

1.0.0

H15: Geavanceerde overerving

protected access modifier

De `private` ("enkel toegankelijk binnen code van deze klasse") en `public` ("toegankelijk van overal") access modifier stellen twee extremen voor met betrekking tot encapsulatie. Wanneer je overerving toepast, is dit niet altijd ideaal.

We nemen volgende code als voorbeeld:

```
public abstract class Persoon
{
    private DateTime geboorteDatum;
    private string naam;
    public string Naam
    {
        get
        {
            return this.naam;
        }
    }
    public virtual string NaamKaartje()
    {
        DateTime nu = DateTime.Now;
        int verschilJaren = nu.Year - this.geboorteDatum.Year;
        if (nu.Month < this.geboorteDatum.Month || nu.Month == geboorteDatum.Month && nu
        {
            verschilJaren -= 1;
        }
        return $"{this.Naam} ({verschilJaren})";
    }
}
```

Deze code kan gebruikt worden om een naamkaartje voor een persoon te genereren, met daarop de naam en tussen haakjes de leeftijd van die persoon. De leeftijd is volledig privé (er is ook geen property om hem op te vragen, in tegenstelling tot de naam). Dat betekent dat we niet, bijvoorbeeld, het volgende kunnen doen:

```


public class Student : Persoon
{
    public override string NaamKaartje()
    {
        string output = $"Hallo, mijn naam is {this.Naam}";
        DateTime nu = DateTime.Now;
        if (nu.Day == this.geboorteDatum.Day && nu.Month == this.geboorteDatum.Month)
        {
            output += "\nHet is mijn verjaardag!";
        }
        return output;
    }
}

```

Dit zou een speciaal naamkaartje afprinten voor studenten op hun verjaardag. Maar code binnen de klasse `Student` kan niet aan de verjaardag, want die is `private` en `Student` is een andere klasse dan `Persoon`. Mogelijk is het **geen** optie om een publieke getter te voorzien voor de verjaardag, omdat het niet de bedoeling is dat andere klassen dan de subklassen van `Persoon` er gebruik van maken: hoe meer encapsulatie, hoe liever.

Daarom is er een tussenoplossing: de `protected` access modifier. Deze zorgt ervoor dat een bepaald deel van een klasse zich als `private` gedraagt naar de buitenwereld, maar als `public` naar de eigen kindklassen. Zo wordt encapsulatie geen alles-of-iets verhaal:

Met andere woorden: als je in bovenstaande code het veld `geboorteDatum` `protected` maakt in plaats van `private`, wordt je code wel uitvoerbaar.

 Ten slotte hernoem je best `geboorteDatum` naar `GeboorteDatum`. De algemene conventie is dat ook `protected` members Pascal case gebruiken.

Base keyword

Het **base** keyword laat ons toe om bij een overridden methode of property in de child-klasse toch te verplichten om de parent-implementatie toe te passen.


Stel dat we volgende 2 klassen hebben in de software van een gastronomisch bedrijf dat restaurants en frituren uitbaat:

```
class Restaurant
{
    protected int Kosten=0;
    public virtual void PoetsAlles()
    {
        Kosten+=1000;
    }
}

class Frituur:Restaurant
{
    public override void PoetsAlles()
    {
        Kosten+= (1000 + 500);
    }
}
```

Het poetsen van een `Frituur` is duurder (1000 basis + 500 voor ontsmetting) dan een gewoon restaurant. Als we echter later beslissen dat de basisprijs (in `Restaurant`) moet veranderen dan moet je ook in alle child-klassen doen. `base` lost dit voor ons. De `Frituur`-klasse herschrijven we naar:

```
class Frituur:Restaurant
{
    public override void PoetsAlles()
    {
        base.PoetsAlles(); //eerste basiskost wordt opgeteld
        Kosten+=500; //kosten eigen aan frituur worden bijgeteld.
    }
}
```

 Dit lijkt sterk op de `base` waarmee je een ouderconstructor kan oproepen, maar deze `base` voor gewone methodes staat in de body, niet na een speciale dubbele punt. Deze `base` hoeft niet de eerste regel van de body te zijn.

