1.0.0

H14: Overerving

Overerving intro

> Kennisclip voor deze inhoud

Overerving

Overerving (**inheritance**) laat ons toe om klassen te specialiseren vanuit een reeds bestaande basisklasse. Wanneer we een klasse van een andere klasse overerven dan zeggen we dat deze nieuwe klasse een child-klasse of sub-klasse is van de bestaande parent-klasse of super-klasse.

De child-klasse kan alles wat de parent klasse kan, maar de nieuwe klasse kan nu ook extra specialisatie code krijgen. Deze kan extra methodes voorzien of kan de werking van bestaande methodes aanpassen.

Is-een relatie

Wanneer twee klassen met behulp van een "x is een y"-relatie kunnen beschreven worden dan weet je dat overerving mogelijk.

- Een paard **is een** dier (paard = child-klasse, dier= parent-klasse)
- Een tulp **is een** plant

Inheritance in C#

Overving duid je aan met behulp van het dubbele punt bij de klassedefinitie:

Een voorbeeld:

```
class Paard : Dier
{
   public void Hinnik(){
     Console.WriteLine("Hihihihiii");
   }
}

class Dier
{
   public void Eet()
   {
    //...
   }
}
```

Objecten van het type Dier kunnen enkel de Eet-methode aanroepen. Objecten van het type Paard kunnen de Eet-methode aanroepen én ze hebben ook een methode Hinnik():

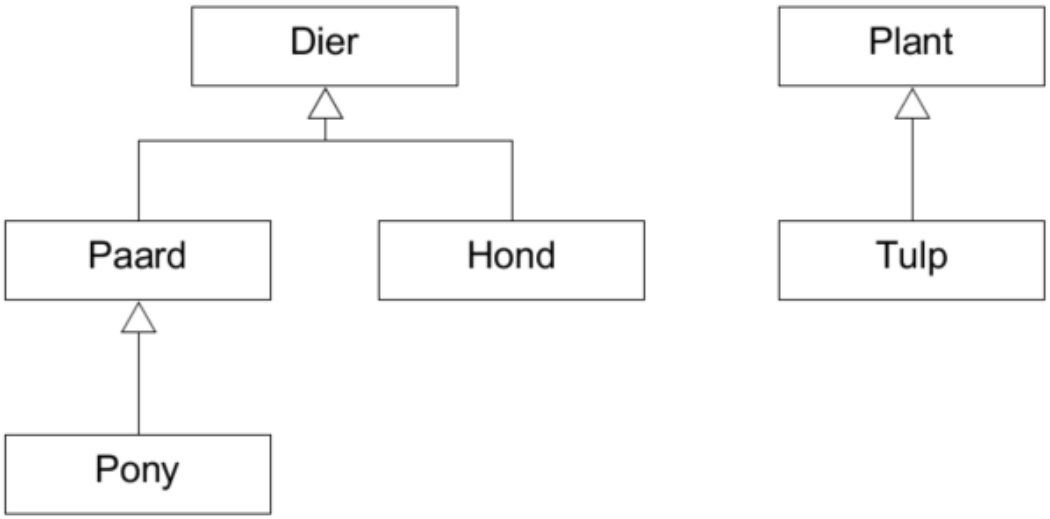
```
Dier aDier = new Dier();
Paard bPaard = new Paard();
aDier.Eet();
bPaard.Eet();
bPaard.Hinnik();
aDier.Hinnik(); // compilatiefout!
```

Transitiviteit

Overerving in C# is transitief, dit wil zeggen dat de child-klasse niet alleen overerft van haar ouderklasse, maar ook van grootouderklassen enzovoort. Je zou bijvoorbeeld een subklasse `Pony` van de klasse `Paard` kunnen toevoegen. Een `Pony` is een `Paard` en een `Paard` is een `Dier`, dus een `Pony` is ook een `Dier` en heeft bijvoorbeeld een methode `Eet`.

UML

In UML-klassediagram zien deze relaties er zo uit:



Voorbeeld UML-klassediagram

De ervende klasse verwijst met een gesloten, maar niet-ingekleurde pijl naar de geërfde klasse. Deze pijlen worden soms bovenop elkaar getekend, zoals bij de relaties tussen Hond, Paard en Dier: zowel Hond als Paard erven van Dier.

