1.0.0

H12: Geheugenmanagement bij klassen

value en reference met eigen objecten

Value types vs reference types

Twee soorten datatypes

Je gegevens in een C#-programma zijn altijd van een bepaald type: `string`, `int`, `DateTime`, `Student`, wat dan ook. Je moet voortdurend nadenken over welke datatype je aan het gebruiken bent. Dit is niet alleen belangrijk om te weten welke methoden en attributen je mag gebruiken. Je moet het ook weten om te begrijpen wat gebeurt als je een waarde toekent of als je een waarde meegeeft als argument van een methode.

Er zijn namelijk twee mogelijkheden:

- bij **value** types overschrijf je de data zelf wanneer je een toekenning doet en geef je een **kopie** van je data mee aan de methode
  - Deze types kunnen standaard niet de waarde `null` aannemen. Als je een variabele van een van deze types maakt en niet toekent, krijgt hij een defaultwaarde, zoals `0`, `false`,...
- bij **reference** types noteer je een geheugenadres wanneer je een toekenning doet en geef je een geheugenadres mee wanneer je een methode oproept
  - deze types kunnen (standaard) de waarde `null` aannemen, wat betekent dat er geen adres voor de data is. Dit is ook de defaultwaarde voor alle reference types.

Dit heeft belangrijke gevolgen. Als je dit systeem niet begrijpt, ga je gegarandeerd bugs in je code zien. Als voorbeeld zullen we het uitvoeren van een methode vergelijken met "iets noteren op papier" in het echte leven. Afhankelijk van de situatie wil je dan met een kopie of met een adres voor het origineel werken.

- Op een toets krijgt iedereen een blad met de vragen en vult hij/zij persoonlijke antwoorden in. Elke toets is individueel. Het is niet zo dat het antwoord van persoon 1 zichtbaar mag zijn voor persoon 2. We nemen dan ook geen toetsen af in Google docs.
- Wanneer er een geboortekaartje moet ondertekend worden, wordt er op de werkvloer vaak een e-mail uitgestuurd waarin staat in welk lokaal het kaartje ligt. Alle aanpassingen komen samen op hetzelfde kaartje.

We vragen niet aan iedereen om dezelfde toets in te vullen en we voorzien geen geboortekaartje per persoon die ondertekent. Je ziet dus andere resultaten wanneer je een handeling telkens uitvoert met een **kopie** dan wanneer je ze uitvoert met een **verwijzing** naar het origineel.

Klassen zijn reference types

Onze eigen klassen zijn **reference types**. Dat wil zeggen dat, in de ruimte die voorzien wordt wanneer we een variabele van een bepaalde klasse declareren, er een **verwijzing** wordt bijgehouden. Zo'n verwijzing is een adres voor de bytes die ons object vormen. Dit is in tegenstelling tot **value** types. Daarvoor wordt de waarde zelf bijgehouden op de plaats die voorzien is voor de variabele. De meeste types die je in het begin gezien hebt, zijn value types: `int` (en varianten), `boolean`, `float` (en varianten), `enum` types.

Demonstratie: leeftijd als onderdeel van een klasse en als losse variabele

```
public class Student {
    public int Leeftijd = 18;
}

public class Program {
    public static void VerhoogLeeftijd(int leeftijd) {
        leeftijd += 1;
    }
    public static void VerhoogLeeftijd(Student student) {
        student.Leeftijd += 1;
    }
    public static void Main() {
        int leeftijdAlsInt = 18;
        Student student = new Student();
        student.Leeftijd = 18;
        VerhoogLeeftijd(leeftijdAlsInt);
        VerhoogLeeftijd(student);
        Console.WriteLine(leeftijdAlsInt);
        Console.WriteLine(student.Leeftijd);
    }
}
```

Deze code produceert toont twee verschillende waarden: `18` en `19`.