System.Object

System.Object

Alle types in C# zijn afstammelingen van de `System.Object` klasse. Indien je een klasse schrijft zonder een expliciete parent dan zal deze steeds `System.Object` als rechtstreekse parent hebben. Ook afgeleide klassen stammen dus af van `System.Object`. Concreet wil dit zeggen dat alle klassen `System.Object`-klassen zijn en dus ook de bijhorende functionaliteit ervan hebben.

⚠ Merk op dat we hier niet alleen onze eigen klassen bedoelen, maar alle types, dus zelfs `int`, `bool`, `string`, ... **Alle** types stammen (al dan niet rechtstreeks) af van `System.Object`.

Indien je de `System` namespace in je project gebruikt door bovenaan `using System;` te schrijven dan hoef je dus niet altijd `System.Object` te schrijven maar mag je ook **`Object`** schrijven.

Hoe ziet System.Object er uit?

Wanneer je een lege klasse maakt dan zal je zien dat instanties van deze klasse reeds een aantal methoden ingebouwd hebben. Dit komt omdat deze methoden gedefinieerd zijn in de `System.Object` klasse en meteen overgeërfd worden door je nieuwe klasse:

Methode	Beschrijving
<code>Equals(Object o)</code>	Gebruikt om te ontdekken of twee instanties "ge-equals" zijn. Wat dit betekent kan bepaald worden door de auteur van de klasse.
<code>GetHashCode()</code>	Geeft een unieke code (hash) terug van het object. Nuttig om o.a. snel te sorteren.
<code>GetType()</code>	Geeft het type (of klasse) van het object terug. Dit is een object van het type <code>Type</code> !
<code>ToString()</code>	Geeft een string terug die het object voorstelt.

i Er zijn er nog een paar, maar de rest ga je minder vaak tegenkomen.

GetType()

Stel dat je een klasse `Student` hebt gemaakt in je project. Je kan dan op een object van deze klasse de `GetType()`-methode aanroepen om te weten wat het type van dit object is:

```
Student stud1 = new Student("Wolfgang Amadeus Mozart");
Console.WriteLine(stud1.GetType());
```

Dit zal als uitvoer de namespace gevolgd door het type op het scherm geven. Als je klasse dus in de namespace `StudentManager` staat, zal er verschijnen: `StudentManager.Student`.

Wil je enkel het type zonder namespace dan is het nuttig te beseffen dat `GetType()` een object teruggeeft van het type `Type` met meerdere eigenschappen, waaronder `Name`. Volgende code zal dus enkel `Student` op het scherm tonen:

```
Student stud1 = new Student("Wolfgang Amadeus Mozart");
Console.WriteLine(stud1.GetType().Name);
```

⚠ Deze methode is vooral nuttig in code voor frameworks en dergelijke. Dat wil zeggen: code waaraan je jouw eigen code kan toevoegen. In een meer typische eigen applicatie zou je hier niet te veel gebruik van hoeven te maken, anders schort er waarschijnlijk iets aan je ontwerp.

ToString()

Deze is de nuttigste waar je al direct leuke dingen mee kan doen. Wanneer je schrijft:

```
Console.WriteLine(stud1);
```

wordt je code eigenlijk herschreven naar:

```
Console.WriteLine(stud1.ToString());
```

Op het scherm verschijnt dan `StudentManager.Student`. Waarom? Wel, de methode `ToString()` wordt in `System.Object()` ongeveer als volgt beschreven:

```
public virtual string ToString()
{ return GetType().ToString(); }
```

Merk twee zaken op:

1. `GetType` wordt aangeroepen en die output krijg je terug.
2. De methode is **virtual** gedefinieerd.

De hierboven vermelde methoden (behalve `GetType`) in `System.Object` zijn `virtual`, en je kan deze overschrijven!