Virtual en override

> Kennisclip voor deze inhoud

Virtual en Override

Soms willen we aangeven dat de implementatie (code) van een property of methode in een parent-klasse door child-klassen mag aangepast worden. Dit geven we aan met het **virtual** keyword:

```
class Vliegtuig
{
   public virtual void Vlieg()
   {
      Console.WriteLine("Het vliegtuig vliegt rustig door de wolken.");
   }
}

class Raket: Vliegtuig
{
}
```

Stel dat we 2 objecten aanmaken en laten vliegen:

```
Vliegtuig f1 = new Vliegtuig();
Raket spaceX1 = new Raket();
f1.Vlieg();
spaceX1.Vlieg();
```

De uitvoer zal dan zijn:

```
Het vliegtuig vliegt rustig door de wolken.
Het vliegtuig vliegt rustig door de wolken.
```

Een raket is een vliegtuig, toch vliegt het anders. We willen dus de methode Vlieg anders uitvoeren voor een raket. Daar hebben we **override** voor nodig. Door override voor een methode in de child-klasse te plaatsen zeggen we "gebruik deze implementatie en niet die van de parent klasse." **Je kan enkel overriden indien de respectievelijke methode of property in de parent-klasse als virtual werd aangeduid**

```
class Raket:Vliegtuig
{
   public override void Vlieg()
   {

      Console.WriteLine("De raket verdwijnt in de ruimte.");
   }
}
```

De uitvoer van volgende code zal nu anders zijn:

```
Vliegtuig f1= new Vliegtuig();
Raket spaceX1= new Raket();
f1.Vlieg();
spaceX1.Vlieg();
```

Uitvoer:

```
Het vliegtuig vliegt rustig door de wolken.
De raket verdwijnt in de ruimte.
```

Properties overriden

Ook properties kan je virtual instellen en overriden.

Stel dat je volgende klasse hebt:

```
    class Auto
    {
        virtual public int Fuel { get; set; }
    }
```

We maken nu een meer luxueuze auto die een lichtje heeft dat aangaat wanneer de benzine-tank vol genoeg is en anders uitgaat, dan kan dit via override.

```
class LuxeAuto : Auto
{
   public bool HeeftVolleTank { get; set; }

   public override int Fuel
   {
      get { return base.Fuel; }
      set
      {
            if (value > 100)
            {
               HeeftVolleTank = true;
            }

            else {
               HeeftVolleTank = false;
            }
            base.Fuel = value;
      }
   }
}
```

Abstract

> Kennisclip voor deze inhoud

Abstract

Abstracte klassen

Soms maken we een parent-klasse waar op zich geen instanties van kunnen gemaakt worden: denk aan de parent-klasse `Dier`. Subklassen van Dier kunnen `Paard`, `Wolf`, etc zijn. Van Paard en Wolf is het logisch dat je instanties kan maken (echte paardjes en wolfjes) maar van 'een dier'? Hoe zou dat er uit zien.

Met behulp van het **`abstract`** kunnen we aangeven dat een klasse abstract is: je kan overerven van deze klasse, maar je kan er geen instanties van aanmaken.

We plaatsen `abstract` voor de klasse om dit aan te duiden.

Een voorbeeld:

```
abstract class Dier
{
  public int Name {get;set;}
}
```

Volgende lijn zal een error geven: `Dier hetDier = new Dier();`

We mogen echter wel klassen overerven van deze klasse en instanties van aanmaken:

```
class Paard: Dier
{
//...
}

class Wolf: Dier
{
 //..
}
```

En dan zal dit wel werken: `Wolf wolfje = new Wolf();`

Abstracte methoden

Het is logisch dat we mogelijk ook bepaalde zaken in de abstracte klasse als abstract kunnen aanduiden. Beeld je in dat je een Methode "WetenschappelijkeNaam" hebt in je klasse Dier. Wetenschappelijke namen zijn eigen aan specifieke dieren. We kunnen dus ook geen implementatie (code) geven in de abstracte parent klasse.

Via abstracte methoden geven we dit aan: we hoeven enkel de methode signature te geven, met ervoor `abstract`:

```
abstract class  Dier
{
   public abstract string WetenschappelijkeNaam();
}
```

Merk op dat er geen accolades na de signature komen.

Child-klassen **zijn verplicht deze abstracte methoden te overriden**.

De Paard-klasse wordt dan:

```
class Paard: Dier
{
  public override string WetenschappelijkeNaam()
  {
      return "equus";
  }
}
```

(en idem voor de wolf-klasse uiteraard)

Abstracte methoden enkel in abstracte klassen

Van zodra een klasse een abstracte methode of property heeft dan ben je, logischerwijs, verplicht om de klasse ook abstract te maken.