We verklaren de 18 als volgt:

- `leeftijdAlsInt` is een `int`, dus een value type, waaraan we de waarde 18 geven
- als we `VerhoogLeeftijd` met een `int` als parameter oproepen, maken we dus een kopie van `18` en geven we die kopie mee aan de methode. Het is alsof we de methode een eigen exemplaar van de toets geven. In de body van de methode is "leeftijd" dus een naam om de kopie aan te duiden.
- als we `leeftijd` met 1 verhogen (via `leeftijd += 1` ofwel `leeftijd = leeftijd + 1`) koppelen we de naam "leeftijd" aan een nieuwe waarde. **Dat heeft geen enkel effect op `leeftijdAlsInt`, waarvan we bij oproep van `VerhoogLeeftijd` een kopie hadden genomen.**
- de `Console.WriteLine` toont de originele leeftijd en die is nooit aangepast

We verklaren de 19 als volgt:

- `Student` is een zelf geschreven klasse, dus een reference type

- als we `student.Leeftijd` instellen op 18, gaan we eerst op zoek naar de geheugenlocatie waar de gegevens over student zijn bijgehouden en overschrijven we daar de bytes die de leeftijd voorstellen.
- als we `VerhoogLeeftijd` met `student` als parameter aanroepen, vertellen we de methode waar in het geheugen de informatie over `student` gevonden kan worden. Het is alsof we zeggen: "Het geboortekaartje (de student) ligt in de refter".
- `VerhoogLeeftijd` gaat de leeftijd dus aanpassen bij de bron: de bytes die overschreven worden, zijn dezelfde die we de eerste keer hebben ingevuld.
- de `Console.WriteLine` gaat naar het adres waar de bytes van het `Student` object zich bevinden en haalt daar de leeftijd op: deze zijn in de vorige stap aangepast.

Demonstratie: wat als klassen value types waren?

Het feit dat klassen reference types zijn, heeft praktische gevolgen. We zullen dit demonstreren door dezelfde functionaliteit te implementeren met een `class` en een `struct`. Je kan `struct` zien als bijna hetzelfde als `class`, in die zin dat je er ook objecten van kan maken, maar `struct`-objecten zijn value types.

> ⓘ We gaan je nergens in deze cursus vragen zelf een `struct` te maken. We gebruiken ze alleen omdat ze het verschil tussen value en reference duidelijker kunnen maken.

Vergelijk volgende twee vereenvoudigde varianten op `DateTime`:

```
struct MiniDatumValue
{
    public int Dag;
    public int Maand;
    public int Jaar;
}

class MiniDatumReference
{
    public int Dag;
    public int Maand;
    public int Jaar;
}
```

Beide stellen een datum voor en hebben dezelfde attributen. We voorzien ook een methode `WijzigDatums`, met twee parameters: één voor ons value type en één voor ons reference type. Ten slotte voorzien we een demonstratie in `Main`:

```
static void Main(string[] args) {
    MiniDatumValue d1 = new MiniDatumValue();
    d1.Dag = 6;
    d1.Maand = 3;
    d1.Jaar = 2016;
    MiniDatumValue d2 = new MiniDatumReference();
    d2.Dag = 6;
    d2.Maand = 3;
    d2.Jaar = 2016;
    Program.WijzigDatums(d1,d2);
    Console.WriteLine($"value na uitvoering: {d1.Dag}/{d1.Maand}/{d1.Jaar}");
    Console.WriteLine($"reference na uitvoering: {d2.Dag}/{d2.Maand}/{d2.Jaar}");
}

static void WijzigDatums(MiniDatumValue val, MiniDatumReference reference)
{
    val.Maand = 2;
    reference.Maand = 2;
}
```

Als je dit programma uitvoert, merk je dat de waarde van `d1` niet gewijzigd is door de methode en `d2` wel. Dat komt omdat val een kopie bevatte van de waarde van `d1`, terwijl reference naar dezelfde data verwees als `d2`.

nullable value types

> Kennisclip **(met demonstratie in SchoolAdmin, zelf mee te maken!)**

Betekenis van null

Normaal gezien kom je `null` tegen wanneer je een variabele van een reference type hebt zonder verwijzing naar data. Het gevolg is dat `null` vaak betekent dat er een waarde zou kunnen staan, maar in de huidige situatie geen geldige waarde is.