ToString() overriden

Het zou natuurlijk fijner zijn dat de `ToString()` van onze student nuttigere info teruggeeft, zoals bv de interne Naam (string autoproperty) en Leeftijd (int autoproperty). We kunnen dat eenvoudig krijgen door gewoon `ToString` to overriden:

```
class Student
{
    public int Leeftijd {get;set;}
    private string naam;
    public string Naam {get;}

    public Student(string naam) {
        this.naam = naam;
    }

    public override string ToString()
    {
        return $"Student genaamd {Naam} (Leeftijd:{Leeftijd})";
    }
}
```

Wanneer je nu `Console.WriteLine(stud1);` zou schrijven, dan wordt je output bijvoorbeeld: `Student Wolfgang Amadeus Mozart (Leeftijd:35)`.

Equals()

Ook deze methode kan je dus overriden om twee objecten met elkaar te vergelijken. Hierbij moet je een **applicatiespecifiek** antwoord kunnen geven op de vraag "wanneer zijn twee objecten aan elkaar gelijk?"

```
if(stud1.Equals(stud2))
    //...
```

De `Equals` methode heeft dus als signatuur: `public virtual bool Equals(Object o)`. .NET maakt volgende afspraken voor een geldige implementatie van `Equals`, maar het is aan jou om te zorgen dat je code deze afspraken volgt:

- Het moet `false` teruggeven indien het argument `o` `null` is
- Het moet `true` teruggeven indien je het object met zichzelf vergelijkt (bv `stud1.Equals(stud1)`)
- Het mag enkel `true` teruggeven als volgende statements beide waar zijn:

```
stud1.Equals(stud2);
stud2.Equals(stud1);
```

- Indien `stud1.Equals(stud2)` `true` teruggeeft en `stud1.Equals(stud3)` ook `true` is, dan moet `stud2.Equals(stud3)` ook `true` zijn.

⚠️ Volgt je eigen code deze afspraken niet, dan krijg je geen compilatiefouten, maar dan kan je wel onverwacht gedrag krijgen van code die gebruik maakt van `Equals` zoals bijvoorbeeld de `IndexOf`-methode van `List<T>`.

Equals overriden

Stel dat we vinden dat een student gelijk is aan een andere student indien z'n `Naam` en `Leeftijd` dezelfde zijn, we kunnen dan de `Equals`-methode overriden als volgt:

```
//In de Student class
public override bool Equals(Object o)
{
    if (o is null) { return false; }
    bool gelijk;
    if(GetType() != o.GetType())
        gelijk=false;
    else
    {
        Student temp = (Student) o; //Zie opmerking na code!
        if(Leeftijd == temp.Leeftijd && Naam == temp.Naam) {
            gelijk=true;
        }
        else {
            gelijk=false;
        }
    }
    return gelijk;
}
```

De lijn `Student temp = (Student) o;` zal het object `o` casten naar een `Student`. Doe je dit niet dan kan je niet aan de interne `Student`-variabelen van het object `o`. Het feit dat een object met het statische type (d.w.z.: zo staat het bij de parameter) `Object` tijdens de uitvoering ook een object met runtime type `Student` (d.w.z.: dat is het soort data dat op de heap staat) kan zijn is een voorbeeld van polymorfisme.

Het verschil met `==`

`Equals` en `==` doen verschillende zaken. `==` is strenger. Het controleert volgende zaken:

- Voor de meeste reference types: gaat het om exact dezelfde data (d.w.z. hetzelfde adres op de heap)?
- Voor strings: gaat het om dezelfde tekst?
- Voor value types: is de waarde aan de linkerkant een kopie van die aan de rechterkant?

GetHashCode()

`GetHashCode()` zorgt ervoor dat je een "vingerafdruk" van een object krijgt. Als twee objecten gelijk zijn onder `Equals`, wordt verondersteld dat ze dezelfde vingerafdruk hebben. Omgekeerd geldt niet, maar het is voor de efficiëntie wel beter als verschillende objecten ook verschillende vingerafdrukken leveren. Het is opnieuw aan de programmeur om dit te garanderen.

Een efficiënte hashfunctie voor een type `K` zorgt er bijvoorbeeld voor dat opzoeken van keys in een `Dictionary<K,V>` erg snel verlopen. Een minder efficiënte hashfunctie zal opzoeken in datzelfde `Dictionary` trager laten verlopen. Een foute hashfunctie kan er voor zorgen dat je `Dictionary` niet meer werkt zoals verwacht.