Abstracte properties

Properties kunnen virtual gemaakt, en dus ook `abstract`. Volgende voorbeeld toont hoe dit werkt:

```
abstract class Dier
{
    abstract public int MaxLeeftijd { get;}
}

class Olifant : Dier
{
    public override int MaxLeeftijd {
        get { return 100; }
    }
}
```

Constructors bij overerving

> Kennisclip voor deze inhoud

Constructors bij overerving

Wanneer je een object instantieert van een child-klasse dan gebeuren er meerdere zaken na elkaar, in volgende volgorde:

- Eerst wordt een constructor aangeroepen van een voorouderklasse opgeroepen.
- Gevolgd door een constructor van de parent klasse.
- Finaal de constructor van de klasse zelf.

Volgende voorbeeld toont dit in actie:

```
class Soldier
{
   public Soldier() {Console.WriteLine("Soldier reporting in");}
}

class Medic:Soldier
{
   public Medic(){Console.WriteLine("Who needs healing?");}
}
```

Indien je vervolgens een object aanmaakt van het type `Medic`:

```
Medic RexGregor= new Medic();
```

Dan zal zien we de volgorde van constructor-aanroep op het scherm:

```
Soldier reporting in
Who needs healing?
```

Er wordt dus verondersteld in dit geval dat er een parameterloze constructor in de basisklasse aanwezig is.

Constructoren met parameters

Indien je klasse Soldier geen parameterloze constructor heeft, dan moeten we uitdrukkelijk zeggen van waar de gebruikte parameters komen. Volgende code zou dus een probleem geven indien je een Medic wilt aanmaken via `new Medic()`:

```
class Soldier
{
   public Soldier(bool canShoot) {//...Do stuff  }
}

class Medic : Soldier
{
   public Medic(){Console.WriteLine("Who needs healing?");}
}
```

Wat je namelijk niet ziet bij child-klassen en hun constructors is dat er eigenlijk een impliciete call naar de basis-constructor wordt gedaan. Bij alle constructors staat eigenlijk `:base()` wat je ook zelf kunt schrijven:

```
class Medic : Soldier
{
    public Medic() : base()
    {
        Console.WriteLine("Who needs healing?");
    }
}
```

`base()` achter de constructor zegt dus eigenlijk "roep de constructor van de parent-klasse aan. Je mag hier echter ook parameters meegeven en de compiler zal dan zoeken naar een constructor in de basis-klasse die deze volgorde van parameters kan accepteren."

We zien hier dus hoe we ervoor moeten zorgen dat we terug Medics via `new Medic()` kunnen aanroepen zonder dat we de constructor(s) van Soldier moeten aanpassen:

```
class Soldier
{
   public Soldier(bool canShoot) {//...Do stuff  }
}

class Medic:Soldier
{
    public Medic() : base(true) {
        Console.WriteLine("Who needs healing?");
    }
}
```

De medics zullen de canShoot dus steeds op true zetten. Uiteraard wil je misschien dit kunnen meegeven bij het aanmaken van een object zoals `new Medic(false)`, dit vereist dat je dus een overloaded constructor in Medic aanmaakt, die op zijn beurt de overloaded constructor van Soldier aanroept. Je schrijft dan een overloaded constructor in Medic bij:

```
class Soldier
{
   public Soldier(bool canShoot) {//...Do stuff  }
}

class Medic:Soldier
{
   public Medic(bool canShoot) : base(canShoot) {}

   public Medic() : base(true) {
       Console.WriteLine("Who needs healing?");
   }
}
```

Uiteraard mag je ook de parameterloze constructor aanroepen vanuit de child-constructor, alle combinaties zijn mogelijk (zolang de constructor in kwestie maar bestaat in de parent-klasse).

Oefeningen

Aparte oefeningen overerving

Deze oefeningen maak je allebei in een klasse genaamd `Overerving`, met haar eigen `ToonSubmenu` methode.

Post

Functionele analyse

We werken een systeem uit voor de post, waarin (aangetekende) brieven geregistreerd worden. Dit zal ons toestaan elke brief op te zoeken in het systeem en interessante informatie over de brief op te vragen.