Hoewel value types niet werken met verwijzingen, zou dezelfde interpretatie ook bij value types zinvol kunnen zijn: soms heb je gewoonweg geen geldige waarde. Daarom kent C# ook **nullable value types**. Dit zijn speciale value types die ook de waarde `null` kunnen aannemen (ook al kom je die laatste anders vooral tegen bij reference types).

Je noteert een nullable value type als een gewoon value type, gevolgd door een vraagteken. Indien je in code bijvoorbeeld een getalwaarde wil voorstellen als een variabele `int mijnVariabele`, maar de mogelijkheid bestaat dat er geen waarde is voor `mijnVariabele`, declareer je als volgt: `int? mijnVariabele`. Dit betekent: "`mijnVariabele` is een getal, maar kan ontbreken."

Dit heeft gevolgen. Je kan code die een value type verwacht niet zomaar gebruiken met een nullable versie van hetzelfde type. Anders gezegd: je mag `mijnVariabele` niet meegeven aan een methode die een gewone `int` verwacht. Je moet ofwel deze methode aanpassen zodat ze een `int?` verwacht, ofwel moet je `mijnVariabele` casten voor je hem meegeeft als argument. Let op: dit werkt alleen als `mijnVariabele` niet `null` is!

Operaties met `null`

Voor de nullable versies van de value types die je al kent, kan je gekende operaties (zoals `+`, `-`, ... voor getallen) blijven gebruiken, maar je moet opletten. Een berekening met `null` in levert je sowieso `null` op als resultaat. Een vergelijking (via `<=`, `<`, `>`, `>=`) met `null` levert je sowieso `false` op als resultaat.

NullReference exception

> Kennisclip voor deze inhoud

Null en NullReferenceException

Zoals nu duidelijk is bevatten variabelen van een reference type steeds een referentie naar een object. Maar wat als we dit schrijven:

```
Student stud1;
// Visual Studio staat niet toe een programme met deze code uit te voeren
// maar je kan het via command line wel doen
stud1.Naam = "Test";
```

Dit zal een fout geven. `stud1` bevat namelijk nog geen referentie. Maar wat dan wel?

Deze variabele bevat de waarde **`null`** . Dit is de defaultwaarde voor reference types. Met andere woorden, als je een reference type declareert en niet initialiseert, zal de waarde `null` zijn.