Wij behandelen de theorie achter goede hashfuncties hier niet. Je kan gewoon deze vuistregel onthouden: als je een nieuwe hashfunctie moet voorzien (normaal omdat je `Equals` hebt overschreven; de compiler waarschuwt je hier ook voor), kan je de hashwaarde van een onderdeel laten berekenen dat gelijk is voor twee objecten die gelijk zijn, op voorwaarde dat dat onderdeel niet verandert. Dat komt omdat je niet wil dat een object nog van hashcode verandert nadat je het in een `Dictionary`, `HashSet` of andere structuur hebt geplaatst.

Bijvoorbeeld:

```
//In de Student class
public override int GetHashCode()
{
    return this.Naam.GetHashCode();
}
```

In ons systeem is dit voldoende, want:

- als twee studenten gelijk zijn onder `Equals`, is hun naam gelijk (en dus de hashcode van hun naam)
- `Naam` wordt ingesteld bij constructie en kan daarna niet veranderd worden op basis van de code die we hebben

Labo

h15-bugfix

Functionele analyse

We schrijven een bestelsysteem. We kunnen gewone bestellingen en internationale bestellingen plaatsen. Voor geïmporteerde producten wordt een extra toelage van 10% aangerekend, maar is er wel korting voor grote bestellingen. We willen de prijzen van onze producten niet publiek zichtbaar maken, want dat verhindert prijsafspraken.

Technische analyse

Schrijf een klasse `Bestelling` met een `uint` property `Aantal` en een `double` **privé**-attribuut `basisPrijs`. Voorzie ook een overschrijfbare property `TotaalPrijs`, namelijk het aantal maal de basisprijs. Schrijf **daarna** een subklasse `InternationaleBestelling` die de totaalprijs bepaalt door de basisprijs met 10% te verhogen, maar vanaf 100 stuks een vlakke korting van 1000 euro toepast. **Dit zal niet meteen werken!** Doe een zo klein mogelijke aanpassing om het toch te doen werken.

Schrijf een methode `DemonstreerBestellingen` in de klasse `Overerving`. Hierin vraag je of de gebruiker een gewone of internationale bestelling wil plaatsen, vraag je om het aantal en de basisprijs en toon je dan de totaalprijs.

Voorbeeldinteractie

```
Aantal stuks?  
> 4  
Basisprijs?  
> 5  
Gewone bestelling (1) of internationale bestelling (2)?  
> 1  
Totaalprijs: 20
```

h15-pizza

Functionele analyse

We schrijven software om bestellingen van pizza's op te volgen. Deze software spreekt met andere software, bijvoorbeeld van Deliveroo of Uber Eats. We willen niet dat die diensten iets kunnen aanpassen aan de ingrediënten van onze pizza's, maar we willen wel zelf wel allerlei pizza's kunnen samenstellen.

Technische analyse

Je krijgt volgende klasse `Pizza` :

```
abstract class Pizza {
    private List<string> ingredienten;

    public Pizza(string[] extraToppings) {
        this.ingredienten = new List<string> { "deeg", "tomatensaus", "kaas" };
        foreach(var topping in extraToppings) {
            this.ingredienten.Add(topping);
        }
    }

    public abstract double BasisPrijs {
        get;
    }

    public double Prijs {
        get {
            return this.BasisPrijs + (this.ingredienten.Count * 0.5);
        }
    }

    public void ToonIngredienten() {
        foreach(var ingredient in ingredienten) {
            Console.WriteLine(ingredient);
        }
    }
}
```

Schrijf nu twee klassen `Margarita` en `Veggie` die overerven van `Pizza` , met basisprijs 5 en 6. Bij constructie krijgt een `Margarita` sowieso "mozzarella" toegevoegd aan de lijst met ingrediënten en krijgt een `Veggie` sowieso "tofu" en "spinazie", maar geen "kaas". Je moet hierbij een aanpassing doen aan `Pizza` , maar hou ze zo klein mogelijk. Het blijft de bedoeling dat een pizza standaard ook kaas bevat, dus schrijf je code zodat de veggie pizza dit ingrediënt verwijdert. Schrijf een demonstratiemethode `DemonstreerPizzas` in de klasse `Overerving` .