Technische analyse

- Maak een klasse AangetekendeBrief, met een property `ReisAfstand` van type `double`, die steeds groter dan 0 moet zijn en de afstand in kilometer uitdrukt die de brief zal moeten afleggen.
- Voeg een computed property `Reistijd` (van type byte) en een property `Kostprijs` (van type double) toe.
  - De reistijd is de reisafstand gedeeld door 100, afgerond naar boven, uitgedrukt in dagen.
  - De kostprijs is 15 euro voor brieven die minder dan 100km moeten afleggen. Daarna komt er 10 euro bij per 100km.
- Voorzie voorlopig geen constructor.
- Maak een subklasse `InternationaleAangetekendeBrief`. Voor deze is de reistijd de reisafstand gedeeld door 50, afgerond naar boven. De kostprijs is 20 euro per 100km. Voorzie voorlopig geen constructor.
- Maak een subklasse `HogePrioriteitsAangetekendeBrief`. Voor deze is de reistijd de reisafstand gedeeld door 200, afgerond naar boven. De kostprijs is 30 euro per 100km. Voorzie voorlopig geen constructor.
- Schrijf een methode `DemonstreerBrieven`. In deze methode vraag je de gebruiker, tot hij wenst te stoppen, om brieven in te geven voor verzending. Deze worden toegevoegd aan een lijst van brieven. Wanneer hij klaar is met brieven in te geven, toon je per brief alle eigenschappen.

Voorbeeldinteractie

```
Wil je nog een brief toevoegen? (ja/nee)
> ja
Wat voor brief wil je toevoegen?
1. standaard
2. internationaal
3. hoge prioriteit
4. geen enkele, we zijn klaar met invoeren
> 1
Hoe ver moet deze brief gaan?
> 150
Wat voor brief wil je toevoegen?
1. standaard
2. internationaal
3. hoge prioriteit
4. geen enkele, we zijn klaar met invoeren
> 2
Hoe ver moet deze brief gaan?
> 1000
Wat voor brief wil je toevoegen?
1. standaard
2. internationaal
3. hoge prioriteit
4. geen enkele, we zijn klaar met invoeren
> 3
Hoe ver moet deze brief gaan?
> 800
Wat voor brief wil je toevoegen?
1. standaard
2. internationaal
3. hoge prioriteit
4. geen enkele, we zijn klaar met invoeren
> 4
Brief 1: 150km, reistijd 2 dagen, kostprijs 25 euro
Brief 2: 1000km, reistijd 5 dagen, kostprijs 200 euro
Brief 3: 800km, reistijd 4 dagen, kostprijs 240 euro
```

Dierenarts

Functionele analyse

Een dierenarts moet zijn "patiënten" opvolgen met een geautomatiseerd systeem. Bepaalde aspecten zijn van toepassing voor alle soorten dieren, maar bepaalde zaken zijn afhankelijk van de diersoort.

Technische analyse

- Schrijf een abstracte klasse `Dier`, met een property Naam (van type `string`), `Geslacht` (van een enum type, genaamd `Geslachten`, met waarden `Mannelijk` en `Vrouwelijk`), een abstracte property (van type `ImmutableList<string>`) `Allergieen` en een abstracte methode `ToonChip` (van type `void`)
- Schrijf concrete subklassen `Hond` en `Papegaai`
  - Voor de implementatie van de allergieën van een hond voorzie je eerst een tweede property, `IndividueleAllergieen`. Deze mag een gewone `List<string>` zijn, met default getter en setter. Voor de implementatie van `Allergieen` geef je dan de samenvoeging van de individuele allergieën met "druiven", "noten", "chocolade" en "avocado". Voor `ToonChip` toon je de waarde van een property `Chip` die je op Hond voorziet (met type `string`).
  - Voor papegaaien geef je een lege lijst van allergieën. Voor de chip toon je het bericht "Papegaaien worden niet gechipt."

Voorbeeldinteractie

Voeg volgende methode toe aan je klasse en maak oproepbaar uit het keuzemenu. Test dat alle resultaten kloppen.

