zijn
string nonNullableString = "Hello"; // Dit kan niet null zijn
```

Als je probeert `null` toe te wijzen aan een niet-nullable referentietype, krijg je een **compiler-waarschuwing**.

```
string nonNullableString = null; // Compiler-waarschuwing: Kan niet null zijn
```

18.3. Nullable reference types inschakelen

Nullable reference types zijn optioneel en moeten worden ingeschakeld in je project. Dit kan op deze manier:

1. **In de `.csproj`-file**: Voeg de volgende regel toe aan je projectbestand:

```
<Nullable>enable</Nullable>
```

18.4. Belangrijke concepten

Nullable annotaties (`?`)

- Als je een vraagteken (`?`) toevoegt aan een referentietype, geef je aan dat het `null` kan zijn.
- Zonder vraagteken wordt aangenomen dat het type **niet `null` kan zijn**.

```
string? nullableString = null; // Kan null zijn
string nonNullableString = "Hello"; // Kan niet null zijn
```

Null-forgiving operator (`!`)

- Soms weet je zeker dat een waarde niet `null` is, maar de compiler geeft een waarschuwing. Met de null-forgiving operator (`!`) kun je de compiler vertellen dat je zeker weet dat de waarde niet `null` is.
- Dit onderdrukt de compiler-waarschuwing.

```
string? nullableString = "Hello";
string nonNullableString = nullableString!; // Geen waarschuwing, omdat we zeker
weten dat het niet null is
```

Null-checking met pattern matching

- Je kunt `null`-checks uitvoeren met behulp van pattern matching.

```
    if (nullableString is null)
    {
        Console.WriteLine("String is null");
    }
```

Null-coalescing operator (`??`)

- Gebruik de null-coalescing operator om een standaardwaarde in te stellen als een waarde `null` is.

```
string? nullableString = null;
string result = nullableString ?? "Default Value"; // Als nullableString null
```

```
is, gebruik dan "Default Value"
```

Null-conditional operator (`?.`)

- Gebruik de null-conditional operator om veilig toegang te krijgen tot leden van een object dat `null` kan zijn.

```
   string? nullableString = null;
   int? length = nullableString?.Length; // Als nullableString null is, wordt
length ook null
```

18.5. Voorbeelden

Database-integratie

Stel dat je een database hebt waar een kolom `Name` `NULL` kan zijn. In C# kun je dit modelleren met een nullable reference type.

```
string? name = GetNameFromDatabase(); // Kan null zijn
if (name is not null)
{
    Console.WriteLine("Name: " + name);
}
```

Optionele invoer

Als een gebruiker een optioneel veld niet invult, kun je dit aangeven met een nullable reference type.

```
string? optionalInput = GetUserInput(); // Kan null zijn
string result = optionalInput ?? "No input provided";
```

Methoden met nullable parameters

Je kunt methoden maken die nullable parameters accepteren.

```
public void PrintMessage(string? message)
{
    if (message is not null)
    {
        Console.WriteLine(message);
    }
    else
    {
        Console.WriteLine("Message is null");
    }
}
```

18.6. Voordelen van nullable reference types

1. **Betere codeveiligheid**: Compiler-waarschuwingen helpen `NullReferenceException`-fouten te voorkomen.
2. **Expliciete intentie**: Je maakt duidelijk of een referentie `null` kan zijn.
3. **Betere onderhoudbaarheid**: Code wordt beter gedocumenteerd en gemakkelijker te begrijpen.

18.7. Samenvatting

- **Nullable reference types** stellen je in staat om expliciet aan te geven of een referentietype `null` kan zijn.
- Ze worden ingeschakeld met een compiler-vlag of projectinstelling.
- Gebruik `?` om een nullable reference type aan te geven en `!` om de compiler te vertellen dat een waarde niet `null` is.
- Ze helpen `NullReferenceException`-fouten te voorkomen en maken je code veiliger en uitdrukkelijker.

19. Toegangsmodifiers (Visibiliteit)

In C# zijn er verschillende **toegangsmodifiers** (visibility modifiers) waarmee je de toegankelijkheid van klassen, methoden, eigenschappen en andere leden kunt regelen. Dit bepaalt welke delen van je code toegang hebben tot deze onderdelen. De belangrijkste toegangsmodifiers zijn:

`private`

- Toegankelijkheid: Alleen binnen dezelfde klasse.
- Gebruik: Om implementatiedetails te verbergen.

Voorbeeld:

```
class MijnKlasse
{
    private int waarde; // Alleen toegankelijk binnen 'MijnKlasse'.
}
```

`public`

- Toegankelijkheid: Overal toegankelijk, zowel binnen als buiten de klasse of assembly.
- Gebruik: Voor onderdelen die publiek beschikbaar moeten zijn.

Voorbeeld:

```
public class MijnKlasse
{
    public int Waarde; // Toegankelijk vanuit andere klassen en assemblies.
}
```

`protected`

- Toegankelijkheid: Alleen toegankelijk binnen dezelfde klasse en afgeleide klassen (subclasses).
- Gebruik: Voor details die alleen nuttig zijn voor de klasse zelf en klassen die ervan erven.

Voorbeeld:

```
class BasisKlasse
{
    protected int waarde; // Beschikbaar in afgeleide klassen.
}

class AfgeleideKlasse : BasisKlasse
{
    void ToonWaarde()
    {
        Console.WriteLine(waarde); // Toegankelijk omdat het 'protected' is.
    }
}
```

`internal`

- Toegankelijkheid: Alleen binnen dezelfde **assembly**.
- Gebruik: Voor onderdelen die binnen een project beschikbaar moeten zijn, maar niet daarbuiten.

Voorbeeld:

```
internal class MijnKlasse
{
    internal int Waarde; // Alleen toegankelijk binnen dezelfde assembly.
}
```

`protected internal`

- Toegankelijkheid: Zowel binnen dezelfde **assembly** als in afgeleide klassen buiten de assembly.
- Combinatie van `protected` en `internal`.
- Gebruik: Als je zowel afgeleide klassen als toegang binnen de assembly nodig hebt.

Voorbeeld:

```
public class MijnKlasse
{
    protected internal int Waarde; // Toegankelijk in afgeleide klassen of dezelfde
assembly.
}
```

`private protected`

- Toegankelijkheid: Alleen binnen dezelfde klasse of afgeleide klassen **in dezelfde assembly**.
- Combinatie van `private` en `protected`.
- Gebruik: Om extra restrictie te leggen op `protected`.

Voorbeeld:

```
class MijnKlasse
{
    private protected int Waarde; // Alleen toegankelijk binnen afgeleide klassen in
dezelfde assembly.
}
```

19.1. Principe van het minste voorrecht

Dit principe ('Principle of least privilege') geeft o.a. aan dat een onderdeel slechts die visibiliteit mag krijgen nodig om zijn taak te volbrengen. Als een onderdeel een bepaalde visibiliteit niet nodig heeft, zou ze deze niet mogen hebben.

Maak initieel alles private. Indien je een geldige reden hebt kan je deze visibiliteit aanpassen.

Een ander voorbeeld is het `readonly` keyword: denk je dat een variabele slechts éénmalig dient ingesteld te worden, maak het dan `readonly`.

> In a general sense, “things” should have the capabilities they need to get their job done, but no more. An example is the scope of a variable. A variable should not be visible when it’s not needed.
>
> — Java™ How To Program (Early Objects)

19.2. Samenvatting

`private`	**Alleen binnen dezelfde klasse.**
`public`	Overal toegankelijk, zonder beperkingen.
`protected`	Alleen binnen dezelfde klasse en afgeleide klassen.
`internal`	Alleen binnen dezelfde assembly.
`protected internal`	Binnen dezelfde assembly én in afgeleide klassen buiten de assembly.
`private protected`	Alleen binnen dezelfde klasse of afgeleide klassen in dezelfde assembly.

Met deze modifiers kun je nauwkeurig bepalen wie toegang heeft tot je codeonderdelen en zorgen voor **encapsulation** en **veiligheid**.