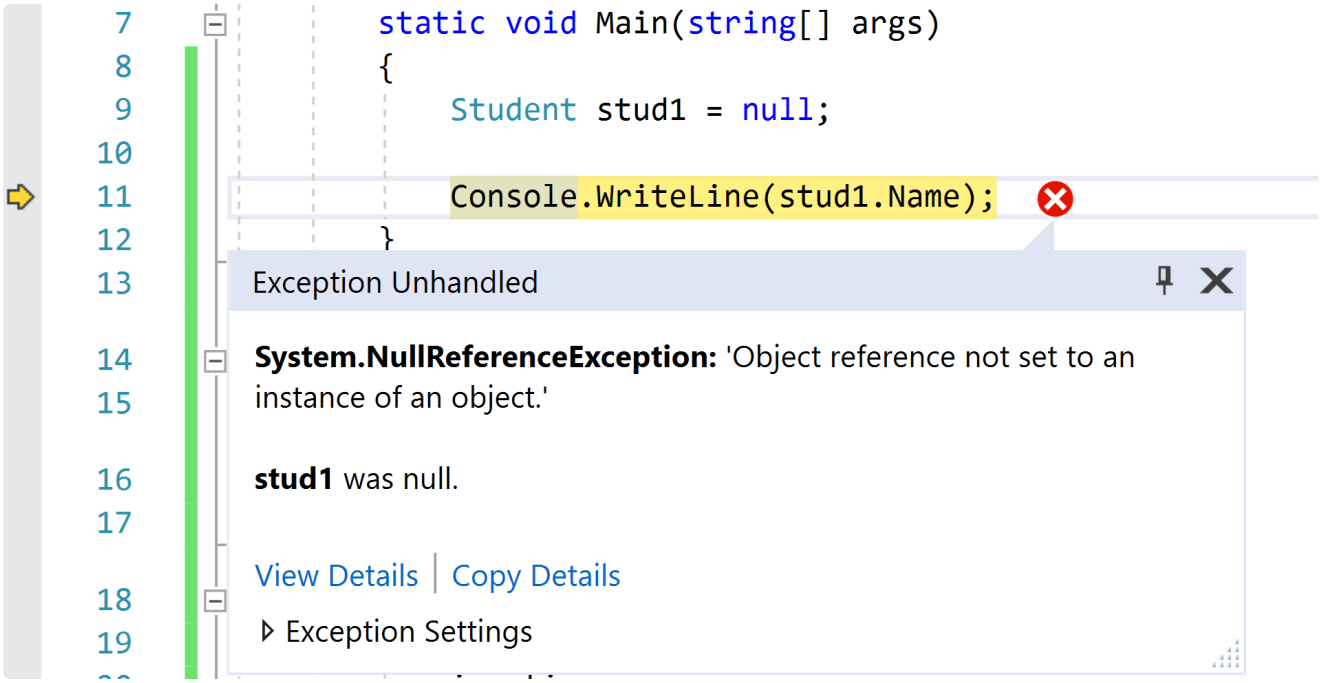
NullReferenceException

Een veel voorkomende foutboodschap tijdens de uitvoer van je applicatie is de zogenaamde `NullReferenceException` . Deze zal optreden wanneer je code een **member** (attribuut, methode of property) van `null` probeert op te vragen.

Laten we dit eens simuleren:

```
Student stud1 = null;
Console.WriteLine(stud1.Name);
```

Dit zal resulteren in volgende foutboodschap:



NullReferenceException error in VS

> We moeten in dit voorbeeld expliciet `= null` plaatsen daar Visual Studio slim genoeg is om je te waarschuwen voor eenvoudige potentiële NullReference fouten en je code anders niet zal compileren.

NullReferenceException voorkomen

Je kan `NullReferenceException` voorkomen door na te gaan dat een object verschillend is van null vooraleer je een van de members van dit object probeert te gebruiken. Bijvoorbeeld, met een klasse `Auto`:

```
static void Main() {
    Auto auto1 = new Auto();
    Auto auto2 = null;
    Console.WriteLine($"{auto1.GeefKilometerstand()}km");
    if (!(auto2 is null)) {
        Console.WriteLine($"{auto2.GeefKilometerstand()}km");
    }
    else {
        Console.WriteLine($"auto2 heeft geen waarde, kilometerstand opvragen zou crashen");
    }
}
```

Deze code zal niet crashen. Als je de `WriteLine` uitvoert zonder if, zal het programma wel crashen met een `NullReferenceException`.

> Waarom `is null` en niet `== null` ? Die vraag leidt ons te ver. Meestal zal `== null` ook werken, maar `==` kan aangepast worden om anders te werken dan gewoonlijk. `is` is dus betrouwbaarder.

Labo

SchoolAdmin project: alle cursussen opvolgen

Functionele analyse

We willen een lijst bijhouden met alle objecten van de klasse `Cursus`. Zorg dat deze lijst automatisch wordt ingevuld.

Technische analyse

Voorzie op klasseniveau een array met plaats voor 10 `Cursus` objecten en noem hem `AlleCursussen`. Zorg ervoor dat een cursus bij aanmaak in de eerste vrije positie in deze array wordt geplaatst. Schrijf hiervoor een private hulpmethode `registreerCursus(Cursus cursus)`. Deze kan je uitvoeren zonder bestaand `Cursus` object. Ze gaat op zoek naar de eerste vrije positie in de array en slaat `cursus` op op deze positie.