Voorbeeldinteractie

```
Een margarita zonder extra's kost: (hier het resultaat)
De ingredienten zijn:
(hieronder het effect van ToonIngredienten)
Een veggie zonder extra's kost:
De ingredienten zijn:
(hieronder het effect van ToonIngredienten)
```

h15-menukaart

Functionele analyse


We willen een digitale menukaart tonen in een online restaurant. Op deze kaart verschijnen gerechten in een standaardformaat. Kindergerechten volgen hetzelfde formaat, maar verschijnen in kleur.

Technische analyse

- Schrijf een klasse `Gerecht` met properties `Naam` en `Prijs` (deze laatste van type `double`)
 - De methode `ToonOpMenu` print de naam, gevolgd door 3 tabs, gevolgd door de prijs
- Schrijf een kindklasse `KinderGerecht`
 - Dit werkt hetzelfde als een gewoon gerecht, maar de weergave op het menu gebruikt een willekeurige kleur. Als we bijvoorbeeld het aantal tabs aanpassen naar 5, moet `KinderGerecht` zonder aanpassingen mee volgen.
 - Je kan een willekeurige kleur krijgen door een willekeurig getal tussen 1 en 15 te bepalen en dat dan te casten naar een waarde van de enum `ConsoleColor`.
- Maak een methode `DemonstreerGerechten` in de klasse `Overerving`. Hierin maak je een lijst met minstens 4 gerechten (waarvan minstens 2 kindergerechten) naar keuze en doorloop je de lijst zodat elk gerecht getoond wordt op het menu.

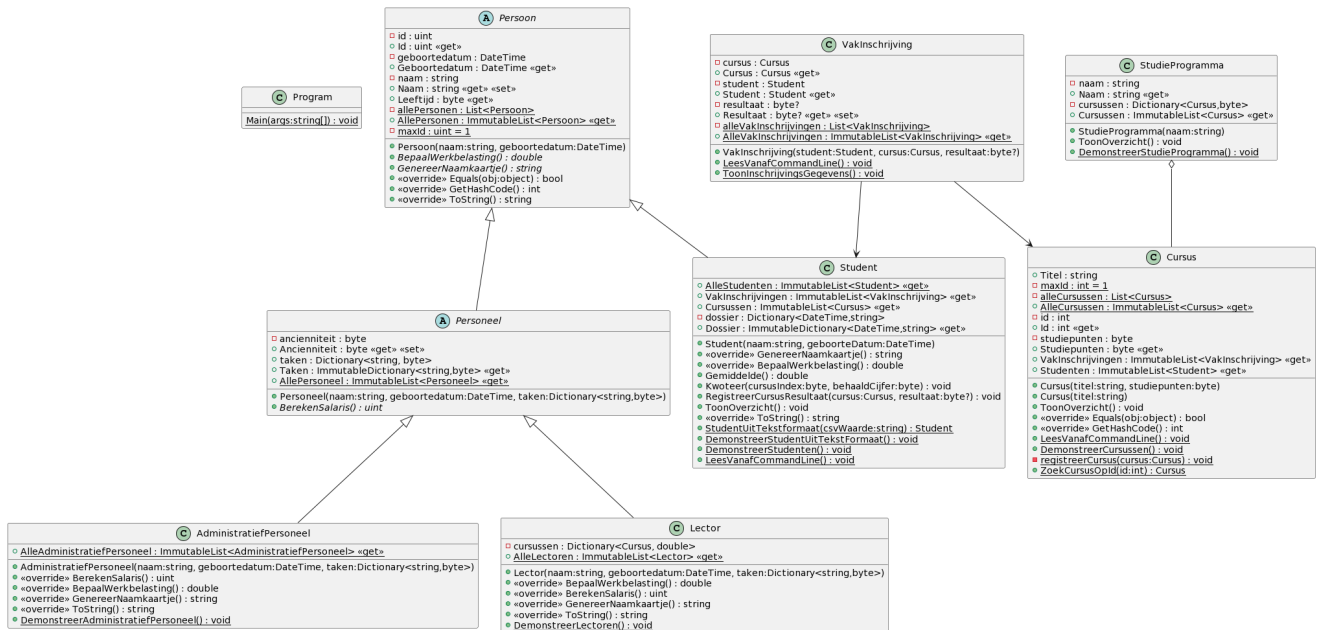
Voorbeeldinteractie

Paling in 't groen	22.00
Vol-au-vent	11.00(deze regel verschijnt in een willekeurige kleur)
Waterzooi	22.00
Kabouterschnitzel	12.00(deze regel verschijnt in een willekeurige kleur)

 Tabs zijn eigenlijk niet ideaal. Zoek, als je sneller klaar bent, uit hoe je stringformattering kan gebruiken om de naam van elk gerecht met exact 35 tekens weer te geven.

Uitbreidingen SchoolAdmin

In de volgende stappen, zullen we SchoolAdmin verder uitbouwen. Volgend diagram toont het hele project na al deze wijzigingen:



Vergelijkbare objecten

Voorzie `Persoon` en `Cursus` van een eigen versie van `Equals`. Hiermee zullen we later nagaan dat een van deze objecten niet dubbel voorkomt in de lijst met geregistreerde objecten.

Een persoon is gelijk aan een andere persoon met hetzelfde ID. Je hoeft hier niet na te gaan dat de objecten van exact hetzelfde type zijn. In plaats daarvan kan je schrijven: `if (obj is Persoon) { ... }`

Een cursus is gelijk aan een andere cursus met hetzelfde ID.

Voorzie ook overal een hash code volgens de vuistregel in de cursus.

ToString

Voorzie `Persoon` van een `ToString` methode die een resultaat van volgende vorm toont:

```

Persoon
-----
Naam: Wouter Roelants
Leeftijd: 43
  
```

i Voorzie in één keer een property `Leeftijd` die berekend wordt aan de hand van de huidige datum.

Zorg dat de concrete klassen hier ook het statuut van de persoon aan koppelen, bijvoorbeeld:

```
Persoon
-----
Naam: Geertrui Willems
Leeftijd: 51
Meerbepaald, administratief personeel
```

Doe dit niet met `GetType`, want dan is de schrijfwijze anders. Doe het met de hand per klasse.

Eenmaking statische lijsten personen

Je hebt momenteel volgende statische properties voor (immutable) lijsten met personen:

- `AllePersonen`
- `AlleLectoren`
- `AlleStudenten`
- `AllePersoneel`
- `AlleAdministratiefPersoneel`

Het is niet ideaal om al deze lijsten te hebben. Elke persoon wordt nu op twee of drie plaatsen bijgehouden, dus als je het systeem zou aanpassen om personen te verwijderen, moet je er aan denken dat op twee of drie plaatsen te doen. Als je klassen zoals `Gastlector`, `Uitwisselingsstudent` of `Roosterverantwoordelijke` zou toevoegen, zou je dat zelfs op nog meer plaatsen moeten doen.

Vervang daarom de lijsten voor de subklassen van `Persoon` zodat er geen achterliggend attribuut wordt bijgehouden. In plaats daarvan, moet de lijst met personen "on-the-fly" berekend worden. Met andere woorden, je moet nog steeds een getter `AlleLectoren` enzovoort voorzien, maar deze verzamelt alle lectoren door `AllePersonen` te doorlopen. Gebruik hier opnieuw het woordje `is` dat we bij `Equals` hebben gebruikt.

Tweerichtingsverkeer voor `VakInschrijving`

In je huidige code heeft de klasse `Student` een lijst `vakInschrijvingen`. Zo wordt een student gelinkt aan de cursussen die hij of zij volgt. Dit is niet ideaal, want in werkelijkheid willen we ook vaak te weten komen welke studenten in een bepaalde cursus zijn ingeschreven. We moeten dus in twee richtingen kunnen gaan.

Een mogelijke oplossing: voorzie de klasse `VakInschrijving` van een (immutable) lijst `AlleVakInschrijvingen`. Zo hoef je geen data dubbel bij te houden en kan je toch de functionaliteit verder uitbreiden. Schrap de huidige lijst met vakinschrijvingen in de klasse `Student`. Voorzie ter vervanging daarvan een property `student` in de klasse `VakInschrijving` die bijhoudt welke student bij de inschrijving hoort. Voorzie ook, in de klasse `Student`, een property `VakInschrijvingen` die "on-the-fly" berekent welke inschrijvingen bij de student in kwestie horen. Voorzie ook een property `Cursussen`. Voorzie bovendien in de klasse `Cursus` een property `VakInschrijvingen` en een property `Studenten`. Al deze properties zijn onveranderlijke lijsten.

Je zal ten slotte de lijst met studenten uit de klasse `Cursus` moeten verwijderen. Dit vraagt een aantal logische aanpassingen. In de methode `DemonstreerCursussen` mag je code om studenten te associëren met een cursus verwijderen.

Cursussen in semesters

Momenteel bestaat een studieprogramma gewoon uit een vlakke lijst cursussen. Dat stemt niet goed overeen met de werkelijkheid. In werkelijkheid wordt een cursus in een bepaald semester ingepland. Eén manier om dit voor te stellen: vervang de vlakke lijst met cursussen door een `Dictionary` met cursussen als keys en getalwaarden (semesters) als values. Doe deze aanpassing in je code. Je zal hiervoor je demonstratiecode moeten aanpassen. Zorg dat communicatie bij de opleiding programmeren in het eerste semester staat, maar bij de opleiding systeem- en netwerkbeheer in het tweede semester. Alle andere vakken staan overal in het eerste semester.

Manueel data invoeren

De demonstratiemethodes hebben bijna overal objecten aangemaakt door ze te "hard coden". Dat wil zeggen dat de instructies C# code zijn en niet gewijzigd kunnen worden eens je programma gecompileerd is. In een echte systeem voor schoolbeheer zou het administratief personeel voortdurend nieuwe entiteiten kunnen toevoegen aan het systeem.

Voorzie daarom vier nieuwe mogelijkheden in je keuzemenu: "student toevoegen", "cursus toevoegen", "vakinschrijving toevoegen" en "inschrijvingsgegevens tonen". De eerste drie vragen om de nodige gegevens om een object van een van deze klassen aan te maken. De laatste toont eerst alle studenten in het systeem, dan alle cursussen, dan alle inschrijvingen. Zorg ook dat je menu opties presenteert in een oneindige lus, zodat je je methodes samen kan testen.

Onderstaande screenshot toont een voorbeeldinteractie (enkel de nieuwe opties zijn getoond om plaats te sparen):

Wat wil je doen?
7. Student toevoegen
8. Cursus toevoegen
9. VakInschrijving toevoegen
10. Inschrijvingsgegevens tonen
7
Naam van de student?
Tijl Uilenspiegel
Geboortedatum van de student?
01/01/1900
Wat wil je doen?
7. Student toevoegen
8. Cursus toevoegen
9. VakInschrijving toevoegen
10. Inschrijvingsgegevens tonen
8
Titel van de cursus?
Nederlandse literatuur
Aantal studiepunten?
6
Wat wil je doen?
7. Student toevoegen
8. Cursus toevoegen
9. VakInschrijving toevoegen
10. Inschrijvingsgegevens tonen
9
Welke student?
1
Persoon

Naam: Tijl Uilenspiegel
Leeftijd: 121
Meerbepaald, een student
1
Welke cursus?
1 Nederlandse literatuur
1
Wil je een resultaat toekennen?
ja
Wat is het resultaat?
12
Wat wil je doen?
7. Student toevoegen
8. Cursus toevoegen
9. VakInschrijving toevoegen
10. Inschrijvingsgegevens tonen
10
Persoon

Naam: Tijl Uilenspiegel
Leeftijd: 121
Meerbepaald, een student
ingeschreven voor
Nederlandse literatuur
Wat wil je doen?

7. Student toevoegen
8. Cursus toevoegen
9. VakInschrijving toevoegen
10. Inschrijvingsgegevens tonen