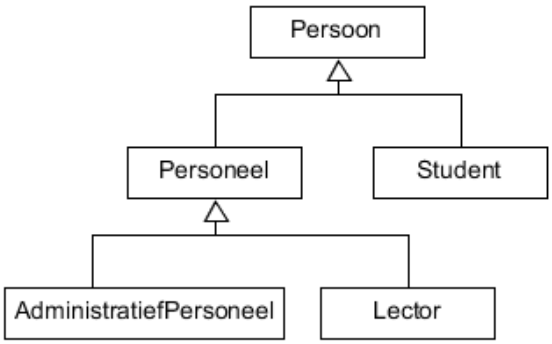
```
public static void DemonstreerDieren() {
    var patientjes = new List<Dier>();
    var dier1 = new Hond();
    dier1.IndividueleAllergieen = new List<string> {"vis"};
    dier1.Chip = "ABC123";
    dier1.Geslacht = Geslachten.Vrouwelijk;
    dier1.Naam = "Misty";
    patientjes.Add(dier1);
    var dier2 = new Papegaai();
    dier2.Geslacht = Geslachten.Mannelijk;
    dier2.Naam = "Coco";
    patientjes.Add(dier2);
    foreach(var dier in patientjes) {
        Console.WriteLine(dier.Naam);
        Console.WriteLine(dier.Geslacht);
        Console.WriteLine("allergieën:");
        foreach(var allergie in dier.Allergieen) {
            Console.WriteLine(allergie);
        }
        dier.ToonChip();
    }
}
```

SchoolAdmin: overerving

Personen overerving

Om naast studenten ook andere personen, zoals lectoren en administratief personeel te kunnen beheren in SchoolAdmin, maken we enkele nieuwe klassen aan:

- Persoon: een abstracte klasse, waarvan de andere klassen zijn afgeleid
- Personeel, met twee kindklassen:
  - AdministratiefPersoneel en Lector
- Student, een klasse die al bestond.



UML-klassediagram Persoon

Persoon

Deze abstracte klassen definieert wat voor alle personen in het systeem hetzelfde is: een id, een naam en een geboortedatum. Voorzie hiervoor dus private attributen met publieke properties. Voorzie voor de properties `Id` en `Geboortedatum` enkel getters, zodat ze read only zijn. Het `Id` wordt automatisch ingesteld bij constructie. Hiervoor wordt een teller `maxId` bijgehouden.

Verder zegt deze klasse ook dat elke klasse die er van erft, ten minste de methoden `GenereerNaamkaartje` en `BepaalWerkbelasting` moet bevatten. Hoe dat naamkaartje er moet uitzien, legt `Persoon` niet vast.

Elke nieuwe persoon die gemaakt wordt, wordt bewaard in een lijst met alle personen. Deze lijst mag door andere klassen niet gewijzigd worden: het beheer van de lijst ligt bij `Persoon`. Deze lijst kan wel opgevraagd worden via een statische property `AllePersonen` die een immutable list produceert.

Student

Deze klasse wordt een kind van `Persoon`. Zorg er dus voor dat deze klasse de verplichte zaken uit Persoon implementeert, maar dat duplicate functionaliteit (bv. Studentnummer, Naam, Geboortedatum, StudentenTeller) vanaf nu aan `Persoon` worden overgelaten: haal deze dus weg uit Student.

Voeg op Student ook een attribuut `dossier` toe. Dit is een collectie waarin opmerkingen genoteerd kunnen worden. De opmerkingen worden geïndexeerd met de datum en het tijdstip waarop ze worden ingegeven. Zorg er voor dat dit dossier niet aangepast kan worden buiten de Student klasse door de getter een immutable dictionary te laten teruggeven.

Intermezzo: controle

Test nu de eerdere methodes `DemonstreerStudenten` en `DemonstreerStudentUitTekstFormaat` opnieuw uit. Alles zou nog moeten werken.

Personeel

Een abstracte klasse die erft van `Persoon`. Op deze klasse wordt een lijst bijgehouden van alle personeelsleden die worden aangemaakt. Ze voorziet ook een ancienniteitsteller: hierin wordt bijgehouden hoeveel jaar het personeelslid al in dienst is. De anciënniteit kan nooit hoger gaan dan 50. Hogere waarden worden verlaagd tot 50.

De klasse eist van al haar kindklassen dat zij een methode `BerekenSalaris` bevatten. Hoe het salaris berekend moet worden, wordt overgelaten aan de kindklassen.

De klasse Personeel voorziet ook een lijst van taken die het personeelslid moet uitvoeren. De taken worden opgeslagen als een naam van een taak, met daarbij het aantal uur per week dat het personeelslid aan die taak zal werken. Deze taken kunnen bij constructie worden meegegeven, maar het oorspronkelijke dictionary mag de taken niet meer kunnen aanpassen. Je moet dus de keys en values kopiëren naar het nieuwe dictionary.

Voorzie ook de mogelijkheid om een immutable list van alle personeel op te vragen.