Je kan `registreerCursus` als volgt implementeren:

- start met een variabele `vrijePositie` van type `int?` met waarde `null`
- controleer één voor één de posities in de array
  - onthoud de eerste positie waarop je `null` tegenkomt
- controleer nadat de array doorlopen is of er nog een vrije positie is
  - zo ja, sla de cursus daar op
  - zo nee, print `"Er zijn geen vrije posities meer"`

> Commit je aanpassingen.

SchoolAdmin project: cursussen opzoeken op Id

Functionele analyse

We willen cursussen makkelijk kunnen opvragen via Id. Schrijf een methode `ZoekCursusOpId` die dit doet.

Technische analyse

Deze methode werkt op klasseniveau, want je hebt geen cursus nodig om een andere cursus te vinden. Ze heeft één parameter, `id`. Het return type is `Cursus`, maar het kan zijn dat je geen cursus met het gevraagde Id kan terugvinden.

De methode werkt door `AlleCursussen` element per element te doorlopen en de waarde van het attribuut `Id` van elk element te vergelijken met het meegegeven argument. Als de gevraagde cursus niet bestaat, mag je programma niet crashen, maar krijg je `null` terug.

> Commit je aanpassingen.

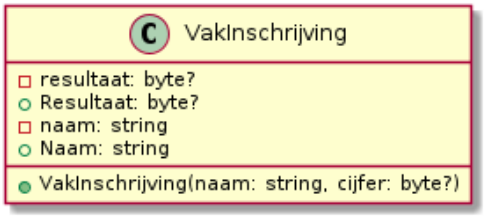
SchoolAdmin project: gelinkte objecten

Functionele analyse

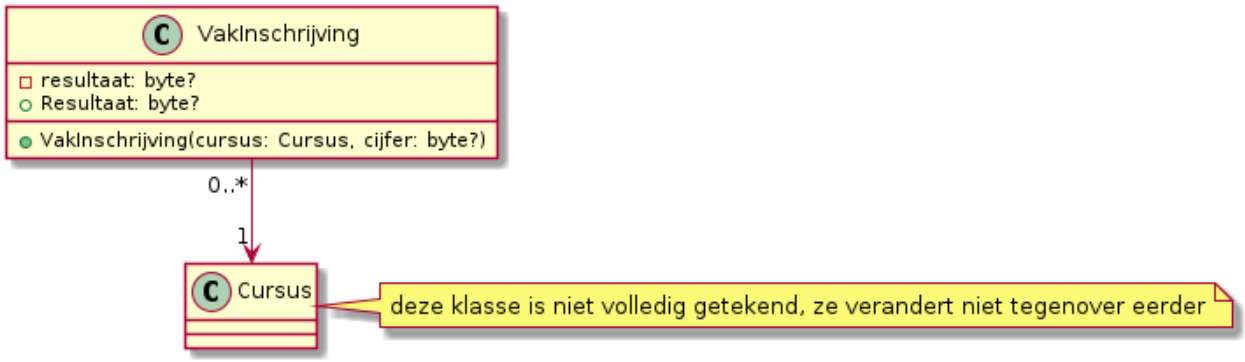
Het is niet handig dat onze klasse `VakInschrijving` een cursus voorstelt als string. Dat maakt dat we de functionaliteit van `Cursus` niet kunnen gebruiken. Pas daarom VakInschrijving aan zodat de klasse echt gelinkt is aan `Cursus`. Dit vereist aanpassingen op een aantal plaatsen.

Technische analyse

Voor de aanpassing heb je dit:



Erna heb je dit:



Controleer ook dat al je testmethodes nog dezelfde resultaten leveren als eerder.

> Commit je aanpassingen.

SchoolAdmin project: Studieprogramma (stap 1)

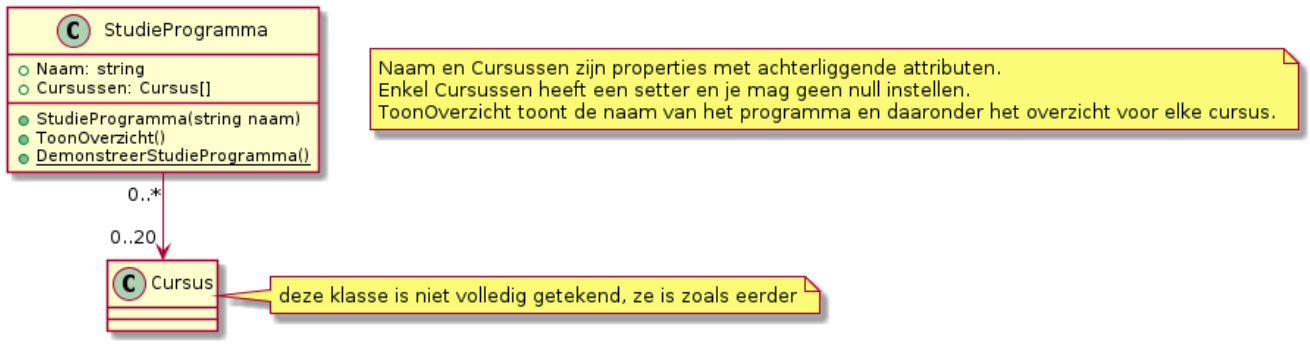
We wensen cursussen te groeperen in studieprogramma's.

Functionele analyse

Schrijf een klasse `StudieProgramma`. Deze heeft een naam, bevat een aantal cursussen en kan getoond worden op het scherm.

Technische analyse

Implementeer als volgt:



Gebruik volgende code voor de demonstratiemethode:

```
Cursus communicatie = new Cursus("Communicatie");
Cursus programmeren = new Cursus("Programmeren");
Cursus databanken = new Cursus("Databanken", new Student[7], 5);
Cursus[] cursussen = { communicatie, programmeren, databanken };
StudieProgramma programmerenProgramma = new StudieProgramma("Programmeren");
StudieProgramma snbProgramma = new StudieProgramma("Systeem- en netwerkbeheer");
programmerenProgramma.cursussen = cursussen;
snbProgramma.cursussen = cursussen;
// later wordt Databanken geschrapt uit het programma SNB
snbProgramma.cursussen[2] = null;
programmerenProgramma.ToonOverzicht();
snbProgramma.ToonOverzicht();
```

Hier loopt iets mis. Benoem zelf de oorzaak en corrigeer de fout.

> ⓘ Commit je aanpassing.

Schooladmin project: Studieprogramma (stap 2)

Zoals boven, maar gebruik nu volgende code voor de demonstratiemethode:

```
Cursus communicatie = new Cursus("Communicatie");
Cursus programmeren = new Cursus("Programmeren");
Cursus databanken = new Cursus("Databanken", new Student[7], 5);
Cursus[] cursussen1 = { communicatie, programmeren, databanken };
Cursus[] cursussen2 = { communicatie, programmeren, databanken };
StudieProgramma programmerenProgramma = new StudieProgramma("Programmeren");

StudieProgramma snbProgramma = new StudieProgramma("Systeem- en netwerkbeheer");
programmerenProgramma.cursussen = cursussen1;
snbProgramma.cursussen = cursussen2;
// later wordt Databanken geschrapt uit het programma SNB
// voor SNB wordt bovendien Programmeren hernoemd naar Scripting
snbProgramma.cursussen[2] = null;
snbProgramma.cursussen[1].Titel = "Scripting";
programmerenProgramma.ToonOverzicht();
snbProgramma.ToonOverzicht();
```

Opnieuw loopt het fout. Benoem zelf de oorzaak en corrigeer de fout.

> Commit je aanpassing.