AdministratiefPersoneel

Deze klasse is een kind van `Personeel` en moet daarom aan alle voorwaarden van `Personeel` én `Persoon` voldoen: er zullen dus enkele methoden verplicht moeten worden geïmplementeerd in deze klasse. Er wordt ook een lijst bijgehouden van alle administratieve personeelsleden die worden aangemaakt.

Het salaris van een administratief personeelslid wordt als volgt berekend: *per 3 jaar, krijgt het personeelslid 75 euro extra bovenop een basisloon van 2000 euro. Dit basisloon wordt vervolgens verrekend met de tewerkstellingsbreuk. De tewerkstellingsbreuk is de werkbelasting van het personeelslid gedeeld door 40 uur (voltijdse tewerkstelling).*

Bijvoorbeeld: Ahmed is 4 jaar in dienst. Hij krijgt dus 2000 EUR basisloon, plus 1 keer 75 EUR ancienniteitstoeslag. Hij werkt echter 30 uur per week in plaats van 40, dus krijgt hij 1556,25 EUR. Cijfers na de komma vallen weg omwille van het datatype.

De werkbelasting van een administratief personeelslid wordt bepaald aan de hand van de taken in zijn of haar takenlijst. De duur van alle taken wordt hiertoe opgeteld.

Het naamkaartje van een administratief personeelslid bevat de naam van het personeelslid, met daarachter de vermelding `(ADMINISTRATIE)`. Bv.

```
Ahmed Azzaoui (ADMINISTRATIE)
```

Er is ook een lijst van alle administratief personeel. Ook hier kan je alleen de immutable versie krijgen buiten de klasse.

Intermezzo: controle

Schrijf nu een methode `DemonstreerAdministratiefPersoneel`. Maak hierin een variabele `ahmed` met de gegevens van bovenstaande persoon. Zijn taken bestaan uit 10u roostering, 10u correspondentie en 10u animatie. Hij is geboren 4 februari 1988.

Doorloop vervolgens met een `foreach` de lijst met alle personeel en toon zo alle naamkaartjes van alle personeel. Herhaal dit ook voor de lijst met administratief personeel. Toon dan ook het salaris en de werkbelasting van Ahmed.

Lector

Deze klasse is een kind van `Personeel` en moet daarom aan alle voorwaarden van `Personeel` én `Persoon` voldoen: er zullen dus enkele methoden verplicht moeten worden geïmplementeerd in deze klasse. Er wordt ook een lijst bijgehouden van alle lectoren die worden aangemaakt. Een `Lector` object bevat dan weer een opsomming van alle Cursussen die deze lector geeft, met voor elke cursus de werkbelasting van deze cursus voor de lector.

Het salaris van een administratief personeelslid wordt als volgt berekend: *per 4 jaar, krijgt het personeelslid 120 euro extra bovenop een basisloon van 2200 euro. Dit basisloon wordt vervolgens verrekend met de tewerkstellingsbreuk. De tewerkstellingsbreuk is de werkbelasting van het personeelslid gedeeld door 40 uur (voltijdse tewerkstelling).*

Bijvoorbeeld: Anna is 9 jaar in dienst. Ze krijgt dus 2200 EUR basisloon, plus 2 keer 120 EUR ancienniteitstoeslag. Ze werkt 10 uur per week in plaats van 40, dus krijgt ze 610,00 EUR.

De werkbelasting van een lector wordt bepaald aan de hand van de cursussen die hij of zij geeft. De werkbelasting van elke cursus in de collectie wordt hiertoe opgeteld.

Het naamkaartje van een lector bevat de naam van de lector, met op een nieuwe lijn `Lector voor:`. Vervolgens worden de titels van alle cursussen die deze lector geeft op telkens een nieuwe lijn toegevoegd. Bv.

```
Anna Bolzano
Lector voor:
Economie
Statistiek
Analystische Meetkunde
```

Intermezzo: controle

Schrijf nu een methode `DemonstreerLectoren`. Maak hierin een variabele `anna` met de gegevens van bovenstaande persoon. Maak hierin ook variabelen voor de drie cursussen in het voorbeeld boven. Anna heeft 3u economie, 3u statistiek en 4u analytische meetkunde. Ze is geboren 12 juni 1975.

Doorloop vervolgens met een `foreach` de lijst met alle personeel en toon zo alle naamkaartjes van alle personeel. Herhaal dit ook voor de lijst met lectoren. Toon dan ook het salaris en de werkbelasting van Anna.

Visual Studio Klassediagram

Om je hierbij te helpen, kan je dit klassediagram bekijken: