

PROGRAMMEREN 2

...:: C#.NET::...

1 INLEIDING OBJECT ORIENTED PROGRAMMING

1.1 INLEIDING

Er is een grens aan het aantal concepten dat een mens tegelijkertijd kan bevatten. Een vaakvermeld aantal is 7 ± 2 . Dit brengt ons ertoe een software systeem als een samenstelling van op zichzelf staande delen te beschouwen (verdeel-en-heers strategie). Zo'n aanpak biedt vele voordelen op vlak van onderhoud, uitbreidbaarheid, testen, etc. in vergelijking met het bouwen van een software systeem als één monoliet blok.

Met het oog op uitbreidbaarheid (aanpassingen doen aan een afgeleverd systeem, in de onderhoudsfase) is het wensbaar een systeem dusdanig op te splitsen dat een toekomstige wijziging in de gebruikersvereisten (hopelijk) geen kettingreactie van wijzigingen doorheen het ganse systeem veroorzaakt en beperkt blijft tot één of enkele delen ervan.

Dergelijke delen van een software systeem noemt men modules. Wat een module precies inhoudt, hangt af de omgeving (het zgn. paradigma) waarin men werkt, bv. Objectgeoriënteerd of functie gericht. Het is belangrijk te onthouden dat men deze doorgaans niet door elkaar gebruikt : ofwel kiest men ervoor een systeem te ontwikkelen als een verzameling functies en data, ofwel als een verzameling objecten.

We beschouwen elke module als een verzameling van diensten (services) die de module aanbiedt aan andere delen van het systeem. Een module is dus steeds een leverancier van diensten : een supplier. Een module A die gebruikt maakt van de diensten van een module B, noemen we een klant (client) van module B. De meeste modules zijn zowel supplier als client.

Een ander voordeel van modulair werken is dat een module afzonderlijk ontwikkeld en getest kan worden.

Omdat een systeem uit vele modules zal bestaan worden deze gegroepeerd in een subsysteem om het geheel overzichtelijk te houden. Bijvoorbeeld : het grafisch subsysteem zal modules bevatten die met de weergave op het scherm te maken hebben, het opslag subsysteem bevat modules die voor de persistentie van de gegevens zorgen, enz...

Een belangrijke verzameling modules is het domein subsysteem, dit zijn modules die concepten uit het probleemdomein beschrijven.

Er zijn natuurlijk onnoemelijk veel manieren waarop een systeem in modules kan verdeeld worden. Een belangrijke doelstelling van de ontwerp activiteiten is het afwegen van de vooren nadelen van een bepaalde opsplitsing tegenover alternatieve opsplitsingen.

Ongeacht de context waarin men werkt (hier : object-georiënteerd), kunnen we enkele wenselijke interne kwaliteiten aan modules toekennen. Dit zijn slechts enkele voorbeelden van interne kwaliteitsfactoren.

1.1.1 Cohesie

(Functionele) cohesie duidt de mate aan waarin de services van een module (functioneel) gerelateerd zijn.

Cohesie is een kenmerk van een individuele module. We zullen ernaar streven om de cohesie in een module zo hoog mogelijk te houden, waarbij we vooral oog hebben voor functioneel gerelateerde services, i.t.t. tot bijvoorbeeld temporeel gerelateerde services.

- Voorbeeld

Een module die het concept van een student voorstelt maar ook services bevat om de huidige tijd op te vragen en een datum om te zetten naar allerlei tekst voorstellingen, bezit een lage

functionele cohesie. Indien we deze module herwerken tot één module die studenten beschrijft en één module voor datum/tijd gerelateerde zaken, bekomen we twee modules die elk een hogere functionele cohesie vertonen

- Voorbeeld

Een module die allerhande initialisatie services biedt (databank verbinding opzetten, Uiforms opbouwen, configuratie gegevens inlezen, ...) bezit een hoge temporele cohesie omdat de services min of meer gelijktijdig gebruikt worden bij het opstarten van het systeem.

De functionele cohesie van zo'n module is echter laag. Het is beter deze initialisatie services onder te brengen in modules die de relevante concepten beschrijven (m.a.w. modules in het databanksysteem, UI-systeem).

1.1.2 Koppeling

Koppeling duidt de mate aan waarin twee modules afhankelijk zijn van elkaar.

Koppeling is een kenmerk van een verzameling modules. Een module is afhankelijk van een andere module indien ze haar services gebruikt, anders gezegd, een client is afhankelijk van z'n supplier(s). Deze afhankelijkheid is groter naarmate de client meer over de supplier moet weten om de services te kunnen gebruiken. We zullen ernaar streven om de koppeling tussen modules zo laag mogelijk te houden door het aantal afhankelijkheden in een systeem (en ook de mate van afhankelijkheid!) te minimaliseren.

Dit betekent niet dat we zomaar afhankelijkheden kunnen vermijden, sommige afhankelijkheden zijn nu eenmaal inherent aan de concepten die door de modules beschreven worden (zo kan bv. een module die studenten beschrijft afhankelijk zijn van een module die adressen beschrijft, omdat studenten nu eenmaal een adres hebben).

Hoe groter de koppeling tussen twee modules, hoe groter de kans dat een wijziging in de supplier zal leiden tot een wijziging in de client. Het is dus belangrijk dat de koppelingen tussen de modules in een systeem expliciet zijn. Zoniet (m.a.w. als we de clients van een gewijzigde module niet kennen), moeten we in het ergste geval elke module in het systeem controleren.

- Voorbeeld

Indien we het domein subsysteem afhankelijk maken van het UI-systeem (doordat enkele domein klassen bv. keuzelijsten in de UI invullen), moeten we deze domein modules wijzigen om een andere UI (bv. andere forms of een webinterface) te kunnen gebruiken. We kunnen deze modules ook niet hergebruiken zonder de bijbehorende UI-klassen.

Het is aanbevolen om het domein subsysteem onafhankelijk te houden van het Uisysteem (bv. door de UI zelf de nodige gegevens uit het domein te laten ophalen).

1.1.3 Abstractie

Abstractie is de mate waarin de interface van een module enkel die zaken van het beschreven concept kenbaar maakt, die relevant zijn voor de client.

We streven ernaar modules te bouwen die precies één concept (compleet) voorstellen (zie ook Cohesie) en dit op een voldoende hoog niveau van abstractie. Dit betekent dat we goed gedefinieerde en uitgebalanceerde interfaces proberen te voorzien (niet te veel en niet te weinig services) en alle implementatie-specifieke zaken voor de client verborgen houden (wegens niet relevant en om ongewenste afhankelijkheden te vermijden).

Abstractie zorgt ervoor dat een module de client enkel dat biedt wat van belang is, terwijl inkapseling ervoor zorgt dat de client geen toegang heeft tot de implementatie. Het resultaat van inkapseling en abstractie samen, noemt men vaak information hiding.

1.1.4 Herbruikbaarheid

Elke module heeft een bepaalde context nodig om in te werken. Hierbij moet vooral gedacht worden aan de interfaces van zijn suppliers. Een module garandeert de implementatie van zijn eigen interface, op voorwaarde dat hij in de juiste context gebruikt wordt.

Modules met een lage koppeling, hoge cohesie, die een ingekapselde abstractie voorstellen met een goed gedefinieerde interface kunnen mogelijkherwijs hergebruikt worden, op voorwaarde dat aan de context-afhankelijkheden is voldaan.

1.2 KLASSEN EN INSTANTIES (OBJECTEN)

De basis van "Object - Oriented Programming" (OOP) zijn objecten.

In een OO- model wordt bij uitvoering alle functionaliteit gerealiseerd door samenwerkende objecten die met elkaar communiceren door boodschappen (messages) te sturen.

Objecten zijn constructies waarin een toestand en een bepaald gedrag wordt gecombineerd. Of anders gezegd: een object bevat bepaalde informatie en vertoont een bepaald gedrag.

Objecten kunnen voorstellingen zijn van zowel materiele/concrete zaken (bijvoorbeeld personen of auto's) als onmateriële/abstracte zaken (bijvoorbeeld afbetalingen, reservaties, verhuringen, ...).

Alle objecten behoren tot een bepaalde klasse (bij "single classification"). Of anders gezegd een object is een instantie van een klasse.

Een klasse kan je bekijken als een sjabloon, een template, een moule, een bouwplan, ... voor objecten. Een object vormt dan een afgietsel, een verwezenlijking, ... van die klasse.

Klassen beschrijven objecten door onder andere te bepalen wat voor informatie deze objecten kunnen bevatten en door te bepalen wat het gedrag is van deze objecten.

In een klasse wordt dus gedefinieerd hoe een bepaald voorstelling wordt geabstraheerd.

Wanneer een probleemdomein wordt geanalyseerd of een oplossing voor wordt ontworpen, kan men vaak dit probleem of deze oplossing het eenvoudigst formuleren door de verschillende concepten te gaan omschrijven. In deze omschrijving gaat men vermelden wat de kenmerken/eigenschappen van deze concepten zijn, en wat er met deze concepten moet kunnen gebeuren.

In een facturatieprogramma zou men kunnen spreken over de concepten artikel, factuurregel en factuur. Het concept artikel heeft als eigenschap ondermeer een bepaalde prijs. Het concept factuurregel heeft als eigenschap een bepaald artikel en een bepaalde hoeveelheid (van dit artikel). Het concept factuur heeft dan als eigenschap een verzameling van factuurregels.

Van een factuurregel moet de totale prijs (eenheidsprijs van het artikel x hoeveelheid) worden bepaald. Van een factuur moet het totaal (van alle factuurregels worden bepaald, en dit ook inclusief tax.

Door gebruikt te maken van het hogere abstractieniveau in het object georiënteerd programmeren kan men op eenvoudige wijze een oplossing formuleren voor dit probleemdomein. Ten slotte dient een klasse net om een concept met eigenschappen en mogelijke acties op deze eigenschappen te gaan definiëren.

Een objectgeoriënteerde aanpak zou als voordeel moeten hebben dat het programma/de programmacode beter herbruikbaar, beter aanpasbaar, beter uitbreidbaar en beter gestructureerd is.

1.3 EEN EERSTE KLASSE

Hieronder zien we een eerste klasse-definitie, het betreft een eenvoudige abstractie voor personen, die over een naameigenschap beschikken.

- Code

```
class Person
{
    private string _Name;           // 1

    public string GetName()         // 2
    {
        return _Name;
    }

    public void SetName(string name) // 3
    {
        _Name = name;
    }
}
```

Naast class vermeld je de naam (identifier) van de klasse, de naam van de klasse zal doorgaans een zelfstandig naamwoord zijn, en dient zo duidelijk mogelijk te omschrijven wat objecten van deze klasse zullen voorstellen.

1.4 MEMBERS

In deze klasse zijn 3 (klasse-)members gedefinieerd:

- (1) een ingekapseld (private) veld om hierin de naam van de persoon in te stockeren
- (2) een getter-method die de client toelaat (publiek (public) toegankelijk) de naam van een persoon op te vragen
- (3) een setter-method die de client toelaat (publiek (public) toegankelijk) de naam van een persoon in te stellen

1.5 ACCESS MODIFIERS

Wat hier ook gedemonstreerd wordt zijn de access-modifiers 'private' en 'public'. Deze staan steeds vooraan de declaratie of definitie van de klasse-member, en geven aan of de member voor de 'Client' (de gebruiker van de klasse (of van objecten van de klasse) 'Person') toegankelijk is. Het veld wordt hier ingekapseld (encapsulation) en is dus enkel maar benaderbaar binnen de klasse waarin die member is gedefinieerd. Typisch is om in de abstractie van de klasse zo'n privaat veld te combineren met een publiek toegankelijke getter en setter. Zo heeft de client toch de mogelijkheid de naam op te vragen of in te stellen voor een bepaalde persoon (bepaald 'Person'-object). En is het dus niet meer noodzakelijk het veld (waarin de naam echt gestockeerd zit) publiek toegankelijk te maken, dus kan deze ingekapseld worden.

1.5.1 Encapsulatie

Een belangrijk element bij het definiëren van een klasse is inkapseling, of liever: afscherming. Het idee is dat een programma dat een klasse gebruikt niets hoeft te weten over de manier waarop de klasse haar werk doet. Het programma creëert simpelweg een instantie van een klasse en roept vervolgens haar methoden aan. Zolang die methoden doen wat ze beloven, maakt het niet uit hoe ze dat doen. Wanneer u

bijvoorbeeld de Show-methode van de MessageBox-klasse aanspreekt hoeft u niet te weten hoe deze klasse het precies voor elkaar krijgt om een dialoogvenster naar het scherm te sturen.

Het kan ook zijn dat een klasse allerlei interne variabelen gebruikt om de methoden hun werk te laten doen. Het is verstandig om de variabelen niet zichtbaar te maken

als dat voor de werking niet nodig is of juist schadelijk kan zijn. Vandaar ook de benaming afscherming in plaats van inkapseling: het gaat om het verbergen van informatie. Afscherming dient eigenlijk twee doelen.

Het afbakenen van de methoden en eigenschappen van een klasse zodat een betekenisvolle klassificatie mogelijk is.

Het beperken van de toegang van de methoden en eigenschappen om gebruik of misbruik in te perken.

1.6 NAMING GUIDELINES

De volgende richtlijnen voor identifiers van klassen en members worden hier gevolgd:

- een prefix _ voor velden
- "PascalCasing" voor identifier van klassen en publieke members
- "camelCasing" voor ingekapselde members

"PascalCasing" houdt in dat je elk woord in een samenstelling met een hoofdletter doet beginnen, bijvoorbeeld GrootsteGetal.

"camelCasing" is zoals "PascalCasing" maar de eerste letter van de identifier is een kleine letter in plaats van een hoofdletter.

1.7 INSTANTIEREN VAN EEN KLASSE

- Code

```
class Program
{
    static void Main(string[] args)
    {
        Person person1 = new Person();           // 4
        person1.SetName("John");                  // 5
        Console.WriteLine(person1.GetName());     // 6
        //
        Person person2 = new Person();
        person2.SetName("Jane");
        Console.WriteLine(person2.GetName());
        //
        Console.ReadLine();
    }
}
```

- Output

```
John
Jane
```

In bovenstaande Client ziet u het gebruik van de Person-klasse.

Allereerst wordt er een object geconstrueerd en de verwijzing naar dit object gestockeerd in een variabele (4). Daarna wordt de naameigenschap voor dit object ingesteld (5), waarna de naameigenschap van dit object wordt opgevraagd (6).

Alle publieke members van een object (gedefinieerd in de klasse die zo'n objecten beschrijven) zijn bereikbaar via de puntnotatie ('dot-notation').

Na de naam/identificer van de object-variabele plaatst men een punt gevolgd door de identificer van de publieke member.

Belangrijk om te vermelden is dat alle members hier objectgerelateerd zijn. Hiermee wordt bedoeld dat als je bijvoorbeeld de `SetName`-method aanroept voor object `person1` (`person1.SetName(...)`), dat ook weldegelijk de naam voor dat object (waar naartoe je verwijst) wordt ingesteld, en er dus geen naam-eigenschappen voor andere objecten zullen worden aangepast.

Men spreekt ook wel over instance-members als de members objectgerelateerd zijn. Een object is tenslotte een bepaalde instantie van een klasse.

Het `New Person` gedeelte is een object-initializer. Dit bestaat steeds minimaal uit het keyword `New` en een type-specifier (klasse-identificer). Deze object-initializer zal een object construeren van dit klasse-type. Dat construeren houdt in dat in het geheugen plaats wordt gereserveerd om de toestand (verzameling van alle veld-waardes) van dit object te bewaren. Deze object-initializer is ook een expressie, een expressie die resulteert in de referentie (een verwijzing) naar het object. Het is deze referentie die vervolgens wordt toegekend aan een variabele. Deze variabele is gedeclareerd, en werd dus in het geheugen aangemaakt, om te kunnen verwijzen naar een object van het geinitialiseerde type.

Het aanmaken van een klasse is eigenlijk het definiëren van een type, om meer specifiek te zijn een reference-type. Andere (data)types als `String` werden bijvoorbeeld ook voorgedefinieerd door een bepaalde klasse.

Het gaat hier om een reference-type omdat men naar objecten (instanties) van dit type kan verwijzen aan de hand van een referentie.

Een objectvariabele (een variabele gedeclareerd van een reference-type) bevat tenslotte niet het object zelf, maar bevat de referentie van dit object.

1.8 NULL EN NULLREFERENCEEXCEPTION

Indien onderstaand voorbeeld wordt uitgevoerd, krijgen we een `Exception` (Use of unassigned local variable 'person1').

- Code

```
Person person1;  
person1.SetName("John");
```

De variabele `person1` bevat `Null` omdat nergens een referentie van een object aan deze variabele werd toegekend.

Het aanroepen van een objectgerelateerde member op deze variabele veroorzaakt dan ook een fout want er is niet geweten op welk object men de implementatie van deze member moet uitvoeren.

2 PROPERTIES

2.1 GETTER (GET) EN SETTER (SET)

Specifiek voor een aantal .NET-programmeertalen zijn Property-constructies. Deze vervangen de get-method en/of de set-method (en worden ook zo op de achtergrond vertaald):

- Code

```
using System;

class Person
{
    private string _Name;

    public string Name
    {
        get
        {
            return _Name;
        }
        set
        {
            _Name = value;
        }
    }
}

class Program
{
    static void Main(string[] args)
    {
        Person person1 = new Person();
        person1.Name = "John";
        Console.WriteLine(person1.Name);
        //
        Person person2 = new Person();
        person2.Name = "Jane";
        Console.WriteLine(person2.Name);
        //
        Console.ReadLine();
    }
}
```

- Output

```
John
```

De getter van de property wordt uitgevoerd wanneer de property wordt uitgelezen. De setter wordt uitgevoerd bij het instellen van de property-waarde. De toegekende waarde (hier de String-literal "John") wordt geassigneerd aan de argumentvariable value van de setter.

Zoals je merkt is de implementatie van de getter en setter identiek aan respectievelijk de get- en set-method uit vorig voorbeeld.

Anders is wel het gebruik voor de client. Deze client kan de property nu aanspreken alsof het een publiek veld zou betreffen van dat object.

De toekenning aan deze property gebeurt op dezelfde wijze als een toekenning aan een String-variabele.

Het uitlezen van de property vormt hier een String-expressie, net zoals de aanroep naar de get-method uit vorig voorbeeld. Dit maakt ook duidelijk dat het hier een String-expressie zal zijn die als return-waarde wordt opgegeven in de getter van de property.

De argumentvariable van de setter krijgt by default de identifier value, dit kan aangepast worden, maar hier is echter geen reden toe.

Het voordeel van het werken met een Property-constructie is echter dat er nu binnen de implementatie van de getter en de setter ruimte is voor validatie.

De getter kan bijvoorbeeld de inhoud van het veld (gebruikt voor de opslag van de eigenschap) valideren (en/of bewerken) alvorens de terugkeerwaarde in te stellen. De setter kan dan bijvoorbeeld de toegekende waarde (inhoud van value) valideren (en/of bewerken) alvorens het gerelateerde veld op te vullen.

Validatie is natuurlijk geen specifiek kenmerk voor de property-constructie, ook in get- en set-methods kan gelijkaardige validatie gebeuren.

Velden worden zelden publiek gemaakt. Deze zijn immers doorgaans louter noodzakelijk om een toestand te bewaren. Het bewaren van een toestand is een implementatiekwestie en implementatiekwesties worden steeds ingekapseld.

Tot en met versie 2005 waren er weinig significante voordelen van het gebruik van properties in tegenstelling tot het gebruik van get- en set-methods. De keuze voor een van beide was dan ook vooral afhankelijk van de voorkeur van de auteur van de klasse.

Sedert versie 2008 biedt het gebruik van properties in tegenstelling tot het gebruik van set- en get-methods meerdere voordelen.

Bemerkt dat de identifier van Properties doorgaans zelfstandig-naamwoorden (bijvoorbeeld Name) zullen zijn, in tegenstelling tot methods die doorgaans een werkwoord of een combinatie van een werkwoord en een zelfstandig-naamwoord (bijvoorbeeld GetName, SetName, ...) zullen zijn.

2.1.1 Access Modifiers voor Getters en Setters

De access modifier van enkel de getter of enkel setter van een property kan beperkt worden (bijvoorbeeld Private in plaats van Public) ten aanzien van de property zelf.

Hieronder beschikt een Counter-klasse over een publieke Value-property met een ingekapselde setter (1):

- Code

```
using System;

class Counter
{
    private int _Value;

    public int Value
    {
        get
        {
            return _Value;
        }
        private set // 1
        {
            _Value = value;
        }
    }

    public void Raise()
    {
        Value += 1; // 2
    }
}

class Program
{
    static void Main(string[] args)
    {
        Counter counter1 = new Counter();
        Console.WriteLine(counter1.Value);
        counter1.Raise();
        counter1.Raise();
        counter1.Raise();
    }
}
```

```

        Console.WriteLine(counter1.Value);
        //
        counter1.Value = 3;           // 3
        //
        Console.ReadLine();
    }
}

```

Binnen de klasse zelf kan Value zowel uitgelezen als ingelezen worden (2). Alle members van een klasse (of ze nu ingekapseld zijn of niet) zijn immers beschikbaar binnen de klasse zelf.

Zoals hieronder gedemonstreerd, is het voor clients niet mogelijk om de setter van de property Value te gebruiken (3).

Men bekommt hier voor client als het ware een "read-only" eigenschap. Binnen de klasse zelf is de property echter wel "writable".

Abstractiegewijs komt het zelden voor dat getters worden ingekapseld wanneer de setter publiek toegankelijk is.

2.1.2 ReadOnly Property

Regelmatig zal het voorkomen dat property enkel uitleesbaar moet zijn, en dus niet instelbaar. Typische voorbeelden hiervoor zijn berekende en afleidbare eigenschappen. Als men bijvoorbeeld voor een Product de Prize en het TaxPercentage kent, kan men hieruit ook de PrizeIncludingTax bepalen.

Zo'n abstracties kunnen we bekomen door gebruik te maken van een 'ReadOnly Property', deze heeft geen setter, maar bepaald enkel in zijn getter welke waarde wordt opgeleverd.

Hieronder een voorbeeld:

- Code

```

using System;

namespace Example
{
    class Product
    {
        private decimal _Price;

        public decimal Price
        {
            get{ return _Price;}
            set{ _Price = value;}
        }

        private decimal _TaxPercentage;

        public decimal TaxPercentage
        {
            get{ return _TaxPercentage;}
            set{ _TaxPercentage = value;}
        }

        public decimal PriceIncludingTax
        {
            get{ return Price * (1 + (TaxPercentage / 100));}
        }
    }

    class Program
    {
        static void Main(string[] args)
        {
            Product product1 = new Product();
            product1.Price = 100;
            product1.TaxPercentage = 8;
        }
    }
}

```

```

        product1.PriceIncludingTax = 108;    // impossible (ReadOnly Property)
        //
        Console.WriteLine(product1.PriceIncludingTax);
        //
        Console.ReadLine();
    }
}

```

- Output

```
108,00
```

In bovenstaande Client kan dus geen waarde worden toegekend aan de property PriceIncludingTax.

Men zou er ook voor kunnen opteren alle 3 de eigenschappen writable te maken, maar dan bekom je een vreemde situatie. Als bijvoorbeeld de PriceIncludingTax wordt ingesteld moet je dan op basis hiervan en op basis van de Tax de Price afleiden, of moet je op basis hiervan en op basis van de Price de Tax afleiden. Je ziet het wordt verwarrend.

Eender welke keuze je uiteindelijk ook zou maken, steeds zal je ervoor moeten zorgen dat de client-auteur hiervan op de hoogte kan zijn (door dit aspect bijvoorbeeld in de klasse-documentatie te verduidelijken). Maar hoe dan ook zo'n verwarrende situatie is beter te vermijden.

Het opstellen van een goede abstractie is continue zoeken naar een delicaat evenwicht, net voldoende mogelijkheden bieden aan de client van die klasse zonder verwarring te creëren is de kunst.

Merk ook op dat in bovenstaand voorbeeld in de implementatie van de property PriceIncludingTax er gebruik wordt gemaakt van de (publiek toegankelijke) properties Price en TaxPercentage, en dus niet van de - in de klasse zelf ook toegankelijke - velden _Price en m_TaxPercentage.

Dit is doorgaans de beste keuze omdat je zeker kunt stellen dat de publieke members (die je dus ook aan de clients aanbiedt) steeds een voorstelling zal blijven van Price en TaxPercentage (eventueel inclusief hun validaties of bewerkingen). Dit zal steeds zo blijven omdat de betekenis van die members gegarandeerd zou moeten zijn, dit kan je niet zeggen van de gerelateerde velden die gebruikt worden voor de interne opslag, deze kunnen eender welke voorstelling zijn, die nog gevalideerd of gemanipuleerd kunnen worden alvorens tot hetgeen te komen dat echt de prijs of de tax voorstelt.

Zelfs al gaat het niet om een publieke property dan zal het om dezelfde redenen doorgaans nog altijd interessanter zijn deze te gebruiken in plaats van een eventueel gerelateerd veld.

In dit voorbeeld merk je ook dat een property-constructie niet noodzakelijk moet gekoppeld worden aan een veld. Binnen de implementatie van de getter en de setter kan men eender welke code uitvoeren.

Hoewel deze PriceIncludingTax te verwezenlijken was via een (readonly) Property kunnen we hiervoor ook een method gebruiken. Dit is ook wat de coding guidelines ons aanraden in het geval dat de implementatie meer inhoud dan enkel het ophalen van een veldwaarde:

- Code

```

using System;

namespace Example
{
    class Product
    {
        private decimal _Price;

        public decimal Price
        {
            get{ return _Price;}
            set{ _Price = value;}
        }
    }
}

```

```

        private decimal _TaxPercentage;

        public decimal TaxPercentage
        {
            get{ return _TaxPercentage;}
            set{ _TaxPercentage = value;}
        }

        public decimal PriceIncludingTax()
        {
            return Price * (1 + (TaxPercentage / 100));
        }
    }

    class Program
    {
        static void Main(string[] args)
        {
            Product product1 = new Product();
            product1.Price = 100;
            product1.TaxPercentage = 8;
            //
            Console.WriteLine(product1.PriceIncludingTax());
            //
            Console.ReadLine();
        }
    }
}

```

- Output

```
108,00
```

2.1.3 ReadOnly Fields

Ook velden kunnen ReadOnly gedeclareerd worden.

Aan deze kan enkel een waarde worden toegewezen in de constructor.

Dit wordt gebruikt voor bevroren toestanden, niet te verwarren met constanten. Een bevroren toestand (in tegenstelling tot een constante) kan ingesteld worden at runtime (enkel in de constructor), maar zal daarna nooit meer van waarde veranderen (net als een constante).

Voor meer details over constructoren en ReadOnly velden, kan je het hoofdstuk over constructoren raadplegen.

2.1.4 WriteOnly Property

Naast ReadOnly Property kan men ook een WriteOnly Property opstellen, deze heeft dan geen getter. Maar abstractiegewijs zijn er bijna geen situaties waarin dit bruikbaar is. Je zou je kunnen afvragen voor welke eigenschap het nuttig kan zijn het toe te laten een bepaalde waarde in te stellen, maar niet toe te laten deze waarde (die toch reeds gekend is omdat de client deze heeft ingesteld) op te vragen.

Een WriteOnly Property is dan ook zeldzaam en getuigt meestal van een povere abstractie, ik verkies dan ook hiervan geen voorbeeld op te nemen.

2.2 AUTO-IMPLEMENTED PROPERTIES

Vele Property constructies worden op dezelfde manier opgebouwd.

De Property wordt aan een Private veld gekoppeld, die zorgt voor het opslaan van de waarde waar deze Property toegang tot verleent. De getter Get levert vaak gewoon de inhoud van dit veld op en de setter Set kent vaak gewoon de aan de Property toegekende waarde toe aan het ingekapselde veld.

- Code

```
class Product
{
    private decimal _Price;

    public decimal Price
    {
        get { return _Price; }
        set { _Price = value; }
    }

    private decimal _TaxPercentage;

    public decimal TaxPercentage
    {
        get { return _TaxPercentage; }
        set { _TaxPercentage = value; }
    }

    public decimal PriceIncludingTax()
    {
        return Price * (1 + (TaxPercentage / 100));
    }
}
```

In C# 3.0 en hoger kunnen de property-declaraties bondiger gemaakt worden via auto-geïmplementeerde eigenschappen. Er geen extra logica nodig is voor de accessors. Bovendien wordt er client-code gecreëerd voor objecten. De compiler creëert een eigen, anoniem veld dat alleen kan worden geraadpleegd via de set- en get-accessors.

- Code

```
class Product
{
    public decimal Price {get; set;}
    public decimal TaxPercentage { get; set; }
    public decimal PriceIncludingTax()
    {
        return Price * (1 + (TaxPercentage / 100));
    }
}
```

Hierdoor hoeft men voor dergelijke basisconstructie, waar de getter enkel de waarde van het gekoppelde veld oplevert, en de setter enkel de toegekende waarde aan het veld toekent, enkel nog een soort van signatuurregel te definiëren.

3 IMPLEMENTATIE

3.1 CACHING VERSUS CALCULATION

Een alternatief voor het berekenen van de PriceIncludingTax in de getter van de desbetreffende property is het berekenen ervan in de setters van Prize en Tax properties. Deze berekende waarde kan dan "gecached" worden, zodoende het raadplegen van de property PriceIncludingTax te versnellen door dan simpelweg de gestockeerde waarde terug te geven.

- Code

```
using System;

namespace Example
{
    class Product
    {
        private decimal _Price;
        public decimal Price
        {
            get { return _Price; }
            set
            {
                _Price = value;
                setPriceIncludingTax();
            }
        }

        private decimal _TaxPercentage;
        public decimal TaxPercentage
        {
            get { return _TaxPercentage; }
            set
            {
                _TaxPercentage = value;
                setPriceIncludingTax();
            }
        }

        private decimal _PriceIncludingTax;
        private void setPriceIncludingTax()
        {
            _PriceIncludingTax = Price * (1 + (TaxPercentage / 100));
        }

        public decimal GetPriceIncludingTax()
        {
            return _PriceIncludingTax;
        }
    }

    class Program
    {
        static void Main(string[] args)
        {
            Product product1 = new Product();
            product1.Price = 100;
            product1.TaxPercentage = 8;
            // Console.WriteLine(product1.GetPriceIncludingTax());
            // Console.ReadLine();
        }
    }
}
```

- Output

Dit kan voordelig zijn wanneer de frequentie van opvragen van de `PriceIncludingTax` (laten uitvoeren van de getter van die property) hoger zal zijn dan de frequentie van instellen van de `Price` of `Tax` (laten uitvoeren van de setters van die properties). De berekening zal zo minder moeten uitgevoerd worden, wat de performantie bevordert.

Hierbij gaat men er natuurlijk wel vanuit te weten dat clients de klasse vooral op deze wijze zouden gebruiken, dit is echter vaak moeilijk te achterhalen of voorspelen. Bijkomend valt ervoor te pleiten de kans tot herbruik van de klasse zo hoog mogelijk te houden (dit zou tenslotte 1 van de grote voordelen van OOP moeten zijn) en er dus rekening mee te houden dat de klasse op allerlei manieren gebruikt kan worden.

In plaats van de formule hier in beide setters te herhalen, wordt hier gebruik gemaakt van een private hulp-method om die formule in te plaatsen, wat als voordeel heeft dat als de formule zou wijzigen men dit op 1 centraal punt zou kunnen aanpassen. De hulp-method wordt ingekapseld (Private) omdat clients deze bewerking niet moeten kunnen initiëren.

Wat nadelig is aan deze werkwijze is dat de logica omtrent het bepalen van de `PriceIncludingTax` (en omtrent het wanneer bepalen) hier verspreidt zit over verschillende members van de klasse, en dus moeilijker te onderhouden zal zijn. Een grotere afhankelijkheid tussen de members (of zelfs tussen verschillende klassen, als de afgeleide waarde gebaseerd is op externe objecten) kan zo worden gecreëerd, en vaak zal extra synchronisatiecode moeten toegevoegd worden.

Ga dan ook doordacht te werk, en wees je bewust van eventuele consequenties wanneer je voor deze aanpak kiest. Waarschijnlijk zal de aanpak uit vorig voorbeeld dus de te preferen aanpak zijn.

In het geval je toch kiest om de berekende waarde als toestand te stockeren, raden coding guidelines aan een Property te gebruiken. De waarde moet enkel uitleesbaar zijn, dus een `ReadOnly Property`:

- Code

```
using System;

namespace Example
{
    class Product
    {
        private decimal _Price;
        public decimal Price
        {
            get { return _Price; }
            set
            {
                _Price = value;
                setPriceIncludingTax();
            }
        }

        private decimal _TaxPercentage;
        public decimal TaxPercentage
        {
            get { return _TaxPercentage; }
            set
            {
                _TaxPercentage = value;
                setPriceIncludingTax();
            }
        }

        private decimal _PriceIncludingTax;
        private void setPriceIncludingTax()
        {
            _PriceIncludingTax = Price * (1 + (TaxPercentage / 100));
        }
    }
}
```

```

        public decimal PriceIncludingTax
        {
            get {return _PriceIncludingTax;}
        }
    }

    class Program
    {
        static void Main(string[] args)
        {
            Product product1 = new Product();
            product1.Price = 100;
            product1.TaxPercentage = 8;
            //
            Console.WriteLine(product1.PriceIncludingTax);
            //
            Console.ReadLine();
        }
    }
}

```

- Output

```
108,00
```

3.2 UNIT TESTEN

Voor elke van de onderstaande klassen is steeds een "testfixture" gecreëerd.

De members van elke klasse (elke "unit") worden getest op het verwachte gedrag.

Deze fixture maakt steeds een instantie aan van de te testen klasse, en zal daar aan de hand van de queries (functies) de initiële toestand van de eigenschappen en het aanpassen van de toestanden door de commands (procedures) gaan testen.

Deze testen definiëren wat we als gedrag verwachten van deze klasse.

Het kan geen kwaad - bij wijze van "testdriven development" - deze testen te schrijven alvorens de klasse zelf te definiëren.

Zo denk je na over de vereiste abstracte, de publieke interface en het verwachte gedrag van objecten van deze klasse.

Eens men over deze testen beschikt kan men makkelijk aan de implementaties van de members van deze klasse werken, en dit tot alle testen slagen.

Indien alle testen slagen zal bij deze fixtures steeds True als output op de console komen.

- Code

```

class Program
{
    static void Main(string[] args)
    {
        Product product1 = new Product();
        //
        product1.Price = 100;
        Console.WriteLine(product1.Price == 100);
        //
        product1.TaxPercentage = 8;
        Console.WriteLine(product1.TaxPercentage == 8);
        //
        Console.WriteLine(product1.PriceIncludingTax == 108);
        //
        Console.ReadLine();
    }
}

```


- Output

```
True
True
True
```

3.3 PUBLIEKE MEMBERS BENADEREN

Indien men in een bepaalde implementatie zowel een ingekapseld veld als een publieke member kan gebruiken om een bepaalde waarde van een object aan te spreken, valt het doorgaans aan te raden deze waarde via de publieke member te benaderen.

Deze publieke member garandeert immers - althans zolang de abstractie van de klasse ongewijzigd blijft - steeds dezelfde waarde op te leveren.

Stel dat we bij het opvragen van de Tax van een Product een percentage wensen te ontvangen:

- Code

```
class Program
{
    static void Main(string[] args)
    {
        Product product1 = new Product();
        product1.Price = 100;
        product1.Tax = 21;
        //
        Console.WriteLine(product1.Price == 100);
        Console.WriteLine(product1.Tax == 21);
        Console.WriteLine(product1.PriceIncludingTax() == 121);
        //
        Console.ReadLine();
    }
}
```

- Output

```
True
True
True
```

Dan kunnen we bijvoorbeeld deze ingestelde Tax-waarde rechtstreeks in het m_Tax-veld bewaren:

- Code

```
class Product
{
    private decimal _Price;
    public decimal Price
    {
        get { return _Price; }
        set { _Price = value; }
    }

    private decimal _Tax;
    public decimal Tax
    {
        get { return _Tax; }
        set { _Tax = value; }
    }

    public decimal PriceIncludingTax()
    {
        return Price * (1 + (Tax / 100));
    }
}
```

Indien we in een GetPriceIncludingTax-implementatie de taxwaarde wensen te gebruiken kunnen we deze waarde zowel via Tax als m_Tax deze waarde benaderen.

- Code

```
public decimal PriceIncludingTax()
{
    return Price * (1 + (Tax / 100));
}
```

En

- Code

```
public decimal PriceIncludingTax()
{
    return Price * (1 + (_Tax / 100));
}
```

Toch valt in zo'n gevallen aan te raden de publieke member te gebruiken om dergelijke waarde aan te spreken. Van 'Tax' zijn we immer zeker dat het steeds dezelfde waarde zal opleveren. We hebben immers in de abstractie vastgelegd dat indien we deze waarde opvragen we het percentage wensen te ontvangen.

Van m_Tax zijn we echter niet zo zeker, momenteel zal deze dezelfde waarde representeren.

We kunnen echter op eender welk moment de implementatie van de property Tax veranderen, en dit zonder de abstractie van het type Product aan te tasten.

3.4 OEFENINGEN

3.4.1 Oefening 1

Maak een klasse waarbij men per object de toestand (Value1 en Value2) van een Addition kan bijhouden (instellen en opvragen), en waarvan men ook de Sum kan gaan opvragen.

Schrijf zelf een client om bovenstaande klasse te testen.

3.4.2 Oefening 2

Schrijf een klasse zodoende dat je Lamp-objecten kan creëren waarop je een schakelaar kan gebruiken om het licht aan te steken.

Schrijf zelf een client om bovenstaande klasse te testen.

3.4.3 Oefening 3

Schrijf een klasse die je kan gebruiken om Counter-objecten van te creëren.

Voor zo'n 'Counter'-object moet je de 'Value' en 'StepValue' kunnen bijhouden (instellen en opvragen). Voorzie ook methods om de ingestelde 'Value' te verhogen of te verlagen met de ingestelde 'StepValue'.

De Value is initieel 0, de StepValue 1.

Schrijf zelf een client om bovenstaande klasse te testen.

3.4.4 Oefening 4

Maak een klasse die een eenvoudige robot simuleert. De robot kan zich over een X-Y coördinatenstelsel voortbewegen. Dit kan ofwel door in horizontale of verticale richting een stap te nemen (1 eenheid op te schuiven over de X- of Y-as), ofwel door te roteren (90 graden naar rechts te draaien, zonder dat de robot hierbij van positie wijzigt).

Het moet mogelijk zijn van de robot zowel de positie op de X-as als op de Y-as op te vragen.

Initieel bevindt de robot zich op positie 0 op zowel de X- als Y-as. Initieel zou de robot zich in opwaartse richting bewegen over het assenstelsel.

Schrijf zelf een client om bovenstaande klasse te testen.



4 NAMESPACES

In grote projecten (waar men met tientallen type-definities te maken heeft) kan het handig zijn om een meer hiërarchische structuur aan te brengen binnen deze type-definities.

Vergelijk het met het filesysteem van de meeste besturingssystemen. In plaats van alle bestanden in een systeem op de root te plaatsen, gaat men folders creëren om daarin bestanden onder te brengen.

Dit heeft ondermeer volgende voordelen:

- men kan door de hiërarchische structuur sneller bestanden terugvinden
- men kan verschillende bestanden met dezelfde naam creëren zolang deze maar in een verschillende folder zitten

Hetzelfde kunnen we binnen onze code verwezenlijken, maar hier maken we geen gebruik van folders, maar van "namespaces".

Toch zijn er op verschillende vlakken grote gelijkenissen met zo'n folders in een filesysteem:

- het creëert een hiërarchische structuur binnen de code
- alles wat in zo'n namespace gedefinieerd is moet een unieke naam hebben, maar 2 verschillende klassen zouden dezelfde naam kunnen hebben zolang ze maar in een verschillende namespace zitten
- namespaces kunnen net als folder in elkaar genest worden
- men kan op relatieve of absolute wijze verwijzen naar de inhoud van een namespace

Alle types worden minimaal op namespace-niveau gedefinieerd.

Bij het starten en uitkiezen van een naam van een nieuw project in Visual Studio .NET wordt steeds een namespace ("root-namespace") gecreëerd.

Deze root-namespace krijgt automatisch dezelfde naam als de naam van het project, maar men kan deze naam (in de project-properties) gaan aanpassen.

Alles wat vervolgens binnen het project wordt gedefinieerd staat in deze root-namespace.

Gezien er verschillende type-definities dezelfde naam/identifier kunnen gebruiken, kan er verwarring ontstaan bij het verwijzen naar die identifier.

Om deze verwarring tegen te gaan kan men de "fully-qualified-identifier" van dat type gaan gebruiken.

De volledig naam van een bepaald type (fully-qualified-identifier) bestaat uit de identifier van het type zelf, voorafgegaan door de onderdelen (namespaces) waarin het zich bevindt.

Zolang men zich binnen een bepaalde namespace bevindt kan men rechtstreeks verwijzen naar de inhoud van die namespace (op relatieve wijze). Bevindt men zich echter buiten die namespace, dan zal men op z'n minst de namespace moeten vermelden waarin het type is geplaatst.

Bijkomend kunnen we zelf eigen namespaces gaan toevoegen aan de hand van Namespace.

In onderstaand voorbeeld wordt gedemonstreerd hoe je verschillend type kunt creëren met dezelfde identifier, en hoe je op relatieve of absolute wijze kunt verwijzen naar zo'n types.

- Code

```
using System;
```

```

namespace Example
{
    class Person
    {
        // ...
    }

    class Program
    {
        static void Main(string[] args)
        {
            Person person1 = new Person();
            Example.Person person2 = new Example.Person();
            Persons.Person person3 = new Persons.Person();

            //
            Console.ReadLine();
        }
    }
}
namespace Persons
{
    class Person
    {
        // ...
    }
    class Program2
    {
        static void Main(string[] args)
        {
            Person person4 = new Person();
            Example.Person person5 = new Example.Person();
            Persons.Person person6 = new Persons.Person();
            //
            Console.ReadLine();
        }
    }
}

```

Ook de klassenbibliotheek van het .NET Framework (FCL) bevat tal van namespaces.

4.1 USING

In plaats van in een source-code document steeds naar de fully-qualified-identifier te verwijzen van een bepaald type, kan men ook bovenaan het source-code document (boven de type-definities) een Using-statement opnemen.

Naast het keyword Using staat steeds de fully-qualified-identifier van de namespace of klasse waarvan men op verkorte wijze naar de inhoud wenst te verwijzen.

- Code

```

using System;
using Persons;

class Person
{
    // ...
}
namespace Persons
{
    class Person
    {
        // ...
    }
    class Student : Person
    {
        // ...
    }
}

```

```

}

class Program
{
    static void Main(string[] args)
    {
        Persons.Student student1 = new Persons.Student();
        Student student2 = new Student();
        //
        Person person1 = new Person();
        Persons.Person person2 = new Persons.Person();
        //
        Console.ReadLine();
    }
}

```

Door hier bovenaan de regel "Using Persons" op te nemen, kan men op vereenvoudigde wijze de "type-specifier" voor het type `Persons.Student` formuleren.

Let ervoor op dat in sommige situaties de type-specifiers nog altijd "ambigue" kunnen zijn. Zo zal het opgenomen Using-statement geen nut hebben om op vereenvoudigde manier te verwijzen naar het type `Persons.Person`.

Imports-statements veranderen niets wezenlijks aan de programma-uitvoer, maar maken het enkel makkelijker type-specifiers te formuleren.

Men kan per source-code document Using-statements aanbrengen, of in Visual Studio .NET kan men op projectniveau meerdere imports verwezenlijken. Doordat bijvoorbeeld in een console-applicatie standaard de using naar de 'System'-namespace vermeld wordt, hoeven we niet steeds de console aan te spreken met 'System.Console', maar volstaat een verwijzing 'Console'.



5 CONTAINMENT

5.1 EEN VERBAND TUSSEN OBJECTEN

Vaak komt het voor dat objecten van (verschillende) types een verband met elkaar hebben. Dat beide types dus geassocieerd zijn.

Zo'n associatie kan ontstaan wanneer 1 object een (ander) object heeft. Van hier spreek ik in dat geval over een containment-relatie, die doorgaans gekenmerkt wordt door de "heeft-een" relatie tussen de entiteiten.

Bijvoorbeeld: een Person heeft een Address.

- Code

```
using System;

namespace Example
{
    class Person
    {
        private string _Name;
        public void SetName(string value)
        {
            _Name = value;
        }
        public string GetName()
        {
            return _Name;
        }
        private Address _Address;
        public void SetAddress(Address value)
        {
            _Address = value;
        }
        public Address GetAddress()
        {
            return _Address;
        }
    }

    class Address
    {
        private string _Street;
        public void SetStreet(string value)
        {
            _Street = value;
        }
        public string GetStreet()
        {
            return _Street;
        }
        private int _Number;
        public void SetNumber(int value)
        {
            _Number = value;
        }
        public int GetNumber()
        {
            return _Number;
        }
        private int _ZipCode;
        public void SetZipCode(int value)
        {
            _ZipCode = value;
        }
        public int GetZipCode()
        {
            return _ZipCode;
        }
    }
}
```

```

        private string _City;
        public void SetCity(string value)
        {
            _City = value;
        }
        public string GetCity()
        {
            return _City;
        }
    }

    class Program
    {
        static void Main(string[] args)
        {
            Address address1 = new Address();
            address1.SetStreet("Teststraat");
            address1.SetNumber(1);
            address1.SetZipCode(9000);
            address1.SetCity("Gent");
            //
            Person person1 = new Person();
            person1.SetName("John");
            person1.SetAddress(address1);
            //
            Console.WriteLine(person1.GetName());
            if (person1.GetAddress() != null)
            {
                Console.WriteLine(person1.GetAddress().GetNumber() + ", "
                                   + person1.GetAddress().GetStreet());
                Console.WriteLine(person1.GetAddress().GetCity() + " "
                                   + person1.GetAddress().GetZipCode());
            }
            //
            Console.ReadLine();
        }
    }
}

```

- Output

```

John
Teststraat 1
9000 Gent

```

Een object van de klasse Person kan hier beschikken over een referentie van een Address-object. Op deze wijze kan een associatie tussen deze 2 entiteiten worden bijgehouden.

In bovenstaande Main-method wordt eerst een variabele (address1) opgevuld met de referentie van een nieuw Address-object, om deze vervolgens te gebruiken als setvalue voor de eigenschap Address van Person.

Als we "person1.SetAddress(address1);" zouden wegdoen, dan zal je merken dat de code in het if-block zal evalueren naar false. De code zal dus niet uitgevoerd worden. Als we deze controle niet zouden uitvoeren, dan krijgen we een NullReferenceException tijdens de runtime, met het vastlopen van ons programma als gevolg. Vandaar dus dat we steeds dergelijke controles (if (person1.GetAddress() != null){}) altijd moeten uitvoeren.

Op zich zou men het gebruik van deze variabele (address1) kunnen vermijden.

- Code

```

class Program
{
    static void Main(string[] args)
    {
        Person person1 = new Person();
        person1.SetName("John");
        //
    }
}

```



```

        person1.SetAddress(new Address());
        person1.GetAddress().SetStreet("Teststraat");
        person1.GetAddress().SetNumber(1);
        person1.GetAddress().SetZipCode(9000);
        person1.GetAddress().SetCity("Gent");
        //
        Console.WriteLine(person1.GetName());
        if (person1.GetAddress() != null)
        {
            Console.WriteLine(person1.GetAddress().GetStreet() + " "
                               + person1.GetAddress().GetNumber());
            Console.WriteLine(person1.GetAddress().GetZipCode() + " "
                               + person1.GetAddress().GetCity());
        }
        //
        Console.ReadLine();
    }
}

```

- Output

```

John
Teststraat 1
9000 Gent

```

Bemerkt de doorgedreven puntnotatie dat gebruikt kan worden om bijvoorbeeld te verwijzen naar de eigenschap `Street` van het `Address` van een `Person`.

We zijn verplicht om `"person1.SetAddress(new Address());"` te schrijven, omdat we anders geen referentie hebben naar een addressobject, maar wel naar een `NullReference`.

5.2 HERBRUIK VAN KLASSEN EN CODE

Bemerkt ook hoe eenvoudig het is om verschillende personen gebruik te laten maken van hetzelfde adres (hetzelfde `Address`-object).

Dit kan natuurlijk enkel doordat `Address` een aparte constructie is. Hadden we de eigenschappen `Street`, `Number`, `ZipCode` en `City` in de constructie `Person` opgenomen, dan hadden we voor deze personen (`person1` en `person2`) de adresgegevens moeten dupliceren.

- Code

```

Address address1 = new Address();
address1.SetStreet("Teststraat");
address1.SetNumber(1);
address1.SetZipCode(9000);
address1.SetCity("Gent");
//
Person person1 = new Person();
person1.SetName("John");
person1.SetAddress(address1);
//
Person person2 = new Person();
person2.SetName("Jane");
person2.SetAddress(address1);

```

In bovenstaand voorbeeld werd ervoor geopteerd de client een object van het type `Address` te laten instantiëren, en vervolgens toe te kennen aan de eigenschap `Address` van het `Person`-object.

Het alternatief is objecten van het `Person`-type zelf een object van het type `Address` te laten instantiëren, zoals in onderstaand voorbeeld gebeurt in de `Person`-klasse.

- Code

```

class Person
{
    private string _Name;

```

```

    public void SetName(string value)
    {
        _Name = value;
    }
    public string GetName()
    {
        return _Name;
    }
    private Address _Address = new Address();
    public void SetAddress(Address value)
    {
        _Address = value;
    }
    public Address GetAddress()
    {
        return _Address;
    }
}

```

Het verschil zit hem hier het feit dat het veld `_Address` meteen geïntialiseerd wordt met een verwijzing naar een nieuw geïntantieerd object van het type `Address`. We hebben hier dus meteen een heleboel initialisatie.

Hierbij is het niet meer nodig de client zelf een object te laten maken van deze `Address`-klasse.

- Code

```

class Program
{
    static void Main(string[] args)
    {
        Person person1 = new Person();
        person1.SetName("John");
        person1.GetAddress().SetStreet("Teststraat");
        person1.GetAddress().SetNumber(1);
        person1.GetAddress().SetZipCode(9000);
        person1.GetAddress().SetCity("Gent");
        //
        Console.WriteLine(person1.GetName());
        if (person1.GetAddress() != null)
        {
            Console.WriteLine(person1.GetAddress().GetStreet() + " "
                               + person1.GetAddress().GetNumber());
            Console.WriteLine(person1.GetAddress().GetZipCode() + " "
                               + person1.GetAddress().GetCity());
        }
        //
        Console.ReadLine();
    }
}

```

5.3 VERANTWOORDELIJKHEID VOOR HET CREËREN VAN EEN OBJECT

Of men al dan niet het contained object laat creëren door het containing object is afhankelijk van een aantal factoren.

Indien het containing object de enigste is die moet beschikken over het contained object, of indien het containing object beschikt over de kennis welke de initialisatiedata is voor het contained object, dan zal men waarschijnlijk het containing object verantwoordelijk maken voor het creëren van het contained object ("GRASPattern Creator").

In dit geval zou je kunnen stellen dat een adres door meerdere personen of klanten kan gedeeld worden. Het lijkt dan ook het meest voor de hand liggend om de client verantwoordelijk te maken voor het expliciet instantiëren van een `Address`-object.

- Code

```

class Customer

```

```

{
    private string _Name;
    public void SetName(string value)
    {
        _Name = value;
    }
    public string GetName()
    {
        return _Name;
    }
    private Address _Address = new Address();
    public void SetAddress(Address value)
    {
        _Address = value;
    }
    public Address GetAddress()
    {
        return _Address;
    }
}

```

Je ziet ook hoe het mogelijk is in meerdere klassen (als Person en Customer) dezelfde klasse (als Address) te herbruiken. Containment kan dus leiden tot het herbruik van code.

Andere voorbeelden van containment kunnen zijn: een computer heeft een processor, een factuur heeft een klant (of anders gesteld een factuur is van toepassing op een klant), een wagen heeft een motor, een boek heeft een index,

5.4 OEFENINGEN

5.4.1 Oefening 1

Pas de klasse Robot uit vorig onderwerp aan. Laat een Robot gebruik maken van een Position, die een positie op een X-Y-assenstelsel voorstelt.

We opteren ervoor om een Robot zelf een Position-object te laten creëren. Dit omdat elk Robot-object intensief gebruik maakt van zijn containde Position-object.

De eigenschap Position in Robot dient readonly gemaakt te worden. Het is immers mogelijk om via de commands Rotate en PlaceStep de toestand van het containde Position-object te gaan manipuleren.

6 COLLECTIES - INLEIDING

Regelmatig komt het voor dat men in applicaties wil werken met een verzameling (een collectie) van objecten. Collectie-objecten (gedefinieerd door collectie-typen) kunnen ook beschouwd worden als een toepassing van containment. Deze collectietypen kunnen het eenvoudiger maken voor clients met een collectie van objecten om te gaan.

- Code

```
using System;

namespace Example
{
    class Persons
    {
        private int _Count = 0;

        private Person[] _Items = new Person[0];
        public Person Item(int index)
        {
            return _Items[index];
        }

        public int GetCount()
        {
            return _Count;
        }
        public void Add(Person person)
        {
            _Count++;
            Array.Resize(ref _Items, GetCount());
            _Items[GetCount() - 1] = person;
        }
    }
    class Person
    {
        private string _Name;
        public void SetName(string value)
        {
            _Name = value;
        }
        public string GetName()
        {
            return _Name;
        }
    }
}

class Program
{
    static void Main(string[] args)
    {
        Person person1 = new Person();
        person1.SetName("John");
        Console.WriteLine(person1.GetName());
        Person person2 = new Person();
        person2.SetName("Jane");
        Console.WriteLine(person2.GetName());
        //
        Persons persons1 = new Persons();
        Console.WriteLine(persons1.GetCount());
        persons1.Add(person1);
        persons1.Add(person2);
        Console.WriteLine(persons1.GetCount());
        Console.WriteLine(persons1.Item(0).GetName());
        Console.WriteLine(persons1.Item(1).GetName());
        //
        Console.ReadLine();
    }
}
```

- Output

```
John
Jane
0
2
John
Jane
```

Bemerkt hoe de identifier van het collectietype (PersonS) duidelijk aangeeft dat een collectie-object van dat type meerdere Person-objecten kan bevatten (meervoud gebruikt voor de identifier).

6.1 INDEXERMETHOD

In bovenstaande code vind je een "indexer" terug. Dit is een typische member voor collectie-typen waarvan de elementen via een index benaderd kunnen worden. Deze indexer levert een Person-object op (vandaar de verwijzing naar het type Person in de membersignatuur).

Hier werd ervoor geopteerd de indexer "readonly" te maken, dit is echter niet noodzakelijk, afhankelijk van de vereisten kan men deze method ook writable maken.

6.2 DEFAULT PROPERTY

Het keyword this als identifier (hier gebruikt voor de indexer) zorgt ervoor dat men in de client niet expliciet meer hoeft te verwijzen naar de identifier van de property. Let ook op de vierkante haken!

- Code

```
using System;

namespace Example
{
    class Persons
    {
        private int _Count = 0;

        private Person[] _Items = new Person[0];
        public Person this[int index]
        {
            get { return _Items[index]; }
        }

        public int GetCount()
        {
            return _Count;
        }
        public void Add(Person person)
        {
            _Count++;
            Array.Resize(ref _Items, GetCount());
            _Items[GetCount() - 1] = person;
        }
    }
    class Person
    {
        private string _Name;
        public void SetName(string value)
        {
            _Name = value;
        }
        public string GetName()
        {
            return _Name;
        }
    }
}

class Program
```

```

{
    static void Main(string[] args)
    {
        Person person1 = new Person();
        person1.SetName("John");
        Console.WriteLine(person1.GetName());
        Person person2 = new Person();
        person2.SetName("Jane");
        Console.WriteLine(person2.GetName());
        //
        Persons persons1 = new Persons();
        Console.WriteLine(persons1.GetCount());
        persons1.Add(person1);
        persons1.Add(person2);
        Console.WriteLine(persons1.GetCount());
        Console.WriteLine(persons1[0].GetName());
        Console.WriteLine(persons1[1].GetName());
        //
        Console.ReadLine();
    }
}

```

Verderop wordt behandeld welke voorgedefinieerde collectiestructuren men allemaal kan gebruiken, en hoe deze eventueel aan te passen.

6.3 OEFENINGEN

6.3.1 Oefening 1

Maak een klasse om voorstellingen te maken van rechthoeken. Een rechthoek heeft als toestand een bepaalde hoogte en breedte (instelbaar en opvraagbaar).

Van rechthoeken moet men ook de oppervlakte kunnen bepalen.

Creëer ook een collectietype om een verzameling van rechthoeken te beheren.

De elementen in zo'n collectie moet men via een index kunnen aanspreken.

Aan de collectie moet men rechthoeken kunnen toevoegen, en rechthoeken op een bepaalde index kunnen verwijderen.

De som van alle oppervlaktes van alle rechthoek-elementen moet men kunnen opvragen van de collectie.

Schrijf zelf een client om bovenstaande klassen te testen.



7 INHERITANCE (OVERERVING)

7.1 WAARVOOR IS INHERITANCE BRUIKBAAR

Het kan voorkomen dat je nieuwe klasse die je wenst te definiëren zou willen baseren op een andere klasse. Waarbij je dus stelt dat hetgene dat gedefinieerd is in die andere klasse zou willen herbruiken in je nieuwe klasse.

Hier komt overerving (ook wel inheritance genoemd) van pas. Inheritance zorgt ervoor dat de afgeleide klasse (derived/child class) alle members overneemt van de basisklasse (base/parent class).

Hieronder zie je een voorbeeld van inheritance.

- Code

```
using System;

namespace Example
{
    class Person
    {
        private string _Name;
        public void SetName(string value)
        {
            _Name = value;
        }
        public string GetName()
        {
            return _Name;
        }
    }

    class Student : Person
    {
        private string _ClassGroup;
        public void SetClassGroup(string value)
        {
            _ClassGroup = value;
        }
        public string GetClassGroup()
        {
            return _ClassGroup;
        }
    }

    class Program
    {
        static void Main(string[] args)
        {
            Person person1 = new Person();
            person1.SetName("John");
            Console.WriteLine(person1.GetName());
            //
            Student student1 = new Student();
            student1.SetName("Jane");
            student1.SetClassGroup("Group 1");
            Console.WriteLine(student1.GetName());
            Console.WriteLine(student1.GetClassGroup());
            //
            Console.ReadLine();
        }
    }
}
```

- Output

```
John
```

```
Jane  
Group 1
```

De Student-klasse erft hier over van de Person-klasse, en neemt daarbij ook alle members over die gedefinieerd zijn in die basisklasse. In onderstaande client kun je merken dat men op een object van het type Student nu ook de (overgeërfde) method `GetName()` kunt gebruiken. Naast `GetName()` kan natuurlijk bijvoorbeeld ook `GetClassGroup()` worden gebruikt die in de klasse Student zelf werd gedefinieerd.

7.2 HOE INHERITANCE GEBRUIKEN

Inheritance wordt toepast door in de afgeleide klasse een ":"-clause op te nemen.

- Code

```
class Teacher : Person  
{  
    private string _Course;  
    public void SetCourse(string value)  
    {  
        _Course = value;  
    }  
    public string GetCourse()  
    {  
        return _Course;  
    }  
}
```

7.3 SINGLE INHERITANCE

Zoals je hierboven merkt kan men van 1 basisklasse meerdere afgeleide klassen definiëren. Men kan in C#.NET echter wel slechts van 1 klasse overerven ("single inheritance"), in tegenstelling tot bepaalde andere OOP's waar men van meerdere klassen kan overerven ("multiple inheritance").

7.4 SINGLE EN STATIC CLASSIFICATION

Een object is in C#.NET altijd een instantie van 1 bepaalde klasse ("single classification"), dat type van object kan ook nooit wijzigen ("static classification"), in tegenstelling tot sommige OOP's waar men een object kan maken van meerdere klassen ("multiple classification") of een object van type/klasse kan laten veranderen ("dynamic classification").

7.5 WANNEER INHERITANCE TOEPASSEN

Wanneer inheritance van toepassing is tussen 2 klassen, is er doorgaans een "is een"-relatie tussen objecten van die 2 entiteiten. In dit voorbeeld kan je stellen een Student of een Teacher is een Person.

Verwar dit niet met de "heeft een"-relatie die bij containment van toepassing is.

Enkele voorbeelden waarbij inheritance van toepassing zou kunnen zijn: een woonhuis is een gebouw, een wagen is een voertuig, een bediende is een werknemer, een tekstverwerker is een programma,

Let echt op dat men gespecialiseerde (afgeleide) types niet verward met instanties, bijvoorbeeld John is een persoon. Hierbij is John eerder een instanties, en niet zozeer een subtype.

Een afgeleide klasse kan ook een basisklasse vormen voor andere afgeleide klassen.

Eigenlijk dient gesteld te worden dat iedere klasse die je zelf definieert een afgeleide klasse is. Voeg je zelf niet expliciet een ":"-clause toe aan een klasse, dan staat er eigenlijk by default ":" `System.Object`". De klasse Person bijvoorbeeld is dus een afgeleide van het type `System.Object`.

Later wordt meer uitleg gegeven over dit overkoepelend basistype (System.Object).

Inheritance zowel als containment zijn 2 technieken om herbruik van code te verwezenlijken. Bij containment kan je de members van het gerefereerde object aanspreken en dus herbruiken, bij inheritance worden de members gewoon overgeërfd, waardoor ze ook herbruikbaar zijn op of voor dat object van de afgeleide klasse.

Misbruik echter inheritance niet om simpelweg in uw afgeleide klasse de members niet te hoeven herdefiniëren die zijn overgeërfd. De kracht van inheritance is immers niet zo zeer herbruik van code, maar eerder "dynamisch polymorfisme" waarbij objecten een andere gespecialiseerde vorm kunnen aannemen. (later meer over polymorfisme)

Goede richtlijnen die je als vereiste kan hanteren voor het gebruik van inheritance zijn:

- de "is een"-regel
- de "100%"-regel: 100% van de members (of dus alle members) die gedefinieerd zijn in de basisklasse moeten ook bruikbaar zijn in de afgeleide klasse

Het valt aan te raden pas inheritance toe te passen indien aan beide regels voldaan is.

Anders gesteld zijn er twee invalshoeken om tot inheritance te komen:

- Generalisatie
hier merk je dat verschillende klassen gemeenschappelijke members hebben, waarbij je dus de beslissing zou kunnen maken een basistype te definiëren waarin het gemeenschappelijke wordt gedefinieerd, ander typen die dan ook deze gemeenschappelijke members nodig hebben, kunnen dan ook van die basisklasse overerven
- Specialisatie
hier bemerk je eerder dan een nieuw type dat je dient te definiëren eigenlijk een specialisatie is van een reeds bestaand type, waardoor je besluit van dat bestaand type over te erven en daaraan de gespecialiseerde kenmerken gaat toevoegen

Het resultaat is bij generalisatie of specialisatie hetzelfde, namelijk overerving, het zijn louter verschillende uitgangspunten van waaruit je kan vertrekken om tot inheritance te komen.

Onthoud voorlopig ook nog dat eens een member met een bepaalde signatuur is overgenomen uit een basisklasse, je niet zomaar in de afgeleide klasse een member kan toevoegen met dezelfde signatuur. Later gaan we hier verder op in.

7.6 EEN ANDER VOORBEELD

We gaan in onderstaand voorbeeld uit van een klasse Counter, waarbij men op objecten van dit type de Value (tellerwaarde) kan opvragen (via de GetValue()-method). En waarbij de mogelijkheid bestaat via members in de publieke interface van dit type de Value te verhogen (Raise()-method) of te verlagen (Lower()-method).

- Code

```
class Counter
{
    protected int _Value;
    public int GetValue()
    {
        return _Value;
    }
    public void Raise()
    {
```

```

        _Value++;
    }
    public void Lower()
    {
        _Value--;
    }
}

```

Naderhand wensen we een gespecialiseerd type van Counter te creëren, namelijk een specialisatie waar het ook mogelijk wordt de Value in te stellen.

We beslissen hiervoor een afgeleide klasse SettableCounter te creëren en er een SetValue(...) -method aan toe te voegen.

Allerhande redenen kunnen ervoor zorgen dat je Counter niet wil of kan wijzigen. Het zou simpelweg kunnen voorvallen dat je niet wil riskeren om de bestaande klasse corrupt te maken. Het zou ook kunnen dat je de abstractie van de klasse niet wil aanpassen, een waarde-instelbare en waarde-niet-instelbare teller is immers niet hetzelfde. Het zou ook kunnen dat je de basisklasse niet kan aanpassen omdat je niet over de broncode beschikt, en enkel over de gecompileerde klasse beschikt.

- Code

```

using System;

namespace Example
{
    class Counter
    {
        protected int _Value;
        public int GetValue()
        {
            return _Value;
        }
        public void Raise()
        {
            _Value++;
        }
        public void Lower()
        {
            _Value--;
        }
    }

    class SettableCounter : Counter
    {
        public void SetValue(int value)
        {
            _Value = value;
        }
    }

    class Program
    {
        static void Main(string[] args)
        {
            Counter counter1 = new Counter();
            Console.WriteLine(counter1.GetValue() == 0);
            counter1.Raise();
            counter1.Raise();
            counter1.Raise();
            Console.WriteLine(counter1.GetValue() == 3);
            counter1.Raise();
            counter1.Raise();
            counter1.Lower();
            Console.WriteLine(counter1.GetValue() == 4);
            //
            SettableCounter counter2 = new SettableCounter();
            Console.WriteLine(counter2.GetValue() == 0);
            counter2.Raise();
            counter2.Raise();
        }
    }
}

```

```

        Console.WriteLine(counter2.GetValue() == 2);
        counter2.SetValue(100);
        Console.WriteLine(counter2.GetValue() == 100);
        counter2.Lower();
        counter2.Lower();
        Console.WriteLine(counter2.GetValue() == 98);
        //
        Console.ReadLine();
    }
}

```

- Output

```

True
True
True
True
True
True
True
True

```

7.7 ACCESS MODIFIER PROTECTED

Zoals we reeds wisten zijn Private-members enkel beschikbaar binnen de klasse zelf en zijn Public-members overal (inclusief voor de client) bereikbaar. Deze keywords geven aan of wijzigen dus het toegangs-/access-bereik van die members, vandaar de term "access modifiers".

Protected-members zijn best vergelijkbaar met Private-members, alleen zijn ze niet louter binnen de klasse (waarin ze zijn gedefinieerd) beschikbaar, maar zijn ze ook toegankelijk voor de afgeleide klasse. Clients echter hebben geen toegang tot deze members.

Het veld `_Value` gedefinieerd in de basisklasse `Counter` mag ingekapseld worden (dus niet Public) omdat het niet toegankelijk moet zijn voor de clients van `Counter`.

Afgeleide klassen van `Counter` zouden echter wel nut kunnen hebben aan toegang tot het veld `_Value`. Zo geeft `SettableCounter` het veld `_Value` nodig om in de `SetValue`-method de `Value`-toestand te kunnen wijzigen. Daarom kan hier aan het veld `_Value` hier een Protected access modifier worden gegeven.

Men zou dus steeds bij het opnemen van ingekapselde members (niet publieke members) moeten nagaan of afgeleiden nut kunnen hebben aan toegang tot deze member. Hebben de afgeleiden geen nut aan toegang tot die member dan kan die member als Private worden gedefinieerd, is er wel nut voor de afgeleiden aan toegang tot die member dan kan deze member als Protected worden gedefinieerd.

Voor alle duidelijkheid dus nog eens stellen: Public-members die zijn overgeërfd zijn ook beschikbaar voor client op objecten van dat afgeleid type.

Vergis je echter niet: niet alleen Public- en Protected-members worden overgeërfd, ook de Private members worden overgenomen, alleen zijn die niet toegankelijk, zelfs niet voor afgeleide klassen. Het gebruik van Private- en Protected-members is dus een vorm van encapsulation (inkapseling) waarvan het gebruik van die members voor de client (Protected) en eventueel ook voor de afgeleide klasse zelf (Private) verhinderd wordt.

7.8 OEFENINGEN

7.8.1 Oefening 1

Maak de nodige klassen om voorstellingen van personeelsleden te kunnen maken.

Een personeelslid heeft als eigenschappen: een naam en een adres (een adres houdt in een straat, een huisnummer, een postcode en een gemeente).

Zorg er ook voor dat we voorstellingen kunnen maken van managers.

Managers zijn personeelsleden met meer verantwoordelijkheden en hebben als eigenschappen: een naam, een adres en een wagen (van een bepaald merk).

7.8.2 Oefening 2

Hieronder worden een aantal zelfstandige naamwoorden of eigennamen vermeld:

Sarkozy, monarch, Paris, country, United Kingdom, president, monarchy, France, capital, London, Elizabeth, republic

Gebruik zelf alle bovenvermelde woorden in uw oplossing. Kies zelf of het woord best past als klasse-identificer, member-identificer, toestands-waarde of object-identificer.

Pas inheritance toe, daar waar je denkt dat het nuttig is.

8 METHOD OVERLOADING

We vertrekken van een collectie-type `Persons` die een collectie van objecten van een item-type `Person` kan beheren.

Naderhand wensen we een gespecialiseerd type van `Persons` te creëren, namelijk een specialisatie waar het ook mogelijk is een `Person`-item uit de collectie op te vragen via een "Name".

We beslissen hiervoor een afgeleide klasse `QueryablePersons` te maken, waar we de members van `Persons` aanvullen met een extra member `Item`.

Dat we `Item` als identifier voor deze member kiezen is logisch, het gaat tenslotte ook over een member die een `Item`-oplevert van de collectie in `Person`-objectvorm.

Van de basisklasse `Persons` erver we echter reeds een `Item` over.

Om ervoor te zorgen dat een aanroep naar deze members `Item` ondubbelzinnig is (duidelijk welke `Item` wordt aangeroepen), moeten we een verschil maken in de argumenten.

- Code

```
using System;
namespace Example
{
    class Persons
    {
        private int _Count = 0;

        protected Person[] _Items = new Person[0];
        public Person Item(int index)
        {
            return _Items[index];
        }

        public int GetCount()
        {
            return _Count;
        }
        public void Add(Person person)
        {
            _Count++;
            Array.Resize(ref _Items, GetCount());
            _Items[GetCount() - 1] = person;
        }
    }

    class QueryablePersons : Persons
    {
        public Person Item(string name)
        {
            int index = 0;
            bool found = false;
            //
            while (index < GetCount() && !found)
            {
                if (_Items[index].GetName() == name)
                {
                    found = true;
                }
                else
                {
                    index++;
                }
            }
            if (found)
            {
                return _Items[index];
            }
        }
    }
}
```

```

    }
    else
    {
        return null;
    }
}

class Person
{
    private string _Name;
    public void SetName(string value)
    {
        _Name = value;
    }
    public string GetName()
    {
        return _Name;
    }
}

class Program
{
    static void Main(string[] args)
    {
        Person person1 = new Person();
        person1.SetName("John");
        Person person2 = new Person();
        person2.SetName("Jane");
        //
        Persons persons1 = new Persons();
        persons1.Add(person1);
        persons1.Add(person2);
        Console.WriteLine(persons1.Item(0).GetName());
        Console.WriteLine(persons1.Item(1).GetName());
        //
        QueryablePersons persons2 = new QueryablePersons();
        persons2.Add(person1);
        persons2.Add(person2);
        Console.WriteLine(persons2.Item("John") == person1);           // true
        Console.WriteLine(persons2.Item("Jane") == person2);           // true
        Console.WriteLine(persons2.Item("Philippe") == person2);       // false
        //
        Console.ReadLine();
    }
}
}

```

- Output

```

John
Jane
True
True
False

```

Meerder members met dezelfde naam (in dezelfde klasse) kan door "method overloading" te gebruiken.

Item (met Integer-argument) die hier in QueryablePersons wordt overgeerd van Persons wordt hier overladen met een extra versie van Item (met String-argument).

De compiler zal bij een aanroep naar deze Item-property de best passende versie uitkiezen gebaseerd op de argumenten. Geeft men bijvoorbeeld een Integer-argumentwaarde op dan wordt dit beschouwd als een aanroep naar Item uit Persons. Geeft men een String-argument door bij de aanroep naar Item dan zal de compiler dit beschouwen als een aanroep naar Item gedefinieerd in QueryablePersons.

Hoe verschillende versies van een method of property van elkaar moeten verschillen om nog steeds voor de compiler duidelijk te houden welke versie je aanroept is bepaald in de "overloadresolution" van het "metamodel" van de gebruikte taal.

In C#.NET kunnen we verschillende versies creëren van een method door een verschil te maken in datatypes van de argumenten, aantal argumenten en volgorde van de datatypes van argumenten.

Wees zuinig met het gebruik van overloading in uw eigen klassen, in combinatie met inheritance kan dit tot uiterst moeilijk leesbare en voorspelbare code leiden. Later meer over dit probleem.

8.1 OEFENINGEN

8.1.1 Oefening 1

Schrijf een klasse die je kan gebruiken om Counter-objecten van te creëren.

Voor zo'n counter-object moet je de waarde enkel kunnen opvragen. Voorzie ook methods om de waarde te verhogen en te verlagen.

De verhoog- en verlaag-stapwaarde mag hier steeds 1 zijn.

Schrijf ook een klasse die bruikbaar is om SpecialCounter-objecten van te creëren. Deze objecten dienen net als Counter-objecten over een waarde te kunnen beschikken, die deze keer ook moet kunnen worden ingesteld.

Bijkomend moet het bij SpecialCounter-objecten niet alleen mogelijk zijn met gelijke methods als van Counter-objecten de waarde met 1 te verhogen of te verlagen, maar het moet ook mogelijk zijn met extra (vergelijkbare) methods de waarde met een specifiek opgegeven waarde te verhogen of te verlagen.



9 CONSTRUCTOREN

In de client tot nu toe hebben we bijna steeds meteen na het instantiëren van een object eigenschapswaardes ingesteld voor dat object. Dit is vrij logisch, want met een "leeg" object zijn we immers meestal niet veel. Met "leeg" wordt hier bedoeld: zonder objectspecifieke toestand, zonder specifieke waardes (andere dan de initiële) voor die instantie.

Een veelgebruikte werkwijze om het voor de clients eenvoudiger te maken initiële waardes te bepalen voor eigenschappen, is het gebruik van constructoren. Dit zijn steeds procedures met de identifier die dezelfde naam draagt als de naam van de klasse zelf. Deze constructoren worden impliciet aangeroepen bij het gebruik van een object initializer (keyword New gevolgd door de identifier van de klasse die men wil instantiëren).

Men kan in de constructor argument-variabelen definiëren, waardoor het mogelijk wordt bij het instantiëren (in de object-initializer) de nodige argumentwaardes door te geven.

Hieronder een voorbeeld van een klasse Person met een constructor die toelaat een initiële Name-eigenschapswaarde op te geven in de object initializer.

- Code

```
using System;

namespace Example
{
    class Person
    {
        public Person(string name)
        {
            this.Name = name;
        }

        public string Name { get; set; }
    }

    class Program
    {
        static void Main(string[] args)
        {
            Person person1 = new Person("John");
            Person person2 = new Person("Jane");
            Console.WriteLine(person1.Name);
            Console.WriteLine(person2.Name);
            //
            Console.ReadLine();
        }
    }
}
```

- Output

```
John
Jane
```

De client MOET hier nu meteen bij het construeren van het object (van het type Person) een initiële Name-waarde opgeven. De doorgegeven argumentwaarde (name) wordt vervolgens toegekend aan de eigenschap (this.Name). Dit vereenvoudigt natuurlijk sterk de client-code.

Onderstaande code zal dus niet meer werken, aangezien we alleen een constructor hebben voorzien met 1 argument. We worden dus verplicht om bij het creëren van een object de initiële waarden te voorzien.

- Code

```
Person person1 = new Person();
```


Ook clients (of clientprogrammeurs) kunnen de naam van argumentvariabelen zien. Bedenk maar hoe bijvoorbeeld in de Visual Studio IDE er een tooltip komt met informatie over de verwachte te doorgegeven argument-waardes als je een method aanroept. In deze informatie staat ook de identifier van de argumentvariabelen vermeldt. Een clientprogrammeur kan steeds de naam van de argumentvariabelen zien.

Dit heeft als gevolg dat we er best voor opteren om zo duidelijk mogelijke identifiers uit te kiezen voor deze argumentvariabelen. Liefst een naam die de rol van de doorgegeven argumentwaarde weerspiegelt.

Gezien het in bovenstaand voorbeeld gaat om een initiële waarde voor een property die ook beschikbaar is voor clients, ligt de keuze voor de identifier van de gerelateerde argumentvariabele voor de hand. Namelijk dezelfde identifier als die voor de gerelateerde property (Name-property en name-argumentvariabele).

Om het voor de compiler duidelijk te houden dat de argumentwaarde moet toegekend worden aan de property (en niet opnieuw aan de argumentvariabele zelf) verwijzen we naar deze property door er de objectexpressie `Me` voor te plaatsen.

`"this"` is steeds een verwijzing naar het object dat op dat moment in uitvoering is. Men kan naar eender welke instancemember verwijzen binnen een klasse door voor de identifier van die member `"this."` te plaatsen, dit is niet noodzakelijk, maar kan eventueel verduidelijken dat het om een aanroep naar een instancemember gaat. `"this"` is enkel noodzakelijk om mogelijke verwarring te vermijden voor de compiler bij het verwijzen een member van het object (dat op dat moment in uitvoering is) wanneer een andere member of variabele dezelfde identifier zou gebruiken.

Constructoren worden niet alleen gebruikt voor het instellen van initiële waardes voor eigenschappen. Eender wat moet gebeuren bij het construeren van een object kan men in de constructor coderen.

9.1 DEFAULT CONSTRUCTOR EN MYBASE

Wat niet opvalt (omdat het achter de schermen wordt toegevoegd) is dat elke klasse die je definieert een constructor heeft. Definieer je dus zelf niet expliciet een constructor in een klasse dan wordt er achter de schermen - wat men noemt - een default constructor toegevoegd.

Deze default constructor ziet er steeds als volgt uit:

- Code

```
public Person() : base() {}
```

Het gaat dus om een publiek beschikbare constructor (access modifier `Public`). Deze dient publiek beschikbaar te zijn, anders kan men geen object instantiëren van dit type.

Onderstaande klasse bijvoorbeeld heeft een constructor die ingekapseld is.

- Code

```
using System;

namespace Example
{
    class Person
    {
        private Person() : base() {}

        public string Name { get; set; }
    }

    class Program
    {
        static void Main(string[] args)
        {
            // Person person1 = new Person();
        }
    }
}
```

```

        //
        Console.ReadLine();
    }
}

```

Clients hebben hier dus niet de mogelijkheid om een object van dit type te construeren. Het inkapselen van constructoren kan bijvoorbeeld nuttig zijn om het "singleton pattern" te implementeren, hierover later meer.

Uit bovenstaand voorbeeld is ook duidelijk geworden dat indien men zelf expliciet een constructor definieert in een klasse, er geen sprake meer is van de default constructor. Deze wordt dus enkel toegevoegd indien er nog geen aanwezig zou zijn.

Voegt men dus bijvoorbeeld een constructor met argumenten toe aan een klasse, dan is men verplicht argumentwaarden op te geven in een object initializer voor het instantiëren van een object van dat type. (althans wanneer er geen extra argumentloze constructor aanwezig zou zijn)

De default constructor is uiteraard argumentenloos. Het systeem dat achter de schermen deze constructor toevoegt heeft immers geen idee welke parameters van nut zouden kunnen zijn. En bijkomend ook geen idee van hebben wat aan te vangen met de doorgegeven argumentwaarden.

Op de eerste implementatieregel van deze default constructor staat steeds een aanroep naar de parameterloze constructor van de basisklasse. We weten ondertussen dat iedere klasse die we definiëren een afgeleide klasse is.

Geven we niet zelf op van welk type onze klasse overerft, dan wordt automatisch overgeërfd van het type `System.Object`.

Het `base`-keyword is vergelijkbaar met `this`, maar kan gebruikt worden om naar members te verwijzen die in de basisklasse zijn gedefinieerd. Dit kan gaan over overgeërfde members of over constructoren van de basisklasse.

Constructoren worden immers nooit overgeërfd in C#.NET, dus een aanroep naar die constructor gedefinieerd in die basisklasse kan enkel aan de hand van het `base`-keyword.

Een verschil tussen `this` en `base` is dat `this` een object-expressie is, in tegenstelling tot `base`.

Onthoud dus dat bij het construeren van een object steeds alle constructoren van de basisklassen worden uitgevoerd.

Bemerkt wel dat het by default gaat om een aanroep naar de parameterloze constructor van de basisklasse. Men zal deze default-aanroep moeten vervangen (door een expliciete call) naar een niet parameterloze constructor indien de basisklasse niet over een zo'n parameterloze constructor beschikt.

Zoals ook in onderstaand voorbeeld wordt verduidelijkt.

- Code

```

using System;

namespace Example
{
    class Person
    {
        public Person(string name)
        {
            this.Name = name;
        }

        public string Name { get; set; }
    }

    class Student : Person

```

```

{
    public Student(string name, string classgroup) : base(name)
    {
        this.ClassGroup = classgroup;
    }

    public string ClassGroup { get; set; }
}

class Program
{
    static void Main(string[] args)
    {
        Person person1 = new Person("John");
        Console.WriteLine(person1.Name);
        //
        Student student1 = new Student("Jane", "A5");
        Console.WriteLine(student1.Name);
        Console.WriteLine(student1.ClassGroup);
        Console.ReadLine();
    }
}

```

- Output

```

John
Jane
A5

```

Hier beschikt de klasse Student enkel over een constructor die twee argumentwaardes verwacht (de default constructor is immers vervangen door de expliciet gedefinieerde constructor). Dus moet in de client van het type Student in de object initializer ook argumentwaardes voor die argumenten worden opgegeven.

Vaak komt dit handig uit, zo ook in dit voorbeeld, die initialisatiecode die hier de Name-eigenschap op een initiële waarde zet, kan zo worden herbruikt, en hoeft niet te worden geherimplementeerd in de constructor van de afgeleide klasse Student.

9.2 MEERDERE CONSTRUCTOREN IN EEN KLASSE

In 1 klasse kunnen meerdere constructoren worden opgenomen, dit is een vorm van method overloading.

- Code

```

using System;

namespace Example
{
    class Person
    {
        public Person(string name)
        {
            this.Name = name;
        }

        public string Name { get; set; }
    }

    class Student : Person
    {
        public Student(string name, string classgroup) : base(name)
        {
            this.ClassGroup = classgroup;
        }

        public string ClassGroup { get; set; }
    }

    class Teacher : Person
    {

```

```

    public Teacher(string name) : base(name) {}
    public Teacher(string name, string course) : base(name)
    {
        this.Course = course;
    }

    public string Course { get; set; }
}

class Program
{
    static void Main(string[] args)
    {
        Person person1 = new Person("John");
        Console.WriteLine(person1.Name);
        //
        Student student1 = new Student("Jane", "A5");
        Console.WriteLine(student1.Name);
        Console.WriteLine(student1.ClassGroup);
        //
        Teacher teacher1 = new Teacher("Elisa");
        Console.WriteLine(teacher1.Name);
        Console.WriteLine(teacher1.Course);
        Teacher teacher2 = new Teacher("Philippe", "Programmeren");
        Console.WriteLine(teacher2.Name);
        Console.WriteLine(teacher2.Course);
        //
        Console.ReadLine();
    }
}

```

- Output

```

John
Jane
A5
Elisa

Philippe
Programmeren

```

9.3 READONLY VELDEN

Naast ReadOnly-properties (die enkel instelbaar zijn, en dus enkel over een "setter" beschikken) kan men ook ReadOnly-velden declareren.

Aan deze ReadOnly-velden kan enkel een waarde worden toegekend in de constructor.

- Code

```

using System;

namespace Example
{
    class Product
    {
        public Product(int price, int tax)
        {
            this.Price = price;
            this._TaxPercentage = tax;
        }

        public decimal Price { get; private set; }

        private readonly decimal _TaxPercentage;
        public decimal TaxPercentage
        {
            get { return _TaxPercentage; }
        }

        public decimal PriceIncludingTax()

```

```

        {
            return Price * (1 + (TaxPercentage / 100));
        }
    }

    class Program
    {
        static void Main(string[] args)
        {
            Product product1 = new Product(200, 21);
            Console.WriteLine(product1.PriceIncludingTax());
            //
            Product product2 = new Product(200, 10);
            Console.WriteLine(product2.PriceIncludingTax());
            //
            Console.ReadLine();
        }
    }
}

```

- Output

```

242,00
210,00

```

In bovenstaand voorbeeld wordt een `TaxPercentage`-waarde beschouwd als een toestand die per `Product`-object configureerbaar moet zijn (dus een instance-gerelateerde toestand). Hiervoor kunnen we dus een veld gebruiken om de toestandswaarde te stockeren.

Maar de `TaxPercentage`-toestand wordt hier ook beschouwd als een onwijzigbare toestand. Wat we kunnen bekomen en garanderen door het gebruikte veld `ReadOnly` te declareren.

9.4 INITIALISATIE VELDEN

Bij het creëren van een object van een bepaalde klasse, zullen eerst de velden geïnitialiseerd worden, waarna de constructor wordt uitgevoerd.

- Code

```

using System;

namespace Example
{
    class ValuesStore
    {
        public static string GetValue(string message)
        {
            Console.WriteLine("GetValue: " + message);
            return "SomeValue";
        }
    }

    class AClass
    {
        public AClass()
        {
            m_Value2 = ValuesStore.GetValue("m_Value2");
        }
        private string m_Value1 = ValuesStore.GetValue("m_Value1");
        private string m_Value2;
        private string m_Value3 = ValuesStore.GetValue("m_Value3");
    }

    class Program
    {
        static void Main(string[] args)
        {
            AClass someObject = new AClass();
            //
            Console.ReadLine();
        }
    }
}

```

```

    }
}

```

- Output

```

GetValue: m_Value1
GetValue: m_Value3
GetValue: m_Value2

```

Men moet ook steeds in gedachte houden dat bij het creëren van een object eerst de constructor van de basisklasse wordt uitgevoerd, nog vooraleer de constructor van de klasse zelf (die geïnstantieerd wordt) wordt uitgevoerd.

Dit kan echter tot subtiele fouten leiden bij inheritance.

Bestudeer eens onderstaande code en probeer te voorspellen wat er zal gebeuren.

- Code

```

using System;

namespace Example
{
    class Class_K
    {
        public Class_K()
        {
            Console.WriteLine(Value());
        }

        public virtual string Value()
        {
            return "Class_K";
        }
    }

    class Class_L : Class_K
    {
        public Class_L()
        {
            String[] _Values = { "Class_La", "Class_Lb", "Class_Lc" };

            private string[] _Values = new string[0];

            public override string Value()
            {
                return _Values[1];
            }
        }
    }

    class Program
    {
        static void Main(string[] args)
        {
            Class_K object1 = new Class_K();
            Class_L object2 = new Class_L();
            //
            Console.ReadLine();
        }
    }
}

```

Op het eerste zicht zou je hier verwachten dat de objectinitializer New Class_L ervoor zal zorgen dat "Class_Lb" op de console wordt geschreven.

Toch is dit niet het geval, bij uitvoer van bovenstaand voorbeeld krijgen we een runtime-fout.

De call naar member Value in de constructor van de basisklasse Class_K wordt dynamisch gebonden aan de implementatie voor Value die gedefinieerd is in de klasse Class_L.

Het veld `_Values` waarvan de Value-implementatie in `Class_L` gebruikt maakt is nog niet geïnitieerd op het moment dat de constructor van de basisklasse `Class_K` wordt uitgevoerd. Hierdoor wordt een runtime-fout veroorzaakt.

Het probleem zou niet opgelost worden door het veld meteen te initialiseren bij declaratie. Ook deze initialisatie zou pas na het uitvoeren van de constructor van de basisklasse `Class_K` worden uitgevoerd.

Dit is een probleem die niet in alle OOP's zou voorkomen. Sommige OOP's gebruik altijd statische binding voor membercalls die gebeuren in de constructor. En dit net om bovenvermelde problemen te vermijden.

9.5 OEFENINGEN

9.5.1 Oefening 1

Schrijf een klasse `Counter`. Van een `Counter`-object kan je de waarde opvragen. Voorzie ook methods om de waarde te verhogen en te verlagen. De verhoog- en verlaag-stapwaarde mag hier steeds 1 zijn.

Het is verplicht bij het instantiëren van een `Counter`-object een initiële waarde op te geven.

10 DOCUMENTATIE MAKEN

10.1 INLEIDING

Wanneer je de programmacode die je schrijft goed voorziet van commentaar, dan zal je code gemakkelijker te begrijpen zijn wanneer je er later naar terugkijkt of wanneer je die doorgeeft aan een andere programmeur.

Wanneer je commentaar op een specifieke manier aanbrengt, kan je met Visual Studio.net automatisch documentatierapporten laten genereren.

10.2 CODE DOCUMENTEREN

Om code te documenteren kan je standaard gebruik maken van `//` voor commentaar op een regel en `/* ... */` voor commentaar op meerdere regels.

Wens je echter met Visual Studio.net automatisch een 'Comment Report' aan te maken, dan plaats je commentaar naast `///` en gebruik je vooraf gedefiniëerde XML-markeringen:

- `summary` Beschrijving van een member: klasse, variabele, methode, constructor, ...
- `c` Code in een regel
- `code` Codeblok van meerdere regels
- `returns` Retourwaarde van een methode
- `param` Parameter voor een methode

10.3 VOORBEELD

Als voorbeeld zie je hieronder de klasse `Persoon` met VS.net commentairstijl:

- Code

```
using System;

namespace Personen
{
    /// <summary>
    /// Klasse voor een Persoon
    /// </summary>
    public class Persoon
    {
        /// <summary>
        /// Enumeratie voor het geslacht.<br></br>
        /// Zo kan enkel Mannelijk of Vrouwelijk worden gekozen
        /// </summary>
        public enum Geslacht { Mannelijk, Vrouwelijk }

        /// <summary>
        /// Naam van de persoon
        /// </summary>
        protected string naam;
        /// <summary>
        /// Geslacht van de persoon
        /// </summary>
        protected Geslacht geslacht;

        /// <summary>
        /// Constructor
        /// </summary>
    }
}
```



```

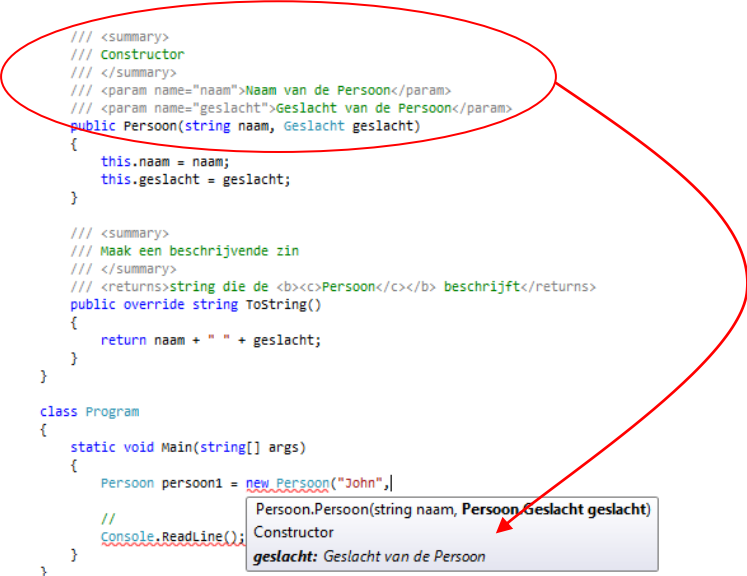
    /// <param name="naam">Naam van de Persoon</param>
    /// <param name="geslacht">Geslacht van de Persoon</param>
    public Persoon(string naam, Geslacht geslacht)
    {
        this.naam = naam;
        this.geslacht = geslacht;
    }

    /// <summary>
    /// Maak een beschrijvende zin
    /// </summary>
    /// <returns>string die de <b><c>Persoon</c></b> beschrijft</returns>
    public override string ToString()
    {
        return naam + " " + geslacht;
    }
}

class Program
{
    static void Main(string[] args)
    {
        Persoon persoon1 = new Persoon("John", Persoon.Geslacht.Mannelijk);
        //
        Console.ReadLine();
    }
}

```

Je kan op deze manier een rapport maken van een volledige Solution of Project.



```

    /// <summary>
    /// Constructor
    /// </summary>
    /// <param name="naam">Naam van de Persoon</param>
    /// <param name="geslacht">Geslacht van de Persoon</param>
    public Persoon(string naam, Geslacht geslacht)
    {
        this.naam = naam;
        this.geslacht = geslacht;
    }

    /// <summary>
    /// Maak een beschrijvende zin
    /// </summary>
    /// <returns>string die de <b><c>Persoon</c></b> beschrijft</returns>
    public override string ToString()
    {
        return naam + " " + geslacht;
    }
}

class Program
{
    static void Main(string[] args)
    {
        Persoon persoon1 = new Persoon("John",
        //
        Console.ReadLine();
    }
}

```

Persoon.Persoon(string naam, Persoon.Geslacht geslacht)
 Constructor
geslacht: Geslacht van de Persoon

11 REDEFINITION

11.1 VIRTUAL EN OVERRIDE

Wanneer men aan een overgeërfde member een andere implementatie wenst te koppelen kan men deze overgeërfde member gaan herdefiniëren. Dit wordt ook wel "redefinition" of in de context van C#.NET "overriding" genoemd.

Een vereiste is wel dat de member die je wenst te herdefiniëren in de basisklasse als herdefinieerbaar staat aangegeven.

Dit kan door in de signatuur van deze member het virtual-keyword op te nemen.

De member wordt herhaald in de klassen en staat daar als herdefinitie aangegeven. Wat men bekomt door aan de signatuur van deze herdefinitie het override-keyword toe te voegen.

- Code

```
using System;

namespace Example
{
    class Class1
    {
        public virtual string Method1()
        {
            return "Class1.Method1()";
        }
    }

    class Class2 : Class1
    {
        public override string Method1()
        {
            return "Class2.Method1()";
        }
    }

    class Program
    {
        static void Main(string[] args)
        {
            Class1 object1 = new Class1();
            Console.WriteLine(object1.Method1()); // 1
            //
            Class2 object2 = new Class2();
            Console.WriteLine(object2.Method1()); // 2
            //
            Console.ReadLine();
        }
    }
}
```

- Output

```
Class1.Method1 ()
Class2.Method1 ()
```

Tijdens uitvoering van de client zal de aanroep op regel (1) "Class1.Method1()" als resultaat geven, en dit omdat object1 een object is van het type Class1.

Regel (2) zal dan als uitvoer "Method1 defined in Class2" geven, gezien de expressie object2 verwijst naar een object van het type Class2.

Vergeet het keyword `override` niet in de signatuur van de herdefiniërende member, anders bekomt men geen redefinition maar shadowing. Je krijgt ook een warning te zien:

Warning: To make the current member override that implementation, add the override keyword. Otherwise add the new keyword.

De uitleg rond het keyword `'new'` volgt verder (shadowing)

- Code

```
using System;

namespace Example
{
    class Class1
    {
        public virtual string Method1()
        {
            return "Class1.Method1()";
        }
    }

    class Class2 : Class1
    {
        public string Method1()
        {
            return "Class2.Method1()";
        }
    }

    class Program
    {
        static void Main(string[] args)
        {
            Class1 object1 = new Class1();
            Console.WriteLine(object1.Method1());    // 1
            //
            Class1 object2 = new Class2();
            Console.WriteLine(object2.Method1());    // 2
            //
            Console.ReadLine();
        }
    }
}
```

- Output

```
Class1.Method1 ()
Class1.Method1 ()
```

Redefinition houdt dus steeds in dat een andere implementatie wordt gekoppeld aan de interface (lees aanwezigheid) van die member in een bepaald afgeleid type. Het spreekt dan ook voor zich dat enkel implementatiemembers (properties en methods) kunnen geherdefinieerd worden.

Velden bijvoorbeeld die niet beschikken over een implementatie, kunnen bijgevolg ook niet gekoppeld worden aan een andere implementatie, redefinition van velden is dan ook onmogelijk.

Ook Private-members kunnen niet worden geherdefinieerd. Private-members worden overgeërfd maar zijn niet aanroepbaar in afgeleide types, dus moet men geen andere implementatie kunnen koppelen aan die members wanneer die worden aangeroepen op of door een object van een afgeleid type.

Public- en Protected-members zijn wel aanroepbaar in of op objecten van een afgeleid type, dus kan het wel nuttig zijn om aan deze aanroep een andere betekenis (implementatie) te gaan koppelen.

11.2 SEALED

Een overschrijvende (herdefiniërende) members is zelf by default ook overschrijfbaar (herdefinieerbaar). Of met ander woorden override impliceert automatisch herdefinieerbaarheid.

Zo zal in onderstaand voorbeeld de herdefinitie van Method1 zonder argumenten in Class2, ook nog eens gedefinieerd worden in Class3, en dit zonder dat in Class2 Method1 expliciet als "Overridable" stond gemarkeerd.

Wenst men dit default gedrag aan te passen, dan kan men in de signatuur het sealed-keyword aan het override-keyword gaan toevoegen.

In onderstaand voorbeeld zal Method1 (met 1 argument) uit Class2 niet herdefinieerbaar zijn in Class3.

- Code

```
using System;

namespace Example
{
    class Class1
    {
        public virtual string Method1()
        {
            return "Class1.Method1.1()";
        }
        public virtual string Method1(string argument)
        {
            return "Class1.Method1.2()";
        }
    }

    class Class2 : Class1
    {
        public override string Method1()
        {
            return "Class2.Method2.1()";
        }
        public sealed override string Method1(string argument)
        {
            return "Class2.Method2.2()";
        }
        public string Method1(string argument1, string argument2)
        {
            return "Class2.Method2.3()";
        }
    }

    class Class3 : Class2
    {
        public override string Method1()
        {
            return "Class3.Method3.1()";
        }
    }

    class Program
    {
        static void Main(string[] args)
        {
            Class1 object1 = new Class1();
            Console.WriteLine(object1.Method1());
            Console.WriteLine(object1.Method1("test"));
            //
            Class2 object2 = new Class2();
            Console.WriteLine(object2.Method1());
            Console.WriteLine(object2.Method1("test"));
            Console.WriteLine(object2.Method1("test", "test"));
            //
            Class3 object3 = new Class3();
        }
    }
}
```

```

        Console.WriteLine(object3.Method1());
        Console.WriteLine(object3.Method1("test"));
        Console.WriteLine(object3.Method1("test", "test"));
        //
        Class1 object4 = new Class2();
        Console.WriteLine(object4.Method1());
        Console.WriteLine(object4.Method1("test"));
        //
        Class1 object5 = new Class3();
        Console.WriteLine(object5.Method1());
        Console.WriteLine(object5.Method1("test"));
        //
        Class2 object6 = new Class3();
        Console.WriteLine(object6.Method1());
        Console.WriteLine(object6.Method1("test"));
        Console.WriteLine(object6.Method1("test", "test"));
        //
        Console.ReadLine();
    }
}
}

```

- Output

```

Class1.Method1.1 ()
Class1.Method1.2 ()
Class2.Method2.1 ()
Class2.Method2.2 ()
Class2.Method2.3 ()
Class3.Method3.1 ()
Class2.Method2.2 ()
Class2.Method2.3 ()
Class2.Method2.1 ()
Class2.Method2.2 ()
Class3.Method3.1 ()
Class2.Method2.2 ()
Class3.Method3.1 ()
Class2.Method2.2 ()
Class2.Method2.3 ()

```

In Class2 werd een extra versie van Method1 toegevoegd (met 2 argumenten).

Hieronder een context voorbeeld waar geïllustreerd wordt dat het koppelen van een andere betekenis (of implementatie) aan een member nut kan hebben voor afgeleide klassen.

We beschikken over een Counter-klasse, die bruikbaar is om een Value met 1 te incrementeren (Raise) of met 1 te decrementeren (Lower).

- Code

```

using System;

namespace Example
{
    class Counter
    {
        private int _Value;
        public int Value
        {
            private set { _Value = value; }
            get { return _Value; }
        }
        public void Raise()
        {
            Value++;
        }
        public void Lower()
        {
            Value--;
        }
    }

    class Program

```

```

{
    static void Main(string[] args)
    {
        Counter counter1 = new Counter();
        Console.WriteLine(counter1.Value == 0);
        counter1.Raise();
        counter1.Raise();
        counter1.Raise();
        Console.WriteLine(counter1.Value == 3);
        counter1.Lower();
        counter1.Lower();
        Console.WriteLine(counter1.Value == 1);
        //
        Console.ReadLine();
    }
}

```

- Output

```

True
True
True

```

Het schiet ons te binnen dat we ook een soort van Counter zouden kunnen gebruiken waar niet steeds stapwaarde 1 wordt gebruikt om de Value te verhogen of verlagen.

We beslissen hiervoor een type SpecialCounter te creëren.

Het dringt tot ons door dat we SpecialCounter kunnen afleiden van Counter. Een SpecialCounter IS tenslotte een specialisatie van de Counter die we reeds hebben, en SpecialCounter kan de aanwezigheid van alle members uit Counter gebruiken.

Gezien een stapwaarde moet kunnen worden gebruikt door onze SpecialCounter-objecten, beslissen we een StepValue-eigenschap toe te voegen aan deze afgeleide klasse.

Het gedrag (dat we ook overerven van Counter) dat ervoor zorgt dat bij het verhogen (Raise) of verlagen (Lower) steeds de Value met 1 respectievelijk gaat incrementeren of decrementeren, moet echter aangepast worden. De (aanwezigheid van de) Raise- en Lower-methods zijn/is nuttig voor SpecialCounter, enkel het gedrag ervan (of dus de implementaties) zijn voor het gespecialiseerde type niet bruikbaar en moet gewijzigd/geherdefinieerd worden.

- Code

```

using System;

namespace Example
{
    class Counter
    {
        private int _Value;
        public int Value
        {
            protected set { _Value = value; }
            get { return _Value; }
        }
        public virtual void Raise()
        {
            Value++;
        }
        public virtual void Lower()
        {
            Value--;
        }
    }

    class SpecialCounter : Counter
    {
        private int _StepValue = 1;
    }
}

```

```

        public int StepValue
        {
            set { _StepValue = value; }
            get { return _StepValue; }
        }
        public override void Raise()
        {
            Value += StepValue;
        }
        public override void Lower()
        {
            Value -= StepValue;
        }
    }

    class Program
    {
        static void Main(string[] args)
        {
            Counter counter1 = new Counter();
            Console.WriteLine(counter1.Value == 0);
            counter1.Raise();
            counter1.Raise();
            counter1.Raise();
            Console.WriteLine(counter1.Value == 3);
            counter1.Lower();
            counter1.Lower();
            Console.WriteLine(counter1.Value == 1);
            //
            SpecialCounter counter2 = new SpecialCounter();
            Console.WriteLine(counter2.Value == 0);
            counter2.Raise();
            counter2.Raise();
            counter2.Raise();
            Console.WriteLine(counter2.Value == 3);
            counter2.Lower();
            counter2.Lower();
            Console.WriteLine(counter2.Value == 1);
            //
            Console.WriteLine(counter2.StepValue == 1);
            counter2.StepValue = 10;
            Console.WriteLine(counter2.StepValue == 10);
            //
            counter2.Raise();
            counter2.Raise();
            counter2.Raise();
            counter2.Lower();
            Console.WriteLine(counter2.Value == 21);
            //
            Console.ReadLine();
        }
    }
}

```

Hadden we in eerste instantie de Raise- en Lower-methods niet virtual gedefinieerd, dan was het niet mogelijk om het gedrag van deze methods te wijzigen in afgeleide klassen.

Men dient er zich dus steeds bij stil te staan (bij het creëren van een afleidbare basisklasse) of het nuttig kan zijn dat afgeleiden bepaalde implementatiemembers gaan herdefinieren. Is dit nuttig, markeer die member dan meteen als herdefinieerbaar (virtual).

Het zelfde kan gesteld worden voor de "access modifiers" die je gebruikt in de afleidbare basisklassen. Gezien de Raise- en Lower-methods herdefinieerbaar werden gedefinieerd in Counter en gezien deze methods de waarde van het veld _Value (de toestand Value) aanpassen, is het ook logisch de velden (hier veld _Value) die mogelijks gemanipuleerd worden door die herdefinieerbare members beschikbaar te maken voor de afgeleiden.

Bij ingekapselde members kan dit bijvoorbeeld door "access modifier" Protected te gebruiken.

- Output

```
True
True
True
True
True
True
True
True
True
True
```

11.3 TOSTRING()

We weten reeds dat elke klasse op zijn minst een afgeleide is van het algemeen overkoepelend basistype `System.Object`. Wordt er niet expliciet een Inheritance-clausule toegevoegd aan een klasse-definitie, dan wordt er immers op de achtergrond `": System.Object"` aan toegevoegd.

Een afgeleide klasse erft de members over van de basisklasse.

`System.Object` is niet leeg, met andere woorden van `System.Object` zullen bepaalde members worden overgeërfd (<http://msdn.microsoft.com/en-us/library/system.object.aspx>).

Een voorbeeld van zo'n overgeërfde member is: `"public virtual string ToString()"`. Deze member heeft als defaultgedrag dat het de "fully-qualified-identifier" van de klasse (volledige naam van de klasse, inclusief onderdeel waarin die klasse is gedefinieerd) oplevert.

Maar omdat die overgeërfde members als virtual staat gedefinieerd, kunnen we ook dit gedrag aanpassen in afgeleide klassen. De `Student`-klasse bijvoorbeeld die van de `Person` klasse deze virtual string `ToString()` overerft, kan deze method overschrijven of dus koppelen aan een andere implementatie.

Bij het intikken van de code krijgen we onderstaande syntax te zien.

- Code

```
public override string ToString()
{
    return base.ToString();
}
```

We passen dit aan naar ons voorbeeld:

- Code

```
using System;

namespace Example
{
    class Person
    {
        public Person(string name)
        {
            this.Name = name;
        }

        public string Name { get; set; }

        public override string ToString()
        {
            return Name;
        }
    }

    class Student : Person
    {
        public Student(string name, string classgroup)
            : base(name)
        {
            this.ClassGroup = classgroup;
        }
    }
}
```



```

    }

    public string ClassGroup { get; set; }

    public override string ToString()
    {
        return base.ToString() + " (" + ClassGroup + ")";
    }
}

class Program
{
    static void Main(string[] args)
    {
        Person person1 = new Person("John");
        Console.WriteLine(person1.ToString());
        //
        Student student1 = new Student("Jane", "A5");
        Console.WriteLine(student1.ToString());
        //
        Console.ReadLine();
    }
}

```

- Output

```

John
Jane (A5)

```

11.3.1 base

Regelmatig komt het voor dat je in een herdefinitie het te herdefiniëren gedrag eerder wil uitbreiden dan vervangen.

De mogelijkheid bestaat in de implementatie van een herdefiniërende/overschrijvende member (override-member) de te herdefiniëren/overschrijven member (virtual-member) aan te roepen. Men kan dit door het base keyword te laten voorafgaan aan de identifier van de te herdefiniëren/overschrijven member.

De ToString()-implementatie van Student herbruikt hier nu de ToString() van Person, en breidt het resultaat ervan uit om tot zijn eigen resultaat te komen.

11.3.2 Het type "System.Object"

Elk type dat in de .NET runtime bruikbaar is moet een type zijn dat afgeleid is van het type System.Object.

Een value type is een type dat overerft van System.ValueType, wat op zich weer een afgeleide is van System.Object.

De runtime gaat anders om met instanties van value en reference types. Value type instanties worden op de "stack" ge-allocceerd, reference type instanties worden op de "heap" ge-allocceerd.

Het alloceren of ophalen van data uit de stack gaat iets vlugger, alhoewel dit sterk afhankelijk is van het aantal instanties en het gebruikte platform. Dit opzicht zou men als een voordeel van value type kunnen beschouwen.

Om het type-systeem wat robuuster/flexibeler te maken heeft men ervoor gezorgd dat men value type waardes kan stockeren in een variabele/instantie van het reference type 'System.Object' zonder dat men expliciet zo'n 'Object'-instantie moet gaan aanmaken (met bijvoorbeeld 'New Object').

Wanneer een value type waarde wordt toegekend aan een objectvariabele dan blijft het zo dat deze objectvariabele een referentie zal bevatten van een object. Wat gebeurt is dat de toegekende waarde in een object wordt gestockeerd ("boxing") en dat de referentie van dat geboxte object dan wordt toegekend aan die Object-variabele. Bijkomend wordt in dat object het type gemarkeerd van de inhoud. Dit om ervoor te

zorgen dat als de Object-variabele vervolgens gebruikt gaat worden, men weet hoe men van dat geboxte object weer moet overgaan naar een waarde van dat gemarkeerde type ("unboxing").

- Code

```
using System;

namespace Example
{
    class Person
    {
        private string _Name = "John";
        public string Name
        {
            get { return _Name; }
            set { _Name = value; }
        }

        public override string ToString()
        {
            return Name;
        }
    }

    class Counter
    {
        private int _Value = 4;
        public int Value
        {
            get { return _Value; }
            set { _Value = value; }
        }

        public override string ToString()
        {
            return Value.ToString();
        }
    }

    class Program
    {
        static void Main(string[] args)
        {
            object object1 = new object();
            Console.WriteLine(object1);
            //
            object object2 = new object();
            object2 = "abcdef";
            Console.WriteLine(object2);
            //
            object object3 = new Person();
            Console.WriteLine(object3);
            //
            object object4 = new Counter();
            Console.WriteLine(object4);
            //
            Console.ReadLine();
        }
    }
}
```

- Output

```
System.Object
abcdef
John
4
```

Zoals je merkt kan alles in een Object-variabele stockeren.

In bovenstaande Client wordt zowel een String-, Person- als Counter-waarde in een Object-variabele gedeponneerd.

Wat uiteraard erg handig is, bijvoorbeeld in situaties waar men een collectie zou willen beheren van elementen van verschillende entiteiten/datatypes. In zo'n gevallen kan men als datatype voor de elementen gewoon het 'System.Object'-datatype gaan gebruiken.

Boxing treedt op bij het toekennen van een value type waarde aan een variabele van het type System.Object.

Wees u ervan bewust dat toekenningen/assignaties in verschillende vormen kunnen bestaan.

11.4 PROBLEMEN

11.4.1 Method Overloading en Redefinition

De combinatie van redefinition en overloading kan tot verwarrende code leiden. Bestudeer onderstaande code, probeer na te gaan of er compilefouten zouden optreden en zoniet welke de output zou zijn.

- Code

```
using System;

namespace Example
{
    class ClassA
    { }

    class ClassB : ClassA
    { }

    class Class1
    {
        public virtual void Print(ClassA classA)
        {
            Console.WriteLine("C1-CA");
        }
    }

    class Class2 : Class1
    {
        public override void Print(ClassA classA)
        {
            Console.WriteLine("C2-CA");
        }
        public void Print(ClassB classB)
        {
            Console.WriteLine("C2-CB");
        }
    }

    class Program
    {
        static void Main(string[] args)
        {
            ClassB objectB = new ClassB();
            ClassA objectA = objectB;
            //
            Class2 object2 = new Class2();
            Class1 object1 = object2;
            //
            object1.Print(objectA);
            object1.Print(objectB);
            object2.Print(objectA);
            object2.Print(objectB);
            //
            Console.ReadLine();
        }
    }
}
```

- Output

```
C2-CA  
C2-CA  
C2-CA  
C2-CB
```

Bovenstaand voorbeeld bevat geen compile-fouten.

In een statisch getypeerde programmeertaal bepaalt de compiler wat je kan doen met een expressie (van een bepaald type). De compiler bepaalt dus ook welke method er wordt aangeroepen. Hiervoor gebruikt de compiler het statische type van de objectexpressie waarop deze method wordt aangeroepen, de compiler kent immers enkel dit statische type en niet het type van een object at runtime.

Het statische type van de ontvangerexpressie 'object1' is 'Class1', waardoor het bij de eerste twee calls over de eerste 'Print' (met argument van type 'ClassA') gaat. Omdat hier deze eerste method 'Print' geherdefinieerd wordt (redefinition), zal pas at runtime beslist worden welke implementatie "C1-CA" of "C2-CA" hieraan gebonden wordt. Het dynamisch type (type at runtime) van de ontvangerexpressie 'object1' is 'Class2', waardoor de implementatie uit 'Class2' namelijk "C2-CA" hieraan gebonden wordt.

Iets moeilijker is het bij de calls 'object2.Print', waar het statisch type van 'object2' 'Class2' is, en er in het type 'Class2' twee 'Print' methods aanwezig zijn (overloading). Het blijft zo dat de compiler bepaalt welke method er wordt aangeroepen. Bij overloading gaat de compiler dit baseren op het statisch type van de argumentwaarde, opnieuw is dit het enigste type dat de compiler kent.

Het statisch type van argumentwaarde 'objectB' in de call 'object2.Print(objectB)' is 'ClassB', waardoor de tweede 'Print' door de compiler wordt uitgekozen, en hier aan deze call de implementatie "C2-CB" wordt gebonden.

Het statisch type van argumentwaarde 'objectA' in de call 'object2.Print(objectA)' is 'ClassA', waardoor de eerste 'Print' door de compiler wordt uitgekozen. Omdat hier deze eerste method 'Print' geherdefinieerd wordt (redefinition), zal pas at runtime beslist worden welke implementatie "C1-CA" of "C2-CA" hieraan gebonden wordt. Het dynamisch type (type at runtime) van de ontvangerexpressie 'object2' is 'Class2', waardoor implementatie uit 'Class2' namelijk "C2-CA" hieraan gebonden wordt.

Je ziet hoe moeilijk leesbaar bovenstaande client-code is. Men dient rekening te houden met het STE door de compiler gebruikt (in de gevallen van overloading) en het DTE gebruikt voor de binding.

Overloading brengt meestal niets bij aan de uitdruktingskracht van de taal, al te vaak zorgt het voor verwarrende of op zijn minst moeilijk leesbare code.

Het gebruik ervan valt dan ook doorgaans af te raden.

11.4.2 Shadowing

De compiler bepaald welke member wordt aangeroepen, het dynamisch type van de objectexpressie bepaald vervolgens (at runtime) welke implementatie gebonden wordt aan deze call.

- Code

```
using System;  
  
namespace Example  
{  
    class Class1  
    {  
        public virtual void Method1()  
        {  
            Console.WriteLine("(1)");  
        }  
    }  
  
    class Class2 : Class1
```

```

{
    public override void Method1()
    {
        Console.WriteLine("(2)");
    }
}

class Class3 : Class2
{
    public override void Method1()
    {
        Console.WriteLine("(3)");
    }
}

class Program
{
    static void Main(string[] args)
    {
        Class1 object_1_1 = new Class1();
        Class1 object_1_2 = new Class2();
        Class1 object_1_3 = new Class3();
        Class2 object_2_2 = new Class2();
        Class2 object_2_3 = new Class3();
        Class3 object_3_3 = new Class3();
        //
        object_1_1.Method1();
        object_1_2.Method1();
        object_1_3.Method1();
        object_2_2.Method1();
        object_2_3.Method1();
        object_3_3.Method1();
        //
        Console.ReadLine();
    }
}

```

- Output

```

(1)
(2)
(3)
(2)
(3)
(3)

```

Wanneer shadowing gebruikt wordt (in tegenstelling tot redefinition), worden verdere herdefinities van de overschaduwde member vanaf de overschaduwende klasse verhinderd.

Bijkomend ontstaat er ook een nieuwe member met dezelfde identifier.

- Code

```

using System;

namespace Example
{
    class Class1
    {
        public virtual void Method1()
        {
            Console.WriteLine("(1)");
        }
    }

    class Class2 : Class1
    {
        public new virtual void Method1()
        {
            Console.WriteLine("(2)");
        }
    }
}

```

```

class Class3 : Class2
{
    public override void Method1()
    {
        Console.WriteLine("(3)");
    }
}

class Program
{
    static void Main(string[] args)
    {
        Class1 object_1_1 = new Class1();
        Class1 object_1_2 = new Class2();
        Class1 object_1_3 = new Class3();
        Class2 object_2_2 = new Class2();
        Class2 object_2_3 = new Class3();
        Class3 object_3_3 = new Class3();
        //
        object_1_1.Method1();
        object_1_2.Method1();
        object_1_3.Method1();
        object_2_2.Method1();
        object_2_3.Method1();
        object_3_3.Method1();
        //
        Console.ReadLine();
    }
}

```

Er is hier sprake van 2 verschillende members met de identifier Method1. De eerste Method1 gedefinieerd in Class1 wordt overgeërfd in Class2 en Class3. De tweede Method1 is gedefinieerd vanaf Class2 deze wordt overgeërfd en geherdefinieerd in Class3.

Het is alsof een nieuwe member vanaf Class2 werd toegevoegd, alleen is het hier zo dat deze nieuwe member dezelfde identifier heeft als een overgeërfde member. Naast de gemeenschappelijke naam hebben deze 2 verschillende Method1 niets met elkaar te maken.

Het is echter wel zo dat vanaf Class2 de eerste Method1 niet meer kan worden geherdefinieerd.

De compiler zal op basis van het statisch type van de ontvangerexpressie bepalen welke Method1 wordt aangeroepen. Hiervoor verkiest hij de meest recent gedefinieerde Method1.

Op ontvangerexpressie van statisch type Class1 kan zowiezo slechts 1 Method1 (de eerste) worden aangeroepen. Wanneer Method1 op ontvangerexpressies van statisch types Class2 of Class3 wordt aangeroepen dan beschouwd de compiler dit als een call naar de tweede Method1.

- Output

```

(1)
(1)
(1)
(2)
(3)
(3)

```

De regels omtrent shadowing zijn ingewikkeld en vaak verschillend.

Verschillende OOP's gebruiken verschillende variaties van shadowing, bijvoorbeeld "shadowing by name", "shadowing by signature", ...

Vele programmeurs hebben moeite met het concept van shadowing. Allen argumenteren om shadowing in uw eigen klassen beter te vermijden. Alleen wanneer het echt noodzakelijk is, raad ik aan shadowing te gebruiken.

11.4.3 Verschillen tussen Overloading, Redefinition of Shadowing

Stel dat zowel Class1 als Class2 (die overerft van Class1) een member met de signatuur "public void Test()" definiëren.

- Code

```
using System;

namespace Example
{
    class Class1
    {
        public void Test()
        {
            Console.WriteLine("(1)");
        }
    }

    class Class2 : Class1
    {
        public void Test()
        {
            Console.WriteLine("(2)");
        }
    }

    class Program
    {
        static void Main(string[] args)
        {
            Class1 object1 = new Class1();
            object1.Test();
            //
            Class1 object2 = new Class2();
            object2.Test();
            //
            Class2 object3 = new Class2();
            object3.Test();
            //
            Console.ReadLine();
        }
    }
}
```

- Output

```
(1)
(1)
(2)
```

Dan zie je hoe shadowing wordt toegepast, ondanks dat we dit niet expliciet hebben opgegeven (geen gebruik van het keyword "new virtual").

De compiler geeft wel een warning dat shadowing zal worden gebruikt.

Stel dat Class1 een "public virtual void Test()" definieert en afgeleide Class2 hier een "public void Test()" aan toevoegt.

- Code

```
using System;

namespace Example
{
    class Class1
    {
        public virtual void Test()
        {
            Console.WriteLine("(1)");
        }
    }
}
```

```

    }

    class Class2 : Class1
    {
        public void Test()
        {
            Console.WriteLine("(2)");
        }
    }

    class Program
    {
        static void Main(string[] args)
        {
            Class1 object1 = new Class1();
            object1.Test();
            //
            Class1 object2 = new Class2();
            object2.Test();
            //
            Class2 object3 = new Class2();
            object3.Test();
            //
            Console.ReadLine();
        }
    }
}

```

- Output

```

(1)
(1)
(2)

```

Ook hier zal shadowing worden toegepast zonder dat we dit expliciet aangeven.

We krijgen opnieuw een warning die ons waarschuwt dat shadowing zal worden toegepast.

Stel basisklasse Class1 met een "public virtual void Test()" waarvan afgeleide klasse een "public override void Test()" toevoegt.

- Code

```

using System;

namespace Example
{
    class Class1
    {
        public virtual void Test()
        {
            Console.WriteLine("(1)");
        }
    }

    class Class2 : Class1
    {
        public override void Test()
        {
            Console.WriteLine("(2)");
        }
    }

    class Program
    {
        static void Main(string[] args)
        {
            Class1 object1 = new Class1();
            object1.Test();
            //
            Class1 object2 = new Class2();
            object2.Test();
            //
        }
    }
}

```



```

        Class2 object3 = new Class2();
        object3.Test();
        //
        Console.ReadLine();
    }
}

```

- Output

```

(1)
(2)
(2)

```

Bovenstaand voorbeeld maakt op correcte wijze gebruik van redefinition.

Stel klasse Class1 met "public void Test()" en een afgeleide klasse Class2 met een "public new virtual void Test()".

- Code

```

using System;

namespace Example
{
    class Class1
    {
        public void Test()
        {
            Console.WriteLine("(1)");
        }
    }

    class Class2 : Class1
    {
        public new virtual void Test()
        {
            Console.WriteLine("(2)");
        }
    }

    class Program
    {
        static void Main(string[] args)
        {
            Class1 object1 = new Class1();
            object1.Test();
            //
            Class1 object2 = new Class2();
            object2.Test();
            //
            Class2 object3 = new Class2();
            object3.Test();
            //
            Console.ReadLine();
        }
    }
}

```

- Output

```

(1)
(1)
(2)

```

Bovenstaand voorbeeld maakt op gebruik van shadowing, hier wordt dit ook expliciet aangegeven via het keyword Shadows.

11.5 OEFENINGEN

11.5.1 Oefening 1

Zorg dat we objecten voor "employees" kunnen creëren. Een employee heeft een bepaald maandloon en een jaarloon (dat 12 keer het maandloon is)

Zorg er ook voor dat we objecten voor "managers" kunnen creëren. Een manager is een employee, met eigenschappen maandloon, premie en jaarloon.

Het jaarloon van een manager wordt net zo berekend als van een gewone employee, met als verschil dat daar nog de premie wordt bij opgeteld.

11.5.2 Oefening 2

Creëer de nodige klasse om met onderstaande clientcode onderstaand programmaverloop te bekomen.

- Code

```
class Program
{
    static void Main(string[] args)
    {
        Addition addition1 = new Addition(3, 4);
        Console.WriteLine(addition1.GetSum());
        Console.WriteLine(addition1.ToString());
        //
        addition1.Operand1 = 5;
        addition1.Operand2 = 6;
        Console.WriteLine(addition1.GetSum());
        Console.WriteLine(addition1.ToString());
        //
        Console.ReadLine();
    }
}
```

- Output

```
7
3 + 4 = 7
11
5 + 6 = 11
```

11.5.3 Oefening 3

Creëer de nodige klassen om met onderstaande clientcode onderstaande programmaverloop te bekomen.

- Code

```
class Program
{
    static void Main(string[] args)
    {
        Airport airport1 = new Airport("ZAV", "Brussels");
        Airport airport2 = new Airport("NYK", "New York Kennedy");
        //
        Flight flight1 = new Flight(airport1, airport2);
        Flight flight2 = new Flight(airport2, airport1);
        //
        Holiday holiday1 = new Holiday();
        holiday1.Add(flight1);
        holiday1.Add(flight2);
        //
        for (int index = 0; index < holiday1.Count - 1; index++)
        {
            Console.WriteLine(holiday1.Item(index).ToString());
        }
        //
        Console.ReadLine();
    }
}
```

- Output

```
ZAV: Brussels -> NYK: New York Kennedy  
NYK: New York Kennedy -> ZAV: Brussels
```



"You should check your e-mails more often. I fired you over three weeks ago."

12 STATIC TYPING

C#.NET is net zoals de meest voor de hand liggende object georiënteerde talen een statisch getypeerde programmeertaal.

Dit houdt in dat elke expressie van een bepaald datatype is. Het datatype kan doorgaans worden afgeleid (door de compiler) op basis van de bijhorende declaratie of definitie.

Een expressie is een uitdrukking die een bepaalde "waarde" aanduidt. Dit kan in allerlei vormen zijn: literals, variabelen (inclusief datavelden en argumenten), voorwaardes (Booleaanse expressies), rekenkundige bewerkingen, member-calls, object initializers, ...

Naast statisch getypeerde programmeertalen bestaan er ook dynamisch getypeerde programmeertalen (ook wel ongetypeerde programmeertalen genoemd). Daar wordt dus niet statisch (at compiletime), maar at runtime nagegaan of een ontvangerexpressie (objectexpressie) wel degelijk de boodschap verstaat die erop wordt aangeroepen.

Dit heeft als gevolg dat er at runtime fouten kunnen optreden (dus tijdens uitvoering) wanneer bijvoorbeeld die ontvangerexpressie de boodschap niet verstaat. Dit in tegenstelling tot statisch getypeerde programmeertalen waar reeds at compiletime (dus nog voor het uitvoeren) deze fout is opgetreden.

Runtime-fouten leiden doorgaans tot programma-afbreking en zouden bijgevolg beter vermeden worden. Bij een compilefout kan niet worden gebuild en kan er geen uitvoerbaar geheel gecreëerd worden, waarin er bijgevolg ook geen fouten (veroorzaakt door bovenvermelde omstandigheid) kunnen zitten.

13 DYNAMIC BINDING

We weten dat we op een objectexpressie alle publieke members kunnen aanroepen uit het bijhorende STE (statisch type van de expressie).

Tot nu toe wees een expressie van een bepaald statisch type ook steeds (at runtime) naar een object van datzelfde type (zoals (person1) en (person2)).

Toch hoeft dit niet altijd zo te zijn, een variabele gedeclareerd van een basistype kan ook opgevuld zijn met een referentie van een object van een afgeleid type (zoals (person3)).

Het "eigenlijke" type van object waar naartoe een objectexpressie wijst at runtime (tijdens uitvoering) wordt ook wel het dynamisch type van een expressie genoemd (hier verder afgekort als DTE).

Het STE en DTE hoeft dus niet overeen te komen, weliswaar moet het DTE op zijn minst een afgeleide zijn van het STE.

- Code

```
using System;

namespace Example
{
    class Person
    { }

    class Student : Person
    { }

    class Program
    {
        static void Main(string[] args)
        {
            // person1: STE (Person) = DTE (Person)
            Person person1 = new Person();
            // person2: STE (Student) = DTE (Student)
            Student person2 = new Student();
            // person3: STE (Person) <> DTE (Student)
            Person person3 = new Student();
        }
    }
}
```

De motivatie om een referentie van een object van een afgeleid type te stockeren in een variabele gedeclareerd van het basistype is vooral de opgeleverde flexibiliteit. Alle objecten (zowel Person- als Student-objecten bijvoorbeeld) kunnen op dezelfde manier behandeld worden.

Of het DTE nu 'Person' of 'Student' is, steeds kan de verwijzing naar zo'n object in een variabele van het type 'Person' (STE 'Person') gestockeerd worden.

Je zou dus kunnen zeggen dat een Student-object (DTE Student) zich in de vorm van een Student of Person kan bevinden, en dit afhankelijk van welke STE en ook dus interface moet worden gebruikt.

13.1 LISKOV SUBSTITUTION PRINCIPLE

Uit vorige voorbeelden weten we reeds dat het STE van belang is voor het bepalen (door de compiler) welke boodschappen op een objectexpressie kunnen worden aangeroepen.

Dat de compiler hiervoor werkt met het STE is logisch gezien at compiletime (en dus voor runtime) het DTE nog niet gekend is.

Het aanroepen van een member uit de interface van het basistype op een objectexpressie met het basistype als STE maar een afgeleid type als DTE, zal nooit problemen opleveren, gezien dat afgeleid object ook altijd zal beschikken over zo'n member. Deze member is minstens overgeërfd, en al dan niet geherdefinieerd door de afgeleide klasse.

Sterker nog, men kan zeggen: overal waar men een instantie van de basisklasse kan gebruiken kan men deze instantie vervangen door een instantie van de afgeleide klasse (Liskov Substitution Principle).

13.2 DYNAMIC BINDING

VisualBasic.net gebruik net zoals alle OOP's dynamic binding.

De kracht van inheritance is niet zozeer het herbruik van members met hun eventuele bijhorende implementaties, maar eerder dynamisch polymorfisme.

Polymorfisme houdt letterlijk in: het kunnen aannemen van vele ("poly") verschillende vormen ("morph").

In onderstaand voorbeeld hebben we een Counter-klasse waar men de Value kan van opvragen, en waar men de Value kan doen toenemen met 1 (via de Raise()-method). Een specialisatie-klasse SpecialCounter wordt ook voorzien, waar men bijkomend een stap-waarde (StepValue) kan opgeven om niet steeds de Value met 1 te hoeven verhogen.

- Code

```
using System;
namespace Example
{
    class Counter
    {
        public Counter(int value)
        {
            this._Value = value;
        }

        protected int _Value;
        public int Value
        {
            get { return _Value; }
        }

        public virtual void Raise()
        {
            _Value++;
        }
    }

    class SpecialCounter : Counter
    {
        public SpecialCounter(int value, int stepValue): base(value)
        {
            this._stepValue = stepValue;
        }

        protected int _stepValue;
        public int stepValue
        {
            get { return _stepValue; }
        }

        public override void Raise()
        {
            _Value += stepValue;
        }
    }
}
```

```

    }

    class Program
    {
        static void Main(string[] args)
        {
            Counter counter1 = new Counter(10);
            Console.WriteLine(counter1.Value);
            counter1.Raise();
            Console.WriteLine(counter1.Value);
            //
            SpecialCounter counter2 = new SpecialCounter(20,5);
            Console.WriteLine(counter2.Value);
            counter2.Raise();
            Console.WriteLine(counter2.Value);
            //
            Console.ReadLine();
        }
    }
}

```

- Output

```

10
11
20
25

```

Het statisch type van de expressie `counter2` is `Counter`, het dynamisch type van deze expressie is echter `SpecialCounter`. We zien dat de method `Raise()` wordt aangeroepen op de ontvanger-expressie `counter2`, dit is mogelijk gezien het STE van die ontvanger (`Counter`) in zijn interface een method `Raise()` voorziet.

We zien ook dat die call naar de `Raise()`-method uit `Counter` gebonden werd aan de (virtual) implementatie gedefinieerd in het afgeleide type `SpecialCounter`. De implementatie gedefinieerd in het DTE `counter2` (zijnde `SpecialCounter`) werd hier uitgevoerd.

Welke members men kan aanroepen op een ontvangerexpressie is afhankelijk van het STE, dit wordt door de compiler gecontroleerd om typefouten te vermijden.

Aan welke implementatie een membercall op een ontvangerexpressie gebonden wordt is afhankelijk van het DTE, dat at runtime wordt bepaald.

C#.NET gebruikt dus in bepaalde gevallen dynamische binding.

Hierbij kan dus slechts tijdens uitvoering van het programma bepaald worden hoe een bepaalde member-call moet worden vertaald in een sprong naar de juiste implementatie. Dit gebeurt slechts tijdens uitvoering omdat pas dan het dynamische type van een expressie kan worden bepaald, gezien deze kan variëren tijdens runtime.

Een performantere vorm van binding is statisch binding. Hierbij wordt tijdens het compileren reeds deze vertaling gemaakt, en hoeft er dus (tijdens uitvoering) geen tijd aan verloren te gaan.

Om de performantie te bevorderen gebruikt C#.NET niet altijd dynamisch binding. Een call naar een `Private` member bijvoorbeeld kan steeds statisch gebonden worden. Een `Private` member kan nooit geherdefinieerd worden, gezien het enkel in de klasse zelf toegankelijk is.

Het type van een expressie tijdens uitvoering kan in zo'n geval dus nooit een criterium vormen voor het bepalen welke klasse de implementatie bevat die moet worden uitgevoerd. Dus meteen (at compiletime) kan deze membercall reeds vertaald worden in een sprong naar de implementatie in de klasse zelf.

Dynamische binding is in zo'n situatie dus niet nodig. Ook andere gevallen waarin men afwijkt van dynamische binding bestaan, deze - net zoals de mogelijkheid om expliciet aangegeven af te wijken van dynamische binding - worden verderop behandeld.

Sommige programmeertalen beschikken enkel over statische binding, dit zijn per definitie geen objectgeoriënteerde talen. Want dan wordt het DTE (of het type van een expressie tijdens uitvoering) nooit gebruikt om te bepalen welke implementatie moet worden uitgevoerd, en kan er dus bijgevolg nooit sprake zijn van bijvoorbeeld redefinition, of dus dynamisch polymorfisme.

Dynamisch polymorfisme is immers een van de basiskenmerken van een OOP.



14 STATIC MEMBERS

14.1 STATIC METHOD

Alle members die we tot nu toe in klassen hadden gedefinieerd waren instancemembers: ofwel velden die een toestand bevatten voor een bepaalde instantie (object) van een bepaalde klasse, ofwel implementatiemembers waarbij de implementatie op een bepaalde instantie (object) worden uitgevoerd.

Een nieuw soort van members die hier wordt behandeld zijn de "class members", ook wel "static members" of in de context van C#.NET "shared members" genoemd.

Deze class members zijn niet zozeer aan een bepaald object gerelateerd zoals instancemembers, maar eerder aan de klasse ("CLASS members") zelf gerelateerd. Zo'n members worden gedeeld ("STATIC members") door alle instanties/objecten van die klasse waarin die members zijn gedefinieerd.

- Code

```
using System;

namespace Example
{
    class Area
    {
        public static double Rectangle(int hoogte, int breedte)
        {
            return hoogte * breedte;
        }

        public static double Circle(int radius)
        {
            return radius * radius * Math.PI;
        }
    }

    class Program
    {
        static void Main(string[] args)
        {
            Console.WriteLine(Area.Rectangle(3, 4));
            Console.WriteLine(Area.Circle(1));
            // Console.ReadLine();
        }
    }
}
```

- Output

```
12
3,14159265358979
```

In bovenstaande klasse zijn alle members "class/static members". Bij implementatiemembers wordt de access modifier Static toegevoegd aan de overige signatuur van die member-definitie om dit te bekomen.

Deze access modifier (static) is als enigste combineerbaar met andere access modifiers (public, protected, private, ...). Deze access modifier is - in tegenstelling tot de andere access modifiers - enkel van toepassing op members van een klasse.

Een class member kan men aanspreken net zoals een instancemember door de puntnotatie te gebruiken maar deze keer verwijzen naar de klasse zelf in plaats van naar een instantie/object van die klasse.

Men verwijst dus op volgende wijze naar een class-member: class-identifier.class-member-identifier (argumentslist)

In een client van klasse Area is het mogelijk maar onzinnig een object te instantiëren van deze klasse. Geen enkele member is object-gerelateerd, dus een object van deze klasse zou toch geen object-specifieke toestand hebben.

Het is mogelijk een classmember aan te roepen via een objectexpressie.

Toch valt dit voor de leesbaarheid af te raden, men krijgt trouwens een warning (maar geen error) wanneer men dit probeert.

14.1.1 Constanten als Class Members

Een constante is een onwijzigbare "dataholder" waarvan de inhoud reeds at compiletime is gekend.

Bij een Const-declaration moet je immers steeds een toekenningsclausule toevoegen. De toegekende waarde uit deze toekenningsclausule moet overigens steeds aan de hand van een constante expressie geformuleerd worden.

Als je een constante als member toevoegt aan een klasse, zullen alle instanties van deze klasse dezelfde waarde (die gekend is at compiletime) gebruiken.

Een constante als member is dan ook steeds een class/shared member, zonder dat men dit expliciet in de declaratie kan/moet vermelden.

- Code

```
using System;
namespace Example
{
    class Area
    {
        public static double Rectangle(int hoogte, int breedte)
        {
            return hoogte * breedte;
        }

        public static double Circle(int radius)
        {
            return radius * radius * PI;
        }

        private const double PI = Math.PI;
    }

    class Program
    {
        static void Main(string[] args)
        {
            Console.WriteLine(Area.Rectangle(3, 4));
            Console.WriteLine(Area.Circle(1));
            // Console.ReadLine();
        }
    }
}
```

- Output

```
12
3,14159265358979
```

Vergeet ook niet bij constante members van een klasse een gepast access modifier uit te kiezen.

14.2 STATIC FIELD

Naast Shared methods, properties en constanten kan men ook velden als class/shared member declareren, deze worden ook wel "class variables" genoemd.

Deze velden bevatten een toestandswaarde die door alle instanties worden gedeeld, deze waarde zit dan ook slecht één maal in het geheugen.

- Code

```
using System;

namespace Example
{
    class Test
    {
        public Test(int instanceValue, int classValue)
        {
            m_InstanceValue = instanceValue;
            m_ClassValue = classValue;
        }

        private int m_InstanceValue;
        private static int m_ClassValue;

        public override string ToString()
        {
            return "InstanceValue: " + m_InstanceValue + " / " +
                "ClassValue: " + m_ClassValue;
        }
    }

    class Program
    {
        static void Main(string[] args)
        {
            Test object1 = new Test(1,10);
            Console.WriteLine(object1.ToString());
            //
            Test object2 = new Test(2, 20);
            Console.WriteLine(object2.ToString());
            //
            Console.WriteLine(object1.ToString());
            //
            Console.ReadLine();
        }
    }
}
```

- Output

```
InstanceValue: 1 / ClassValue: 10
InstanceValue: 2 / ClassValue: 20
InstanceValue: 1 / ClassValue: 20
```

Zoals je merkt wordt de waarde uit het veld `_ClassValue` gedeeld door alle instanties van de klasse `Class`.

Het is geen enkel probleem (zoals hierboven ook gedemonstreerd) om een class member aan te roepen/te gebruiken in de implementatie van een instance member.

Het omgekeerde (een instance member aanroepen/gebruiken in de implementatie van een class member) kan echter niet. Een class member werkt immers onafhankelijk van een instantie, waardoor men geen idee zou hebben waarop (op welke instantie/object) de aanroep naar de instance member moet worden uitgevoerd.

Een expressie als Me.class-member-identifier is mogelijk maar onzinnig, men krijgt hierbij een warning (maar geen error).

14.3 STATIC CONSTRUCTOR

Ook voor constructoren kan de Static access modifier worden gebruikt.

Zo'n gedeelde constructoren worden eenmalig uitgevoerd, en dit de eerste keer dat een member wordt gebruikt uit de klasse waarin die constructor is gedefinieerd.

Dit eerste gebruikt van een member kan een impliciete call zijn naar een constructor (zoals bij het gebruik van een objectinstializer (1)), kan een expliciet call zijn naar een constructor (zoals bijvoorbeeld bij het aanroepen van een constructor van de basisklasse) of kan het gebruikt zijn van een classmember.

- Code

```
using System;

namespace Example
{
    class Class1
    {
        static Class1()
        {
            Console.WriteLine("Class1: static Class1()");
        }
        public Class1()
        {
            Console.WriteLine("Class1: public Class1()");
        }
    }

    class Program
    {
        static void Main(string[] args)
        {
            Class1 object1 = new Class1();
            //
            Console.ReadLine();
        }
    }
}
```

- Output

```
Class1: static Class1 ()
Class1: public Class1 ()
```

- Code

```
using System;

namespace Example
{
    class Class1
    {
        static Class1()
        {
            Console.WriteLine("Class1: static Class1()");
        }
        public Class1()
        {
            Console.WriteLine("Class1: public Class1()");
        }
    }

    class Class2 : Class1
    {
        static Class2()
        {
        }
    }
}
```

```

    {
        Console.WriteLine("Class2: static Class2()");
    }
    public Class2()
    {
        Console.WriteLine("Class2: public Class2()");
    }
}

class Program
{
    static void Main(string[] args)
    {
        Class2 object2 = new Class2();
        //
        Console.ReadLine();
    }
}

```

- Output

```

Class2: static Class2()
Class1: static Class1()
Class1: public Class1()
Class2: public Class2()

```

Bemerkt dat de static constructoren hier geen access modifiers als Public, Protected of Private hebben. Een shared constructor wordt immers steeds impliciet aangeroepen, in tegenstelling tot "gewone" constructoren die ook expliciet aangeroepen kunnen worden in clients als die constructor Public is of in clients en afgeleide klassen als die constructoren Protected is.

Gezien static constructoren steeds impliciet worden aangeroepen, zijn deze steeds parameterloos.

14.4 OEFENINGEN

14.4.1 Oefening 1

Ontwerp de vereiste klasse Calculation die 2 argumenten verwacht. Bij het aanmaken van een object krijgen we via de method GetSum het resultaat van de som, en bij GetProduct het resultaat van het product op ons scherm.

14.4.2 Oefening 2

Ontwerp de vereiste klasse, om onderstaande clientcode correct te kunnen uitvoeren

- Code

```

class Program
{
    static void Main(string[] args)
    {
        Rechthoek rechthoek1 = new Rechthoek(20, 30);
        Rechthoek rechthoek2 = new Rechthoek(10, 20);
        //
        Console.WriteLine(rechthoek1.Oppervlakte());
        Console.WriteLine(rechthoek2.Oppervlakte());
        //
        Console.WriteLine(rechthoek1.IsGroterDan(rechthoek2) == true);
        Console.WriteLine(rechthoek2.IsGroterDan(rechthoek1) == false);
        //
        Console.WriteLine(Rechthoek.IsGrootste(rechthoek1, rechthoek2) == rechthoek1);
        //
        Console.ReadLine();
    }
}

```

"IsGrootste" wordt gebaseerd op de oppervlakte. Zorg dat de logica voor het bepalen van de grootste nergens overbodig herhaald wordt.



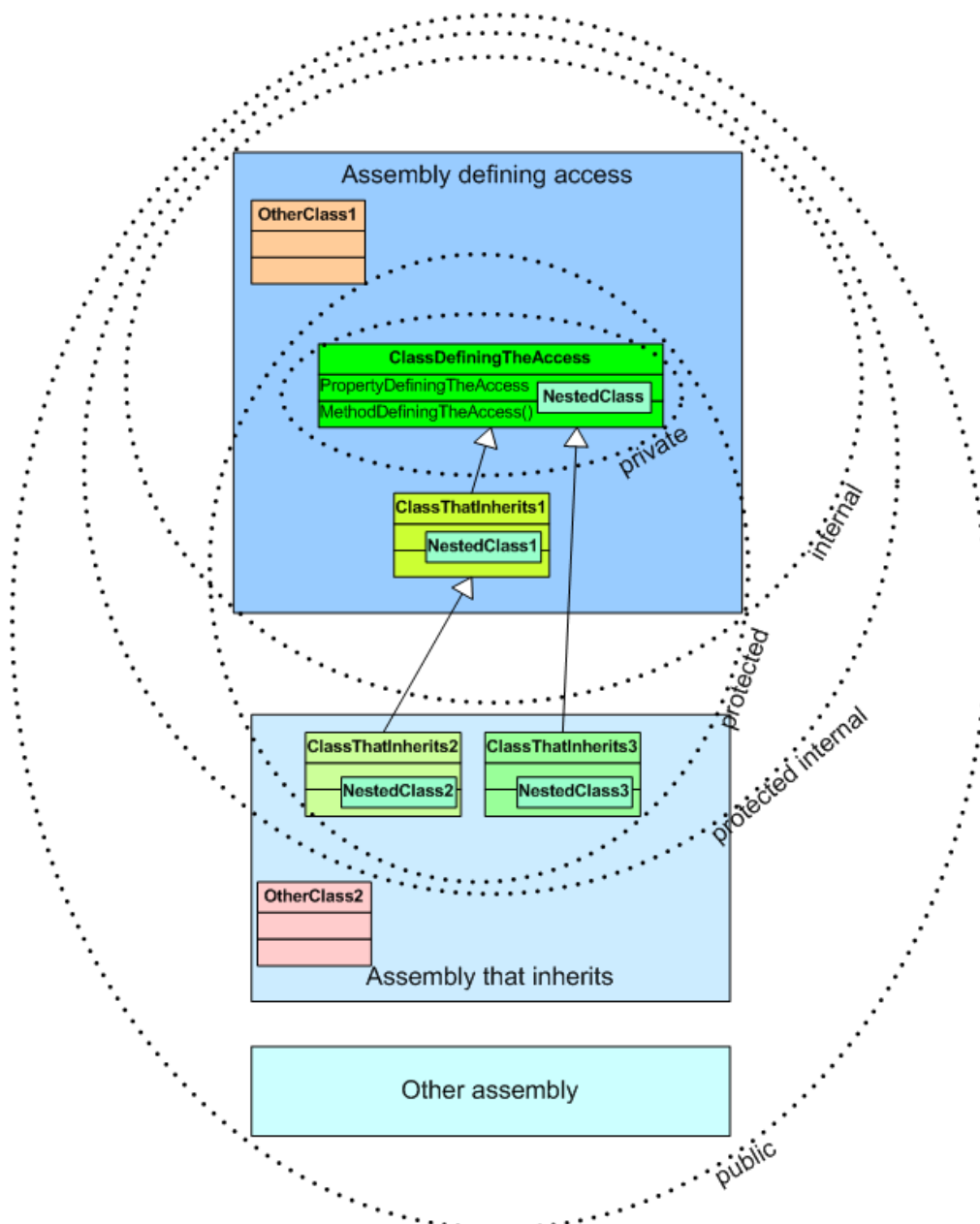
15 ACCESS MODIFIERS

15.1 DEFINITION

An access modifier is a keyword you can add to your declaration of a class or class member to allow or restrict their accessibility.

In de eerdere voorbeelden zijn de instructies `private` en `public` u wellicht opgevallen. Deze sleutelwoorden beperken de toegankelijkheid tot een klasse en haar methoden. We noemen dit access modifiers. Dit zijn de instructies `public`, `private`, `protected`, `internal` en `protected internal` die we voor de klasse en methode-declaratie plaatsen. Hiermee kunnen we ervoor zorgen dat bepaalde methoden of klassen niet zonder meer gebruikt kunnen worden. In de tabel ziet u een overzicht van deze access modifiers.

De meest gebruikt access modifiers zijn `public` en `private`. Hiermee kunnen we immers de externe toegankelijkheid van variabelen en methoden in een klasse bepalen.



- Code

```
internal class Class1
{
    private int Member1;
    protected int Member2;
    internal int Member3;
    protected internal int Member4;

    private class NestedClass
    {
        void Test(Class1 foo)
        {
            foo.Member1 = 0; //OK
            foo.Member2 = 0; //OK
            foo.Member3 = 0; //OK
            foo.Member4 = 0; //OK
        }
    }
}

public class Class2
{
    private void Method1()
    {
    }

    private int _Member1;

    protected int Member1
    {
        get
        {
            return _Member1;
        }
        set
        {
            Method1();
            _Member1 = value;
        }
    }

    internal int Member2;
    public int Member3;
    protected internal int Member4;
}
```

The above access modifiers prevent direct access as seen by the following access attempts. These attempts produce compilation errors:

From the same assembly as the above classes:

- Code

```
class Test
{
    void Test1()
    {
        Class1 c1 = new Class1(); //OK
        c1.Member1 = 0; //Compilation error here
        c1.Member2 = 0; //Compilation error here
        c1.Member3 = 0; //OK
        c1.Member4 = 0; //OK
    }
}

internal class DerivedClass : Class1
{
    private void Test2()
    {
        Member1 = 0; //Compilation error here
        Member2 = 0; //OK
        Member3 = 0; //OK
    }
}
```



```

        Member4 = 0; //OK
    }

    private class NestedClass
    {
        void Test(DerivedClass foo)
        {
            foo.Member1 = 0; //Compilation error here
            foo.Member2 = 0; //OK
            foo.Member3 = 0; //OK
            foo.Member4 = 0; //OK
        }
        void Test(Class1 foo)
        {
            foo.Member1 = 0; //Compilation error here
            foo.Member2 = 0; //Compilation error here
            foo.Member3 = 0; //OK
            foo.Member4 = 0; //OK
        }
    }
}

```

From a different assembly from the above classes:

- Code

```

public class Test
{
    void Test1()
    {
        Class1 c1 = new Class1(); //Compilation error here

        Class2 c2 = new Class2(); //OK
        c2.Method1(); //Compilation error here
        c2._Member1 = 0; //Compilation error here
        c2.Member1 = 0; //Compilation error here
        c2.Member2 = 0; //Compilation error here
        c2.Member3 = 0; //OK
        c2.Member4 = 0; //Compilation error here
    }
}

internal class DerivedClass : Class2
{
    private void Test2()
    {
        Member1 = 0; //OK
        Member2 = 0; //Compilation error here
        Member3 = 0; //OK
        Member4 = 0; //OK
    }
    private class NestedClass
    {
        void Test(DerivedClass foo)
        {
            foo.Member1 = 0; //Compilation error here
            foo.Member2 = 0; //Compilation error here
            foo.Member3 = 0; //OK
            foo.Member4 = 0; //OK
        }
        void Test(Class2 foo)
        {
            foo.Member1 = 0; //Compilation error here
            foo.Member2 = 0; //Compilation error here
            foo.Member3 = 0; //OK
            foo.Member4 = 0; //Compilation error here
        }
    }
}

```

15.2 OVERVIEW

In C# there are 5 different types of Access Modifiers.

- public

There are no restrictions on accessing public members.

- private

Access is limited to within the class definition. This is the default access modifier type if none is formally specified

- protected

Access is limited to within the class definition and any class that inherits from the class

- internal

Access is limited exclusively to classes defined within the current project assembly

- protected internal

Access is limited to the current assembly and types derived from the containing class. All members in current project and all members in derived class can access the variables.

15.2.1 Public

The public keyword is an access modifier for types and type members. Public access is the most permissive access level.

There are no restrictions on accessing public members.

Accessibility:

- Can be accessed by objects of the class
- Can be accessed by derived classes

In the following example num1 is direct access.

- Code

```
using System;

namespace AccessModifiers
{
    class Program
    {
        class AccessMod
        {
            public int num1;
        }

        static void Main(string[] args)
        {
            AccessMod ob1 = new AccessMod();
            //Direct access to public members
            ob1.num1 = 100;
            Console.WriteLine("Number one value in main {0}", ob1.num1);
            Console.ReadLine();
        }
    }
}
```

15.2.2 Private

Private access is the least permissive access level. Private members are accessible only within the body of the class or the struct in which they are declared.

Accessibility:

- Cannot be accessed by object
- Cannot be accessed by derived classes

In the following example num2 is not accessible outside the class.

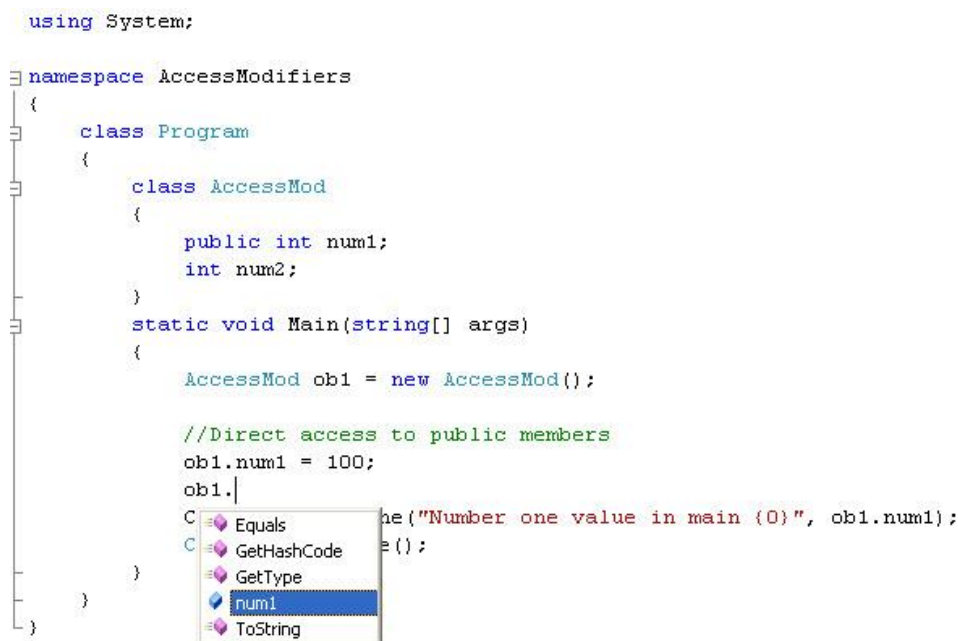
- Code

```
using System;

namespace AccessModifiers
{
    class Program
    {
        class AccessMod
        {
            public int num1;
            int num2;
        }

        static void Main(string[] args)
        {
            AccessMod ob1 = new AccessMod();
            //Direct access to public members
            ob1.num1 = 100;
            //Access to private member is not permitted
            ob1.num2 = 20;
            Console.WriteLine("Number one value in main {0}", ob1.num1);
            Console.ReadLine();
        }
    }
}
```

The above program will give compilation error, as access to private is not permissible. In the below figure you can see the private member num2 is not available.



```
using System;

namespace AccessModifiers
{
    class Program
    {
        class AccessMod
        {
            public int num1;
            int num2;
        }

        static void Main(string[] args)
        {
            AccessMod ob1 = new AccessMod();

            //Direct access to public members
            ob1.num1 = 100;
            ob1.
            C# Equals
            C# GetHashCode
            C# GetType
            C# num1
            C# ToString
            Console.WriteLine("Number one value in main {0}", ob1.num1);
            Console.ReadLine();
        }
    }
}
```

15.2.3 Protected

A protected member is accessible from within the class in which it is declared, and from within any class derived from the class that declared this member.

A protected member of a base class is accessible in a derived class only if the access takes place through the derived class type.

Accessibility:

- Cannot be accessed by object
- By derived classes
- Code

```
using System;

namespace AccessModifiers
{
    class Program
    {
        class Base
        {
            protected int num1;
        }

        class Derived : Base
        {
            public int num2;
            static void Main(string[] args)
            {
                Base ob1 = new Base();
                Derived ob2 = new Derived();
                ob2.num1 = 20;
                // Access to protected member as it is inherited by the Derived class
                ob2.num2 = 90;
                Console.WriteLine("Number2 value {0}", ob2.num2);
                Console.WriteLine("Number1 value which is protected {0}", ob2.num1);
                Console.ReadLine();
            }
        }
    }
}
```

In the above program we try to access protected member in main it is not available as shown in the picture below that num1 is not listed in intellisense.

```
using System;

namespace AccessModifiers
{
    class Program
    {
        class Base
        {
            protected int num1;
        }

        class Derived : Base
        {
            public int num2;
            static void Main(string[] args)
            {
                Base ob1 = new Base();
                Derived ob2 = new Derived();
                ob2.num1 = 20;

                // Access to protected member as it is inherited by the Derived class
                ob2.num2 = 90;

                ob1.
                // Intellisense dropdown showing:
                // Equals
                // GetHashCode
                // GetType
                // ToString
                Console.WriteLine("Number2 value {0}", ob2.num2);
                Console.WriteLine("Number1 value which is protected {0}", ob2.num1);
                Console.ReadLine();
            }
        }
    }
}
```

15.2.4 internal

The internal keyword is an access modifier for types and type members. We can declare a class as internal or its member as internal. Internal members are accessible only within files in the same assembly (.dll).

In other words, access is limited exclusively to classes defined within the current project assembly.

Accessibility:

- In same assembly (public)
- Can be accessed by objects of the class
- Can be accessed by derived classes

In other assembly (internal)

- Cannot be accessed by object
- Cannot be accessed by derived classes

15.2.5 protected internal

The protected internal accessibility means protected OR internal, not protected AND internal.

In other words, a protected internal member is accessible from any class in the same assembly, including derived classes.

The protected internal access modifier seems to be a confusing but is a union of protected and internal in terms of providing access but not restricting. It allows:

Inherited types, even though they belong to a different assembly, have access to the protected internal members.

Types that reside in the same assembly, even if they are not derived from the type, also have access to the protected internal members.

15.3 DEFAULT ACCESS

	namespace	class	struct	interface	enum
private		■	■		
internal	■	□	□		
public	□	□	□	■*	■*
protected		□			
protected internal		□			

A default access level is used if no access modifier is specified in a member declaration. The following list defines the default access modifier for certain C# types:

- enum: The default and only access modifier supported is public.
- class: The default access for a class is private. It may be explicitly defined using any of the access modifiers.
- interface: The default and only access modifier supported is public.
- struct: The default access is private with public and internal supported as well.

The default access may suffice for a given situation, but you should specify the access modifier you want to use to ensure proper application behavior.

Note: Interface and enumeration members are always public and no access modifiers are allowed.



16 GELIJKHEID

16.1 OPERATOR "=="

Indien je gelijkheid tussen verschillen instanties wil nagaan, moet je een onderscheid maken tussen fysieke en logische gelijkheid.

Twee objectvariabelen zijn fysiek gelijk aan elkaar als ze beiden wijzen naar hetzelfde object (naar dezelfde plaats in het geheugen).

Met de == operator kan je testen op fysieke gelijkheid van objecten (instanties van referencetypes), met andere woorden gaat het om hetzelfde object. Hiermee wordt dus gewoon de referenties met elkaar vergeleken.

- Code

```
using System;

namespace Example
{
    class Person
    {
        public string Name { get; set; }

        public override string ToString()
        {
            return Name;
        }
    }

    class Program
    {
        static void Main(string[] args)
        {
            Person person1 = new Person();
            Person person2 = new Person();
            Person person3 = person1;
            Console.WriteLine(person1 == person2);
            Console.WriteLine(person1 == person3);
            //
            Console.ReadLine();
        }
    }
}
```

- Output

```
False
True
```

16.2 OBJECT.REFERENCEEQUALS METHOD

Een alternatief voor de == operator is het gebruik van de method static bool ReferenceEquals(object objA, object objB) uit de System.Object klasse.

- Uit de library

```
// Summary:
//     Determines whether the specified System.Object instances are the same instance.
//
// Parameters:
//     objA:
//         The first System.Object to compare.
//
//     objB:
```

```
// The second System.Object to compare.
//
// Returns:
// true if objA is the same instance as objB or if both are null references;
// otherwise, false.
public static bool ReferenceEquals(object objA, object objB);
```

Deze zal van beide parameters nagaan of ze naar hetzelfde object verwijzen.

- Code

```
using System;

namespace Example
{
    class Person
    {
        public string Name { get; set; }

        public override string ToString()
        {
            return Name;
        }
    }

    class Program
    {
        static void Main(string[] args)
        {
            Person person1 = new Person();
            Person person2 = new Person();
            Person person3 = person1;
            Console.WriteLine(object.ReferenceEquals(person1, person2));
            Console.WriteLine(object.ReferenceEquals(person1, person3));
            //
            Console.ReadLine();
        }
    }
}
```

- Output

```
False
True
```

16.3 VERGELIJKEN MET NULL

Indien men wil nagaan of een objectvariabele wel een referentie bevat, kan men zowel bij de == operator, als de ReferenceEquals method met null vergelijken.

- Code

```
using System;

namespace Example
{
    class Person
    {
        public string Name { get; set; }

        public override string ToString()
        {
            return Name;
        }
    }

    class Program
    {
        static void Main(string[] args)
        {
            Person person1 = new Person();
```



```

        Person person2 = null;
        //
        Console.WriteLine(person1 == null);
        Console.WriteLine(person2 == null);
        Console.WriteLine(object.ReferenceEquals(person1, null));
        Console.WriteLine(object.ReferenceEquals(person2, null));
        Console.WriteLine(null == null);
        Console.WriteLine(Object.ReferenceEquals(null, null));
        //
        Console.ReadLine();
    }
}

```

- Output

```

False
True
False
True
True
True
True

```

16.4 FYSIEKE GELIJKHEID TUSSEN VALUE TYPES

Fysieke gelijkheid nagaan tussen twee valuetype instanties is onzinnig. Elke valuetype variabele is immers geassocieerd met zijn eigen valuetype instantie. Twee valuetype variabelen kunnen bijgevolg ook nooit naar dezelfde instantie verwijzen.

16.5 LOGISCHE GELIJKHEID TUSSEN VALUE TYPES

Naast fysieke gelijkheid bestaat er ook nog zoets als logische gelijkheid. De logica die gebruikt wordt om na te gaan of twee objecten logisch gelijk zijn is per type te bepalen.

Voor een aantal datatypes kan men de == vergelijingsoperator gebruiken om logische gelijkheid na te gaan. Dit geldt voor de valuetypes, maar ook voor sommige ingebouwde referencetypes.

Zo kan men bijvoorbeeld twee String objecten met de = operator op logische gelijkheid testen, bijvoorbeeld "abc" == "abc".

- Code

```

using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;

namespace ConsoleApplication1
{
    class Program
    {
        static void Main(string[] args)
        {
            Console.WriteLine(1 == 2);
            Console.WriteLine(1 != 2);
            //
            string string1 = "Hallo";
            string string2 = "Hallo";
            Console.WriteLine(string1 == string2);
            //
            Console.ReadLine();
        }
    }
}

```

- Output

```
False
True
True
```

Operator `!=` kan gebruikt worden om na te gaan of er een verschil is tussen de twee operanden. Waarbij verschil betekent dat ze logisch gezien niet gelijk zijn aan elkaar.

16.6 LOGISCHE GELIJKHEID TUSSEN REFERENCE TYPES

Logische gelijkheid is minder éénduidig dan fysieke gelijkheid, waarop logische gelijkheid wordt gebaseerd is afhankelijk van het type van de vergeleken objecten.

Men zou bijvoorbeeld twee `Person` objecten logische gelijk kunnen beschouwen als ze dezelfde naam hebben. Of twee `Addition` objecten kunnen beschouwen als logisch gelijk wanneer ze hetzelfde resultaat hebben.

Van het type `System.Object` erft iedere referencetype een `Equals(Object obj)` als Boolean method over. Het is trouwens deze die door `System.ValueType` wordt overgeërfd en geherdefinieerd.

De implementatie die echter door de referencetypes (rechtstreeks) van `System.Object` wordt overgeërfd zal de referenties (net als de `==` operator of de `ReferenceEquals` method) gaan vergelijken.

By default gaat het dus om fysieke gelijkheid die door deze implemenatie wordt nagegaan :

- Code

```
using System;

namespace Example
{
    class Person
    {
        public string Name { get; set; }
    }

    class Program
    {
        static void Main(string[] args)
        {
            Person person1 = new Person();
            person1.Name = "John";
            Person person2 = new Person();
            person2.Name = "John";
            Person person3 = new Person();
            person3 = person1;
            //
            Console.WriteLine(person1.Equals(person2));
            Console.WriteLine(person1.Equals(person3));
            //
            Console.ReadLine();
        }
    }
}
```

- Output

```
False
True
```

Zoals we reeds gezien hebben is deze `Equals` method echter wel virtual, wat maakt dat we het gedrag van deze kunnen herdefiniëren, en zelf kunnen uitkiezen waarop logische gelijkheid tussen twee instanties van ons type kan worden gebaseerd.

17 ENUMERATIES EN STRUCTS

Stel we hebben in ons systeem Customer-objecten waarbij hun leeftijdscategorie (GetCategory) voor het systeem belangrijk is om bepaalde logica op te baseren.

Voor het systeem zijn er bijvoorbeeld 3 leeftijdscategorieën waarbij steeds bepaalde logica gepaard gaat, er wordt in het systeem gesproken over de categorieën Child, Adult en Retired.

Een voorbeeld van logica die hiermee gepaard zou kunnen gaan is de van de AgeCategory afhankelijke kortingspercentage (Discount.Percentage).

Gezien de AgeCategory van een Customer afhankelijk is van de Age van de Customer is het logisch in de Customer-definitie de verantwoordelijkheid te stoppen om de AgeCategory van de bepaalde Customer-instantie op te vragen (Customer is de "information expert" van deze verantwoordelijkheid).

Om de lijst in ons systeem te definiëren/opsommen kunnen we gebruik maken van een enumeratie. Een enumeratie is een eenvoudige typedefinitie waarbij men enkel de mogelijke enumeratie-/opsommingswaardes dient te definiëren.

Net zoals andere typedefinities kan men nu variabelen van dit type gaan declareren (zie de AgeCategory argumentvariabele) of dit type als returntype voor een functie gebruiken.

Men kan in een client verwijzen naar een enumeratiewaarde door de puntnotatie te gebruiken. De identifier van de enumeratie wordt na de punt gevolgd door de identifier van de enumeratiewaarde.

- Code

```
using System;

namespace Example
{
    enum AgeCategory
    {
        Child,
        Adult,
        Retired,
    }

    class Customer
    {
        public byte Age{get; set;}
        public AgeCategory AgeCategory
        {
            get
            {
                if (Age < 18) return AgeCategory.Child;
                if (Age >= 60) return AgeCategory.Retired;
                else return AgeCategory.Adult;
            }
        }
        public override string ToString()
        {
            return Age.ToString() + " - " + AgeCategory.ToString();
        }
    }

    class Discount
    {
        public static byte Percentage(AgeCategory ageCategory)
        {
            if (ageCategory == AgeCategory.Child) return 10;
            if (ageCategory == AgeCategory.Adult) return 0;
            else return 20;
        }
    }
}
```

```

class Program
{
    static void Main(string[] args)
    {
        Customer customer1 = new Customer();
        customer1.Age = 15;
        Console.WriteLine(customer1);
        Console.WriteLine(Discount.Percentage(customer1.AgeCategory));
        //
        Customer customer2 = new Customer();
        customer2.Age = 45;
        Console.WriteLine(customer2);
        Console.WriteLine(Discount.Percentage(customer2.AgeCategory));
        //
        Customer customer3 = new Customer();
        customer3.Age = 75;
        Console.WriteLine(customer3);
        Console.WriteLine(Discount.Percentage(customer3.AgeCategory));
        //
        Console.ReadLine();
    }
}

```

- Output

```

15 - Child
10
45 - Adult
0
75 - Retired
20

```

Ieder enumeratietype dat gedefinieerd wordt is een type dat impliciet afgeleid wordt van het System.Enum-type (die in de klassenbibliotheek van het .NET Framework is voorgedefinieerd). Van dat System.Enum-type wordt een ToString()-implementatie overgeërfd die de corresponderende tekstuele waarde oplevert van die enumeratiewaarde.

Een enumeratiedefinitie zou je kunnen vergelijken met het Boolean-type, ook deze heeft een vaste reeks van waarden met een specifieke naam (True en False).

De enumeratiewaardes moet vastgelegd worden at designtime en kan dus na compiletime (bijvoorbeeld at runtime) niet meer wijzigen.

Aan elke enumeratiewaarde is een gehele numerieke waarde gekoppeld. By default zijn dit Integer-waardes. De eerste enumeratiewaarde wordt gekoppeld aan 0, de volgende aan 1, de volgende aan 2,

- Code

```

using System;

namespace Example
{
    enum Month : byte
    {
        January = 1,
        February = 2,
        March = 3,
        April = 4,
        May = 5,
        June = 6,
        July = 7,
        August = 8,
        September = 9,
        October = 10,
        November = 11,
        December = 12
    }
}

```

```

class Program
{
    static void Main(string[] args)
    {
        Month month1 = 0;
        Console.WriteLine(month1);
        Console.WriteLine(GetMonthName(month1));
        //
        month1 = Month.January;
        Console.WriteLine(month1);
        Console.WriteLine(GetMonthName(month1));
        //
        Month month2 = (Month)9;
        Console.WriteLine(month2);
        Console.WriteLine(GetMonthName(month2));
        //
        if (month1 < month2)
        {
            Console.WriteLine(month1.ToString() + " comes before "
                               + month2.ToString());
        }
        else
        {
            Console.WriteLine(month1.ToString() + " doesn't comes before "
                               + month2.ToString());
        }
        //
        Console.WriteLine(GetMonthName((Month)5));
        Console.WriteLine(GetMonthName((Month)13));
        //
        Month month3 = (byte)Month.January + Month.March;
        Console.WriteLine(month3);
        Console.WriteLine(GetMonthName(month3));
        //
        foreach (byte month in Enum.GetValues(typeof(Month)))
        {
            Console.WriteLine(month);
        }
        foreach (string month in Enum.GetNames(typeof(Month)))
        {
            Console.WriteLine(month);
        }

        Console.ReadLine();
    }
    static string GetMonthName(Month month)
    {
        if (System.Enum.IsDefined(month.GetType(), month))
        {
            return month.ToString();
        }
        else
        {
            return "unknown";
        }
    }
}

```

- Output

```

0
unknown
January
January
September
September
January comes before September
May
unknown
April
April
1
2
3

```

```
4
5
6
7
8
9
10
11
12
January
February
March
April
May
June
July
August
September
October
November
December
```

In bovenstaande client merk je dat de inhoud een enumeratiewaarde correspondeert met de gekoppelde numerieke waarde. Enkel bij het vormen van de enumeratie-expressie of bij het opvragen van de ToString()-representatie zal de tekstuele vorm worden gebruikt.

Doordat de enumeratie-waarde intern als numerieke waardes worden gerepresenteerd is het ook mogelijk vergelijkingen of bewerkingen met deze enumeratiewaardes te maken.

De GetType()-method overgeërfd van de System.Enum-klasse is bruikbaar om in System.Type-vorm informatie over het desbetreffende type op te vragen.

Een System.Type-object is een abstractie van een datatype, dit object bevat metadata omtrent het geabstraheerde datatype.

Zo'n 'System.Type'-object die een bepaald enumeratietype abstraheert is nodig als eerste argument van de 'System.Enum.IsDefined'-classmethod. Deze method levert 'True' of 'False' op naargelang de tweede (numerieke) enumeratiewaarde gedefinieerd is in het opgegeven enumeratietype (opgegeven in het eerste argument).

17.1.1 Startwaarde aanpassen

De startwaarde van een enumeratie is 0, maar kan als volgt worden aangepast:

- Output

```
enum Seizoen { lente = 1, zomer, herfst, winter }
```

17.1.2 Elementen met eenzelfde waarde

Je kan elementen opnemen in een enumeratie die dezelfde waarde hebben als een reeds aanwezig element (een soort van synoniem dus)

- Output

```
enum Seizoen { lente, zomer,herfst, winter,spring = lente,summer = zomer,autumn = herfst }
```

17.2 STRUCTURE

- Enumeraties zijn heel eenvoudig namen die in een bepaalde volgorde kunnen worden gezet, en worden als value doorgegeven (op de stack).

- Klassen zijn elementen die je kan gebruiken om volwaardige objecten te maken, op de stack plaatsen we een adres, waarbij op de heap de data van het object wordt gestockeerd.

Soms is het goed een tussenvorm te gebruiken, meer mogelijkheden dan een enumeratie, maar toch niet de overhead van een klasse, waarbij communicatie is tussen stack en heap en acties intern dus omslachtiger en trager zijn.

- Een Structure is hier het antwoord op: een structure kan fields, methods en constructors hebben, maar wordt gemanaged op de stack en dus doorgegeven by value.

Concreet wordt binnen .Net veel gebruik gemaakt van structures: int, long en float zijn eigenlijk niks anders dan verkorte schrijfwijzen voor de Structures System.Int32, System.Int64 en System.Single. Deze structures hebben fields en methods (zoals ToString()).

Een structure wordt voornamelijk gebruikt voor entiteiten die belangrijk zijn om hun waarde, zoals een getal, een tijd, een datum, ...

- Output

```
using System;

namespace Example
{
    struct Tijd
    {
        public int uren, minuten, seconden;

        public Tijd(int uu, int mm, int ss)
        {
            uren = uu % 24;
            minuten = mm % 60;
            seconden = ss % 60;
        }

        public string ToonTijd()
        {
            return String.Format("{0}:{1}:{2}"
                , TweeCijfersGewenst(uren)
                , TweeCijfersGewenst(minuten)
                , TweeCijfersGewenst(seconden));
        }

        private string TweeCijfersGewenst(int getal)
        {
            string getaltekst = getal.ToString();
            if (getaltekst.Length < 2)
                getaltekst = "0" + getaltekst;
            return getaltekst;
        }
    }

    class Program
    {
        static void Main(string[] args)
        {
            Tijd tijd = new Tijd(11,5,40);
            Console.WriteLine("Tijd = " +tijd.ToonTijd());
            //
            Console.ReadLine();
        }
    }
}
```

17.2.1 Beperkingen van een struct

Een struct lijkt wellicht een aantrekkelijk alternatief voor klassen, maar het heeft enkele beperkingen:

- U kunt geen standaardconstructor opnemen voor een struct. Deze wordt namelijk automatisch door de compiler ingevuld. Als u een constructor nodig hebt, dan moet deze minimaal een parameter bevatten.
- Een struct kan niet afleiden van een andere struct en evenmin kan een struct dienen als basis voor een andere struct. Een struct is dus per definitie sealed.
- U kunt een veld in een struct geen initiële waarde geven.

Om deze redenen zult u in de praktijk de struct slechts toepassen om objecten met een beperkte hoeveelheid functionaliteit te typeren.

18 PARTIAL

18.1 PARTIAL NAMESPACE EN CLASSES

Het is mogelijk om de code van zowel een namespace als een klasse over verschillende broncodebestanden te spreiden.

In het geval van een klasse dienen alle klassen van de gedeeltelijke klassendefinities met het keyword `Partial` gemarkeerd te worden. Zo zie je meteen dat deze klassendefinitie nog ergens anders wordt aangevuld. Bij namespaces is dit niet nodig.

- Code

```
using System;

namespace Example
{
    partial class Product
    {
        private decimal _Price;
        public decimal Price
        {
            get { return _Price; }
            set { _Price = value; }
        }
    }
    partial class Product
    {
        private decimal _Tax;
        public decimal Tax
        {
            get { return _Tax; }
            set { _Tax = value; }
        }
    }
}

namespace Example
{
    partial class Product
    {
        public decimal PriceIncludingTax()
        {
            return Price * (1 + (Tax / 100));
        }
    }

    class Program
    {
        static void Main(string[] args)
        {
            Product product1 = new Product();
            product1.Price = 200;
            product1.Tax = 21;
            //
            Console.WriteLine(product1.PriceIncludingTax() == 242);
            //
            Console.ReadLine();
        }
    }
}
```

- Output

```
True
```

There are several situations when splitting a class definition is desirable:

- Then working on large projects, spreading a class over separate files enables multiple programmers to work on it at the same time.
- When working with automatically generated source, code can be added to the class without having to recreate the source file. Visual Studio uses this approach when it creates Windows Forms, Web service wrapper code, and so on. You can create code that uses these classes without having to modify the file created by Visual Studio.

The `partial` keyword indicates that other parts of the class, struct, or interface can be defined in the namespace. All the parts must use the `partial` keyword. All the parts must be available at compile time to form the final type. All the parts must have the same accessibility, such as `public`, `private`, and so on.

If any part is declared `abstract`, then the whole type is considered `abstract`. If any part is declared `sealed`, then the whole type is considered `sealed`. If any part declares a base type, then the whole type inherits that class.

All the parts that specify a base class must agree, but parts that omit a base class still inherit the base type. Parts can specify different base interfaces, and the final type implements all the interfaces listed by all the partial declarations. Any class, struct, or interface members declared in a partial definition are available to all the other parts. The final type is the combination of all the parts at compile time.

- Code

```
using System;

public partial class CoOrds
{
    private int x;
    private int y;

    public CoOrds(int x, int y)
    {
        this.x = x;
        this.y = y;
    }
}

public partial class CoOrds
{
    public void PrintCoOrds()
    {
        Console.WriteLine("CoOrds: {0},{1}", x, y);
    }
}

class TestCoOrds
{
    static void Main()
    {
        CoOrds myCoOrds = new CoOrds(10, 15);
        myCoOrds.PrintCoOrds();

        // Keep the console window open in debug mode.
        Console.WriteLine("Press any key to exit.");
        Console.ReadKey();
    }
}
```

- Output

```
Output: CoOrds: 10,15
```

18.2 PARTIAL METHODS

Ook methods kunnen als Partial worden gedefinieerd. Deze methods kunnen dan met implementatie aangevuld worden op een andere plaats binnen de klassedefinitie.

De Partial methods kunnen enkel Private Methods zijn zonder implementatie.

- Code

```
// Definition in file1.cs
partial void onNameChanged();

// Implementation in file2.cs
partial void onNameChanged()
{
    // method body
}
```

A partial class or struct may contain a partial method. One part of the class contains the signature of the method. An optional implementation may be defined in the same part or another part. If the implementation is not supplied, then the method and all calls to the method are removed at compile time.

Partial methods enable the implementer of one part of a class to define a method, similar to an event. The implementer of the other part of the class can decide whether to implement the method or not. If the method is not implemented, then the compiler removes the method signature and all calls to the method. The calls to the method, including any results that would occur from evaluation of arguments in the calls, have no effect at run time. Therefore, any code in the partial class can freely use a partial method, even if the implementation is not supplied. No compile-time or run-time errors will result if the method is called but not implemented.

Partial methods are especially useful as a way to customize generated code. They allow for a method name and signature to be reserved, so that generated code can call the method but the developer can decide whether to implement the method. Much like partial classes, partial methods enable code created by a code generator and code created by a human developer to work together without run-time costs.

A partial method declaration consists of two parts: the definition, and the implementation. These may be in separate parts of a partial class, or in the same part. If there is no implementation declaration, then the compiler optimizes away both the defining declaration and all calls to the method.

- Partial method declarations must begin with the contextual keyword `partial` and the method must return `void`.
- Partial methods can have `ref` but not `out` parameters.
- Partial methods are implicitly private, and therefore they cannot be virtual.
- Partial methods cannot be extern, because the presence of the body determines whether they are defining or implementing.
- Partial methods can have static and unsafe modifiers.
- Partial methods can be generic. Constraints are put on the defining partial method declaration, and may optionally be repeated on the implementing one. Parameter and type parameter names do not have to be the same in the implementing declaration as in the defining one.
- You can make a delegate to a partial method that has been defined and implemented, but not to a partial method that has only been defined.

Enkel de implementatieloze methods worden gemarkeerd met het keyword `Partial`. De aanvullende implementatie mag dit keyword niet opnemen in zijn signatuur.

Het is technisch gezien mogelijk om binnen dezelfde (partial) klassedefinitie - als waar de `Partial` method is gedefiniëerd - de aanvullende implementatie te definiëren.

Toch zal doorgaans de aangevulde implementatie zich in een andere partial definitie van deze klasse bevinden.

Partial methods worden vooral toegepast in situatie waar men wil vermijden dat de code - gegenereerd door bepaalde code-generatie - wordt aangepast.

Veronderstel dat onderstaande klasse `Stock` door een code-generator werd gecreëerd.

Indien men het toe wil laten, de programmeur te laten bepalen wat er dient te gebeuren bij het instellen van de `Price` eigenschap. Dan kan de generator nu `Partial` methods definiëren en aanroepen in de relevante setter.

- Code

```
using System;

namespace Example
{
    #region "Some Generated Code"
    partial class Stock
    {
        private decimal _Price;
        public decimal Price
        {
            get{return _Price;}
            set
            {
                onPriceChanging(value);
                _Price = value;
                onPriceChanged();
            }
        }
        partial void onPriceChanging(decimal value);
        partial void onPriceChanged();
    }
    #endregion

    class Program
    {
        static void Main(string[] args)
        {
            Stock stock1 = new Stock();
            stock1.Price = 10;
            //
            Console.ReadLine();
        }
    }
}
```

Een programmeur die wil bepalen wat er moet gebeuren bij het instellen van de `Price` eigenschap heeft nu de mogelijkheid aanvullende implementaties te definiëren.

- Code

```
partial class Stock
{
    partial void onPriceChanging(decimal value)
    {
        Console.WriteLine("Doing something when the price is changing to " +
                           value + ".");
    }
    partial void onPriceChanged()
}
```

```
{  
    Console.WriteLine("Doing something when the price changed.");  
}
```

- Output

```
Doing something when the price is changing to 10.  
Doing something when the price changed.
```

Om performantieredenen zal indien geen aanvullende implementatie is voorzien voor een Partial method, de compiler de call naar deze method negeren.

19 CONVERSIONS

19.1 IMPLICIETE CONVERSIES

Conversie is het at runtime omzetten van data naar een ander formaat. Conversie houdt in dat men aan variabelen van bepaalde datatypes waarden gaat toekennen die geformuleerd zijn aan de hand van expressies van bepaalde andere datatypes.

Impliciete conversie staat tegenover expliciete conversie, waarbij men expliciet opgeeft VAN welk datatype NAAR welk ander datatype men wil converteren.

- Narrowing conversie vindt plaats wanneer men van een "groter" naar een "kleiner" datatype wil converteren. "Kleiner" kan slaan op een mindere opslagcapaciteit of een "kleinere" precisie of nauwkeurigheid.
- Widening conversie is het tegenovergestelde van narrowing conversie, daar gaat men van een "kleiner" naar een "groter" datatype converteren.

No special syntax is required because the conversion is type safe and no data will be lost. Examples include conversions from smaller to larger integral types, and conversions from derived classes to base classes.

Bij narrowing conversie kan een runtime-fout (OverflowException) optreden indien het bereik van het doeldatatype te beperkt zou zijn. Dit wordt door de compiler gezien, en geretourneerd met een error.

- Code

```
using System;

namespace Example
{
    class Program
    {
        static void Main(string[] args)
        {
            int intGetal = 25;
            long lngGetal = intGetal;
            //
            //long lngGetal2 = 25;
            //int intGetal2 = lngGetal2;
            //
            //double dblGetal = 33.5;
            //intGetal = dblGetal;
            //
            Console.ReadLine();
        }
    }
}
```

- Error

```
Error
Cannot implicitly convert type 'double' to 'int'. An explicit conversion exists (are you missing a cast?)
```

19.2 EXPLICIETE CONVERSIES (CASTING)

Naast het impliciet converteren, kunnen we dus ook expliciet converteren, waarbij we opgeven VAN welk datatype, NAAR welk datatype we wensen te converteren.

Expliciet converteren kan door een aantal static functies uit de klasse `System.Convert` te gaan gebruiken. Deze functies hebben tal van overladen versies om argumentwaarden van zowat alle intrinsic types aan te kunnen doorgeven.

- Code

```
using System;

namespace Example
{
    class Program
    {
        static void Main(string[] args)
        {
            int lngGetal;
            double dblGetal = 34.5;
            int intGetal = 100;
            byte bytGetal;
            //
            bytGetal = System.Convert.ToByte(dblGetal);
            //bytGetal = (byte)dblGetal;

            Console.WriteLine(bytGetal);
            //
            bytGetal = System.Convert.ToByte(intGetal);
            //bytGetal = (byte)intGetal;
            lngGetal = bytGetal;
            lngGetal = intGetal;
            //
            object obj1 = Convert.ChangeType(123, TypeCode.Byte);
            object obj2 = Convert.ChangeType(-1, Convert.GetTypeCode(false));
            Console.WriteLine(obj2);
            //
            Console.ReadLine();
        }
    }
}
```

- Output

```
34
True
```

19.2.1 Een woordje uitleg

- EXPLICIETE NARROWING CONVERSIES
- EXPLICIETE WIDENING CONVERSIE
- EXPLICIETE CONVERSIE RESULTEREND IN BOXING OPERATIE via `Convert.ChangeType(...)`:
 - de waarden (1^{ste} argument-waarde) worden omgezet naar het opgegeven type (2^{de} argument-waarde, van enumeratietype `System.TypeCode`) en zo geboxed in een Object-variabele waarvan de referentie wordt opgeleverd
- Voor zowat alle intrinsic types bestaan conversie functies, bijvoorbeeld:
 - `System.Convert.ToInt16(...)` voor conversie naar Short
 - `System.Convert.ToInt32(...)` voor conversie naar Integer
 - `System.Convert.ToSingle(...)` voor conversie naar Single
 - `System.Convert.ToDouble(...)` voor conversie naar Double
 - `System.Convert.ToBoolean(...)` voor conversie naar Boolean
 - `System.Convert.ToDateTime(...)` voor conversie naar DateTime

- `System.Convert.ToChar(...)` voor conversie naar `Char`
- ...

19.2.2 Converteren naar String

Hieronder zien we een voorbeeld hoe men expliciet naar het String-formaat kan converteren.

- Code

```
using System;

namespace Example
{
    class Person
    {
        public string Naam {get; set; }
    }
    class Program
    {
        static void Main(string[] args)
        {
            Person person1 = new Person();
            person1.Naam = "John";
            Console.WriteLine(person1);
            Console.WriteLine(person1.ToString());
            //
            Console.ReadLine();
        }
    }
}
```

- Output

```
Example.Person
Example.Person
```

De `System.Convert.ToString(...)`-method zet de opgegeven argumentwaarde om naar een tekstuele representatie, dit wordt gebaseerd op de `ToString()`-implementatie van dat type zelf.

Zoals we reeds weten kunnen we ook zelf deze implementatie bepalen, door die implementatie te gaan overschrijven, zoals in bovenstaand voorbeeld is gebeurt voor het `Person`-type.

Bij `Convert.ToString(...)` zullen nooit runtime-fouten optreden. Elk type beschikt immers over een `ToString()`-implementatie, minimaal is deze implementatie overgeërfd van het `System.Object`-basistype.

Zoals we reeds vroeger zagen, kunnen we de `ToString()`-methode herdefiniëren.

- Code

```
using System;

namespace Example
{
    class Person
    {
        public string Naam {get; set; }
        public override string ToString()
        {
            return Naam;
        }
    }
    class Program
    {
        static void Main(string[] args)
        {
            Person person1 = new Person();
            person1.Naam = "John";
            Console.WriteLine(person1);
            Console.WriteLine(person1.ToString());
        }
    }
}
```



```

        //
        Console.ReadLine();
    }
}

```

- Output

```

John
John

```

In onderstaand voorbeeld zien we in de Example1 een expressie person1 waarvan het DTE verschillend is van het STE. Het STE van person1 is Person, het DTE van person1 is Student.

- Code

```

using System;

namespace Example
{
    class Person
    {
        public Person(string naam)
        {
            this.Naam = naam;
        }
        public string Naam { get; set; }
    }

    class Student : Person
    {
        public Student(string naam, string cursus) : base(naam)
        {
            this.Cursus = cursus;
        }
        public string Cursus { get; set; }
    }

    class Program
    {
        static void Main(string[] args)
        {
            Person person1 = new Student("John", "A5");
            Console.WriteLine(person1.Naam);
            Console.WriteLine(person1.Cursus); //compile error
            //
            Console.ReadLine();
        }
    }
}

```

- Output

```

John

```

Ondanks de expressie person1 wijst naar een object van het type Student (DTE), is het toch niet mogelijk via deze ontvanger-expressie de member Cursus aan te roepen.

Dit is logisch gezien het STE person1 Person is, en de compiler dus zal opzoeken of de member (Cursus) die je hierop aanroept zich in de interface (de publieke members) van het type Person bevindt - wat hier niet het geval is. Enkel wanneer je een member uit de interface van die klasse (STE) aanroept, weet de compiler zeker dat er een implementatie aan kan worden gekoppeld. Het zal ten slotte minimaal gaan om een object van het type Person waar naartoe deze expressie verwijst. Zelfs al gaat het om een object van een afgeleid type waar naartoe deze expressie verwijst (zoals hier in het voorbeeld waar het DTE verschilt van het STE), kan er nog steeds een (minimaal overgeërfde) implementatie aan worden gekoppeld.

Om het probleem te vermijden waar het niet mogelijk is via een ontvangerexpressie van een bepaald STE de members van het eigenlijke DTE aan te roepen kan men gaan casten.

- Code

```
class Program
{
    static void Main(string[] args)
    {
        Person person1 = new Student("John ", "A5");
        Console.WriteLine(person1.Naam);
        Console.WriteLine(((Student)person1).Cursus);
        //
        Console.ReadLine();
    }
}
```

- Output

```
John
A5
```

Casten is het duidelijk maken aan de compiler dat deze expressie van een ander statisch type moet worden beschouwd dan wat de declaratie of de definitie van deze expressie ervan aangeeft.

De Cast-expressie die zo gevormd wordt, wordt inline gecompileerd en vormt dan eigenlijk een expressie van dat type dat de tweede argumentwaarde opgeeft. Inline compileren is ervoor zorgen dat de conversiecode deel uitmaakt van de code die deze expressie evalueert.

De expressie `person1` heeft als statisch type `Person`, de expressie `“(Student)person1”` heeft als statisch type `Student`.

Door middel van casting kan men dus het STE gaan wijzigen. Hiermee wordt het dus opnieuw mogelijk de members van het eigenlijke type (DTE) te gaan aanroepen ook al bevindt de expressie zich in de vorm van een basis type (STE) van dit eigenlijke type (DTE).

Men kan enkel neerwaarts en opwaarts in de hiërarchie van de inheritanceketen gaan casten, men spreekt respectievelijk over down- en upcasting.

Andere vormen van casting zijn onzinnig gezien er geen erfelijk verband zal zijn tussen die 2 types.

Upcasting is technisch gezien mogelijk, maar weinig zinvol (bij talen met enkel single-inheritance) gezien LSP.

We kunnen reeds op een expressie met statisch type van een afgeleide klasse de members uit de interface aanroepen die zijn overgeërfd uit het basistype. LSP geeft ons immers reeds aan dat overal waar een expressie van een basistype wordt gebruikt men deze expressie kan vervangen door een van een afgeleid type.

Opgelet: casting is niet hetzelfde als converten. Converten is het at runtime omzetten van data naar een ander formaat, casting is at-compiletime (of er net voor) aangeven dat een expressie van een ander type moet worden geëvalueerd als het type waaronder de expressie is gedeclareerd of gedefinieerd.

19.3 VOORWAARDELIJK CASTEN EN DE TYPEOF OPERATOR

Het gebruik van 'DirectCast' zal een compile-fout geven indien men probeert te casten tussen 2 types waartussen geen inheritance (of implementation) relation heerst. In zo'n gevallen kan men dus nooit een foutief programma bekomen.

Wanneer men echter probeert te casten naar een foutief type - weliswaar binnen de inheritanceketen, zal er een `InvalidCastException` runtime-fout optreden.

- Code

```

using System;

namespace Example
{
    class Person
    {
        public Person(string naam)
        {
            this.Naam = naam;
        }
        public string Naam { get; set; }
    }

    class Student : Person
    {
        public Student(string naam, string cursus): base(naam)
        {
            this.Cursus = cursus;
        }
        public string Cursus { get; set; }
    }

    class Teacher : Person
    {
        public Teacher(string naam, string klasLokaal): base(naam)
        {
            this.KlasLokaal = klasLokaal;
        }
        public string KlasLokaal { get; set; }
    }

    class Program
    {
        static void Main(string[] args)
        {
            Person person1 = new Student("John ", "A5");
            Console.WriteLine(person1.Naam);
            Console.WriteLine(((Teacher)person1).KlasLokaal); // runtime error
            // Console.ReadLine();
        }
    }
}

```

De runtime-fout die hierbij optreedt bevat de vermelding: "Unable to cast object of type Student to type Teacher.". Het DTE person1 is immers Student en niet Teacher.

Bovenstaande Client gaat dus onvoorwaardelijk casten.

In een "weakly-typed" programmeertaal (een taal waar er at runtime nog typefouten kunnen optreden) zoals C# .NET (maar ook zoals alle voor de hand liggende OOP's) gaat men beter voorwaardelijk casten.

- Code

```

class Program
{
    static void Main(string[] args)
    {
        Person person1 = new Student("John ", "A5");
        Teacher teacher1 = new Teacher("John ", "123");
        Console.WriteLine(person1.Naam);
        if (person1.GetType() == typeof(Teacher))
        {
            teacher1 = (Teacher)person1;
        }
        else
        {
            Console.WriteLine("No teacher");
        }
        // Console.ReadLine();
    }
}

```

```
}
```

- Output

```
No Teacher.
```

De `typeof`-operator kan gebruikt worden om het dynamisch type van een expressie na te gaan, en dat op volgende wijze: `typeof expression is type-specifier`.

Deze booleaanse expressie zal evalueren naar `True` indien het DTE het type (of een afgeleid type) is dat is opgegeven in de type-specifier.

- Code

```
class Program
{
    static void Main(string[] args)
    {
        Person[] persons = { new Person("John"),
                             new Student("Jane", "A5"),
                             new Teacher("Philippe", "123")
        };

        //
        foreach (Person element in persons)
        {
            string value = element.GetType().Name;
            //
            switch (value)
            {
                case "Student":
                    Console.WriteLine("Student.");
                    break;
                case "Teacher":
                    Console.WriteLine("Teacher.");
                    break;
                default:
                    Console.WriteLine("Person.");
                    break;
            }
        }
        //
        Console.ReadLine();
    }
}
```

- Output

```
Person.
Student.
Teacher.
```

19.3.1 GetType Method

Ook de `GetType()`-method wordt in bovenstaand voorbeeld gedemonstreerd.

Deze method is gedefinieerd in het `System.Object`-type en wordt van deze overgeërfd in alle andere types. Deze method levert een `System.Type`-object op dat een abstractie vormt voor het dynamisch type van de expressie waarop deze method wordt gebruikt. De `Name` property van een `System.Type`-object levert in `String` vorm de identifier van het geabstraheerde type op.

Zowel de `typeof`-operator als de `GetType()`-method zijn vormen van reflection. Reflection is het inspecteren van types (van objecten), dit kan zowel statisch als dynamische informatie opleveren.

20 ABSTRACTE BASIS KLASSEN

20.1 SEALED KLASSEN

Wanneer een klasse gedefinieerd wordt dan is die by default overerfbaar (inheritable).

Dit zouden we kunnen aanpassen door het Sealed keyword op te nemen in de klasse-definitie.

Het is hierbij niet meer mogelijk om in een overervingsclausule te verwijzen naar de "niet-overerfbare" klasse.

- Code

```
class InheritableBaseClass
{
}
class SubClass1: InheritableBaseClass
{
}

sealed class NotInheritableClass
{
}
class SubClass2: NotInheritableClass
{
}
```

Sealed-klassen worden ook wel "final" klassen genoemd.

Sealed classes worden vooral gebruikt om de klasse af te schermen, wanneer je dus wenst dat de klasse niet herbruikt kan worden bij wijze van overerving.

20.2 ABSTRACT

Naast de default inheritable klassen bestaat er nog een andere optie, namelijk abstracte klassen. Deze klassen worden met het Abstract-keyword gemarkeerd.

Overerving wordt hier als het ware verplicht. "Verplicht" is het in de zin dat je geen instantie kan creëren van zo'n abstracte klasse, maar je slecht iets kan doen met de members van die abstracte klasse indien je van die klasse gaat overerven. Het is enkel mogelijk om instanties te creëren van concreet (dus niet abstracte) afgeleide types van dit abstracte type.

Zo'n constructie is dus nuttig in geval van een generalisatie-type die zelf een eerder abstract concept omschrijft waarvan zelf geen objecten gemaakt moeten worden. Het zal uiteraard enkel nuttig zijn wanneer men beschikt over een concreet afgeleid type.

In een systeem hebben we bijvoorbeeld nood aan figuren als rechthoeken en cirkels. Alle soorten van figuren hebben een bepaalde kleur. Een generalisatie-type als Figuur met de gemeenschappelijke kenmerken (zoals Kleur) en gemeenschappelijk gedrag voor alle figuren zou hier dus nuttig zijn.

Maar omdat er in ons systeem niet zoiets is als "een gewone figuur", maar het altijd om specifieke soorten van figuren gaat, kunnen we beter de Figuur-klasse abstract (abstract) maken. Dit voorkomt dat we in een client objecten van dit type zouden maken.

- Code

```
using System;

namespace Example
{
    abstract class Figuur
    {
        public string Kleur { get; set; }
    }
}
```

```

class Rechthoek : Figuur
{
    public int Hoogte { get; set; }
    public int Breedte { get; set; }
}

class Cirkel : Figuur
{
    public int Straal { get; set; }
}

class Program
{
    static void Main(string[] args)
    {
        Rechthoek fig1 = new Rechthoek();
        Cirkel fig2 = new Cirkel();
        //Figuur fig3 = new Figuur();
        Figuur fig4 = new Rechthoek();
        //
        Console.ReadLine();
    }
}

```

Aan de overgeërfdde eigenschap *Kleur* kunnen nu in de gespecialiseerde types nog gespecialiseerde eigenschappen als *Hoogte*, *Breedte* of *Straal* bijvoorbeeld worden toegevoegd.

Abstracte klassen kunnen ook worden overgeërfd door andere abstracte klassen. Maar ook met deze afgeleide abstracte klassen zul je weinig zijn zolang je daarvan geen concrete klassen laat overerven.

Een klasse die niet geïnstantieerd kan worden is uiteraard slechts nuttig door de members ervan over te nemen in een concreet afgeleid type waarvan je wel een instantie kunt creëren.

20.3 CONSTRUCTOREN VAN EEN ABSTRACTE KLASSE

Ondanks dat we geen objecten kunnen instantiëren van het abstracte type, kan het toch nuttig zijn een constructor op te nemen in deze klassen.

De reden hiervoor is dezelfde zoals bij concrete klassen, namelijk de mogelijkheid in deze constructiecode op te nemen.

De *Figuur*-klasse zou bijvoorbeeld een constructor kunnen gebruiken om de initiële *Kleur*-waarde aan door te geven.

- Code

```

using System;

namespace Example
{
    abstract class Figuur
    {
        protected Figuur(string kleur)
        {
            this.Kleur = kleur;
        }
        protected Figuur() { }

        public string Kleur { get; set; }
    }

    class Rechthoek : Figuur
    {
        public Rechthoek(string kleur, int hoogte, int breedte)
            : base(kleur)
        {

```

```

        this.Hoogte = hoogte;
        this.Breedte = breedte;
    }
    public Rechthoek() { }

    public int Hoogte { get; set; }
    public int Breedte { get; set; }
}

class Cirkel : Figuur
{
    public Cirkel(string kleur, int straal)
        : base(kleur)
    {
        this.Straal = straal;
    }
    public Cirkel() { }

    public int Straal { get; set; }
}

class Program
{
    static void Main(string[] args)
    {
        Rechthoek fig1 = new Rechthoek("red", 10, 20);
        Cirkel fig2 = new Cirkel("yellow", 15);
        //Figuur fig3 = new Figuur();
        Figuur fig4 = new Rechthoek();
        //
        Console.ReadLine();
    }
}

```

De constructor van een abstracte klasse kan zelfs de Public access modifier krijgen. Doe dit echter niet, dit werkt louter verwarrend omdat Public gebruikt wordt om aan te geven dat de member in de publieke interface van dat type zit en de member dus aanroepbaar is voor clients van dat type.

Maar bij een abstracte klasse met een Public constructor blijft het onmogelijk een object van dit type te instantiëren.

Protected is een veel meer voor de hand liggende keuze. Een Protected-member kan worden aangeroepen door de afgeleide klassen, dus kan hierbij ook de overgeërfde initialisatie-code van de constructor van die abstracte basis klasse aangeroepen worden (en herbruikt worden) in de afgeleide klassen.

20.4 ABSTRACTE MEMBERS

Niet alleen een klasse zelf kunnen we abstract maken, ook members kunnen abstract worden gemaakt. Deze abstracte members hebben een signatuur (zijn aanwezig in een klasse met een bepaalde interface), maar hebben geen implementatie.

Uiteraard kunnen we zo'n members enkel in een abstracte klasse definiëren, want anders zouden we voor objecten van concrete types die members kunnen aanroepen, zonder dat geweten is welke implementatie aan die aanroep moet gekoppeld worden (welke binding moet worden gebruikt).

Een member maak je abstract via het keyword abstract, en de definitie hiervan bestaat enkel uit de signatuurregel. Een concreet afgeleide klasse is dan verplicht een override implementatie te gaan binden aan die memberinterface die hij van die abstracte klasse heeft overgeërfd.

Alle figuren in ons systeem (alle afgeleide types van Figuur) zouden bijvoorbeeld nut hebben aan een member die de oppervlakte oplevert.

Gezien deze member voor alle soorten van figuren van toepassing is, kunnen we deze member best in het generalisatie-type Figuur opnemen, zo kan de member worden overgenomen naar alle gespecialiseerde afgeleide types.

We kunnen echter moeilijk een implementatie koppelen aan die member omdat er simpelweg geen algemene formule bestaat voor het bepalen van de oppervlakte die voor alle gespecialiseerde soorten figuren van toepassing is.

Dit is geen probleem we kunnen die member als abstracte member (abstract-member) opnemen in het generalisatie-type Figuur, alle concrete afgeleide types kunnen/moeten daarvoor dan een implementatie voorzien (override-implementatie).

- Code

```
using System;

namespace Example
{
    abstract class Figuur
    {
        protected Figuur(string kleur)
        {
            this.Kleur = kleur;
        }

        public string Kleur { get; set; }
        public abstract double GetOppervlakte();
    }

    class Rechthoek : Figuur
    {
        public Rechthoek(string kleur, int hoogte, int breedte)
            : base(kleur)
        {
            this.Hoogte = hoogte;
            this.Breedte = breedte;
        }

        public int Hoogte { get; set; }
        public int Breedte { get; set; }
        public override double GetOppervlakte()
        {
            return Hoogte * Breedte;
        }
    }

    class Cirkel : Figuur
    {
        public Cirkel(string kleur, int straal)
            : base(kleur)
        {
            this.Straal = straal;
        }

        public int Straal { get; set; }
        public override double GetOppervlakte()
        {
            return Straal * Straal * Math.PI;
        }
    }

    class Program
    {
        static void Main(string[] args)
        {
            Rechthoek fig1 = new Rechthoek("red", 10, 20);
            Cirkel fig2 = new Cirkel("yellow", 15);
            //Figuur fig3 = new Figuur();
            Figuur fig4 = new Rechthoek("red", 20, 30);
            //
        }
    }
}
```



```

        Console.WriteLine(fig1.GetOppervlakte());
        Console.WriteLine(fig2.GetOppervlakte());
        Console.WriteLine(fig4.GetOppervlakte());

        //
        Console.ReadLine();
    }
}

```

- Output

```

200
706,858347057703
600

```

De abstracte member wordt dus gedefiniëerd door het abstract-keyword, en bestaat dus enkel uit de signatuur van die member. Public is hier opgenomen in de signatuur, omdat we wensen dat Area ook beschikbaar is in de publieke interface van die afgeleide types, en clients dus op die gespecialiseerde Figuur-objecten die member kunnen aanroepen.

Enkel implementatie-members (methods en properties) kunnen als abstract worden gedefinieerd. De herdefinitie (bijhorende override) moet voorzien in een nieuwe implementatie voor die member.

Een member zonder implementatie (zoals een field) kan ook moeilijk gebonden worden aan een nieuwe implementatie.

Private-members kunnen ook niet als abstract worden gemarkeerd.

Gezien er geen interface naar die member (aanwezigheid van die member in de interface) wordt overgeërfd, kan deze niet overgenomen interface niet geherdefiniëerd worden.

Enkel concrete afgeleide klassen moeten een implementatie voorzien van de overgeërfde abstracte member. Is de afgeleide klasse zelf abstract dan kan, maar hoeft deze klasse geen implementatie te voorzien.

Dat abstracte klassen die een abstracte member overerven geen implementatie moeten voorzien voor die member is niet zo vreemd.

Van dat afgeleide type zelf kan immers toch geen object worden aangemaakt waarop die members zou worden aangeroepen en waaraan dus een implementatie zou gekoppeld worden.

Dat concrete afgeleide klassen die een abstracte member overerven wel een implementatie moeten voorzien is ook logisch, want hiervan kunnen welk objecten worden aangemaakt waarop die member kan worden aangeroepen en waaraan dus wel een implementatie zou moeten worden gekoppeld.

Om een collectietype te creëren voor figuren (Figuren) kunnen we als volgt te werk gaan:

- Code

```

class Figuren
{
    private int _Count = 0;

    private Figuur[] _Items = new Figuur[0];

    public int GetCount()
    {
        return _Count;
    }

    public void Add(Figuur fig)
    {
        _Count++;
        Array.Resize(ref _Items, GetCount());
        _Items[GetCount() - 1] = fig;
    }
}

```

```

    public Figuur this[int index]
    {
        get { return _Items[index]; }
    }

    public int this[string kleur]
    {
        get
        {
            int aantal = 0;
            foreach (Figuur element in _Items)
            {
                if (element.Kleur == kleur)
                {
                    aantal++;
                }
            }
            return aantal;
        }
    }

    public double GetTotaleOppervlakte()
    {
        double totaleOppervlakte = 0;
        foreach (Figuur element in _Items)
        {
            totaleOppervlakte += element.GetOppervlakte();
        }
        return totaleOppervlakte;
    }
}

class Program
{
    static void Main(string[] args)
    {
        Rechthoek fig1 = new Rechthoek("red", 10, 20);
        Console.WriteLine(fig1.Kleur);
        Console.WriteLine(fig1.GetOppervlakte());
        //
        Cirkel fig2 = new Cirkel("yellow", 15);
        Console.WriteLine(fig2.Kleur);
        Console.WriteLine(fig2.GetOppervlakte());
        //
        Figuren figColl = new Figuren();
        figColl.Add(fig1);
        figColl.Add(fig2);
        figColl.Add(new Rechthoek("yellow", 20, 30));
        figColl.Add(new Rechthoek("yellow", 2, 60));
        //
        Console.WriteLine(figColl.GetCount());
        Console.WriteLine(figColl.GetTotaleOppervlakte());
        Console.WriteLine(figColl["yellow"].ToString());
        //
        Console.ReadLine();
    }
}

```

- Output

```

red
200
yellow
706,858347057703
4
1626,8583470577
3

```

Een array met elementen van type Figuur wordt in het collectietype gebruikt, daarin kunnen zowel de referenties van Rechthoek- als van Cirkel-objecten worden gestockeerd.

Bij het implementeren van de members `GetTotaleOppervlakte` en `this[string kleur]` kan men de elementen in die array overlopen. Van elk element in die array is men dan ook zeker dat een Oppervlakte-implementatie zal voorzien zijn.

Slecht 1 Add-method is noodzakelijk, aan de argument-variabele `figuur` kunnen immers zowel Rechthoek- als Cirkel-object-referenties worden doorgegeven.

Ook is er maar 1 indexer (`this`) nodig. De defenitie van deze property geeft aan een `Figuur`-expressie op te leveren, wat hier ook correct is, want zowel Rechthoek- als Cirkel-object-referenties zijn `Figuur`-expressies.

Je kunt in de client nu niet op een aanroep naar die Item-method gespecialiseerde members van de Rechthoek- of Cirkel-objecten die worden opgeleverd aanroepen. Het statisch type van de aanroep naar de Item-method is hier immers `Figuur` (als `Figuur` gedefinieerd).

Het grote voordeel van bovenstaand collectietype is de uitbreidbaarheid.

Bedenk maar eens hoe eenvoudig het is om een nieuw concreet type toe te voegen aan onze applicatie.

We zouden bijvoorbeeld een 'Vierkant'-klasse kunnen toevoegen, deze doen overerven van de 'Figuur'-klasse, hierdoor zullen we verplicht worden een 'GetOppervlakte'-implementatie te voorzien in 'Vierkant'. Maar hierdoor zijn we ook zeker dat objecten van die 'Vierkant'-klasse probleemloos kunnen worden toegevoegd aan zo'n collectie-object, zonder ook maar een aanpassing aan het collectietype te moeten aanbrengen.

Dit is dus een voordeel dat we bekomen door Liskov's Substitutie Principe.

Overall waar we een expressie van het basistype voor kunnen gebruiken, kan dit ook toegepast worden op een object van een afgeleid type (wanneer die expressie dus wijst naar een referentie van een object van afgeleid type).

Dit voordeel heeft dus op zich niets met abstracte basis klassen te maken.

Door hier een abstracte basis klasse `Figure` te gebruiken vermijden we wel dat geen specifieke, gewone figuren worden geïntantieerd en kunnen we de gemeenschappelijke members zelfs abstract definiëren waardoor we vereisen van de concrete afgeleide klassen dat ze hiervoor een implementatie voorzien.



21 INTERFACING

21.1 AANWEZIG VAN MEMBERS AFDWINGEN VIA ABSTRACTE KLASSEN

Abstract Base Classes (ABC's) kunnen we definiëren zodoende in de hiervan afgeleide klassen de aanwezigheid van bepaalde members af te dwingen.

Hiervoor maken we gebruik van Abstract members die verplicht overschreven moeten worden in de concrete (niet-abstracte) afgeleide klassen.

- Code

```
using System;

namespace Example
{
    abstract class Class1
    {
        public abstract void Command1();
        public abstract string Query1();
        public abstract string Property1 { get; set; }
    }

    class Class2 : Class1
    {
        public override void Command1()
        {
            Console.WriteLine("Class2.Command1()");
        }
        public override string Query1()
        {
            return "Class2.Query1()";
        }
        public override string Property1
        {
            get
            {
                return "Class2.Property1";
            }
            set
            {
                //...
            }
        }
    }

    class Class3 : Class1
    {
        public override void Command1()
        {
            Console.WriteLine("Class3.Command1()");
        }
        public override string Query1()
        {
            return "Class3.Query1()";
        }
        public override string Property1
        {
            get
            {
                return "Class3.Property1";
            }
            set
            {
                //...
            }
        }
    }
}
```

```

    }
}

class Program
{
    static void Main(string[] args)
    {
        Class1 object1 = new Class2();
        object1.Command1();
        Console.WriteLine(object1.Query1());
        Console.WriteLine(object1.Property1);
        //
        object1 = new Class3();
        object1.Command1();
        Console.WriteLine(object1.Query1());
        Console.WriteLine(object1.Property1);
        //
        Console.ReadLine();
    }
}

```

- Output

```

Class2.Command1 ()
Class2.Query1 ()
Class2.Property1
Class3.Command1 ()
Class3.Query1 ()
Class3.Property1

```

Eender welk object waarnaar een variabele gedeclareerd van het type Class1 kan wijzen zal nu zeker over implementaties beschikken voor:

- public void Command1()
- public string Query1()
- public string Property1

21.2 AANWEZIGHEID VAN MEMBERS AFDWINGEN VIA INTERFACES

Een alternatief voor het gebruik van abstracte basis klassen met enkel MustOverride-members is een "interface".

In een interfacetype definieer je de interface van de members waarvan je wenst dat in de klassen - die dat interfacetype "implementeren" - een implementatie wordt gekoppeld aan die memberinterface.

Een interfacetype (of dus een abstracte basis klasse met enkel MustOverride-members) is een soort van contract dat je definieert. Je definieert hierin dus aan wat voldaan moet zijn (in de afgeleiden of "implementeerders"), maar niet hoe dit afgehandeld dient te worden (dus niet de implementatie).

Een interface wordt geïmplementeerd (en niet overgeerfd). Een Implements-clausule wordt gebruikt om aan te geven dat een type implementaties zal/moet voorzien voor alle members uit die interface.

Ook bij implementatie wordt via een Implements-clausule aangegeven welke member uit het interfacetype wordt geïmplementeerd.

- Code

```

using System;

namespace Example
{
    interface Class1
    {
        void Command1();
    }
}

```

```

        string Query1();
        string Property1 { get; set; }
    }

    class Class2 : Class1
    {
        public void Command1()
        {
            Console.WriteLine("Class2.Command1()");
        }
        public string Query1()
        {
            return "Class2.Query1()";
        }
        public string Property1
        {
            get
            {
                return "Class2.Property1";
            }
            set
            {
                //...
            }
        }
    }

    class Class3 : Class1
    {
        public void Command1()
        {
            Console.WriteLine("Class3.Command1()");
        }
        public string Query1()
        {
            return "Class3.Query1()";
        }
        public string Property1
        {
            get
            {
                return "Class3.Property1";
            }
            set
            {
                //...
            }
        }
    }

    class Program
    {
        static void Main(string[] args)
        {
            Class1 object1 = new Class2();
            object1.Command1();
            Console.WriteLine(object1.Query1());
            Console.WriteLine(object1.Property1);
            //
            object1 = new Class3();
            object1.Command1();
            Console.WriteLine(object1.Query1());
            Console.WriteLine(object1.Property1);
            //
            Console.ReadLine();
        }
    }
}

```

- Output

```

Class2.Command1 ()
Class2.Query1 ()
Class2.Property1

```

```
Class3.Command1()  
Class3.Query1()  
Class3.Property1
```

Merk op dat enkel de signatuur van members wordt opgenomen in het interfacetype, en dat hierbij geen Access-modifier wordt vermeld. Over dit laatste in een volgend voorbeeld meer.

In alle volgende voorbeelden gaan we de naamgeving van een Interface aanpassen. De prefix "I" voor interfacetypes worden gebruikt. Overigens wordt er ook vaak een suffix "able" gebruikt in de identifiers van interfacetypes.

21.3 ABSTRACTE KLASSEN VERSUS INTERFACES

Een van de belangrijkste beslissingen die u bij het ontwerpen van uw programma moet nemen, is de keuze voor een interface of een abstracte klasse om zogenoemd polymorf gedrag in een objectmodel te krijgen. Polymorfisme staat voor veelvormigheid. Klassen zijn in dit verband gelijkvormig, maar hebben verschillende implementaties

De gelijkvormigheid betreft dan voornamelijk het gebruik en de naamgeving van methoden.

De keuze tussen het maken van een interface of een abstracte klasse ligt echter niet altijd voor de hand. Laten we eens kijken naar de verschillen tussen deze twee mogelijkheden en bepalen wanneer we de een en wanneer we de ander het beste kunnen gebruiken.

21.3.1 Interfaces

Interfaces definiëren een set van eigenschappen en methoden net als klassen. Maar in tegenstelling tot een klasse bevat een interface geen implementatie. Een interface kunt u beschouwen als een contract. Een klasse die een interface implementeert, moet ieder aspect van deze klasse implementeren, precies zoals in de interface is vastgelegd. De manier waarop dat aspect wordt ingevuld is echter niet vooraf bepaald. Een interface wordt gedeclareerd via het keyword interface. Een voorbeeld:

- Code

```
interface ILog  
{  
    void Write(string msg);  
    string Target { get; set; }  
}
```

Het enige wat we met deze code bereiken, is dat er klassen zouden kunnen zijn die de methode Write en de eigenschap Target kennen. Zo'n klasse kan bijvoorbeeld deze zijn:

- Code

```
public class TextLog : ILog  
{  
    public void Write(string msg)  
    {  
        FileInfo fi = new FileInfo(Target);  
        StreamWriter sw = fi.AppendText();  
        sw.WriteLine(DateTime.Now + " " + msg);  
        sw.Close();  
    }  
    public string Target { get; set; }  
}
```

De klassen FileInfo en StreamWriter komen uit de namespace System.IO. Het is dus van belang boven in het codebestand de instructie using System.IO; te plaatsen. De volledige code ziet er zo uit:

- Code

```

using System;
using System.IO;

namespace Example
{
    interface ILog
    {
        void Write(string msg);
        string Target { get; set; }
    }

    public class TextLog : ILog
    {
        public void Write(string msg)
        {
            FileInfo fi = new FileInfo(Target);
            StreamWriter sw = fi.AppendText();
            sw.WriteLine(DateTime.Now + " " + msg);
            sw.Close();
        }
        public string Target { get; set; }
    }

    class Program
    {
        static void Main(string[] args)
        {
            ILog log;
            log = new TextLog();
            log.Target = "C:\\temp\\test.log";
            log.Write("Tekstlogger via interface");
        }
    }
}

```

Wanneer u deze code uitvoert, wordt de tekst tekstlogger via interface in het bestand c:\temp\test.log geplaatst, met vooraan de datum en tijd. Zorg dat de map "temp" uiteraard bestaat!

U ziet dat het object log gedeclareerd wordt met de interface ILog. U kunt daarom ook alleen methoden en eigenschappen gebruiken die bekend zijn in deze interface. Wanneer u meer methoden toevoegt aan de klasse TextLog, worden deze niet herkend. De compiler zal dan ook een foutmelding geven als u ze toch wilt aanroepen in het object.

21.3.2 Verschillen tussen een interface en een klasse

Er zijn enkele kenmerkende verschillen tussen een interface en een klasse:

- Een interface wordt geïmplementeerd, niet uitgebreid.
- Een klasse kan meerdere interfaces implementeren.
- Een interface bevat geen data-declaraties, maar wel eigenschappen.
- Alle methode-declaraties in een interface zijn public.
- Een interface bevat geen implementatie-code.
- Een klasse die de interface implementeert moet voor ieder onderdeel van de interface een implementatie bevatten.

21.3.3 Abstracte klassen

Als alternatief voor een interface wordt vaak een abstracte klasse gebruikt. Een abstracte klasse kan niet geïntantieerd worden; alleen klassen die ervan afleiden kunnen dit. Een abstracte klasse kan een volledige

implementatie bevatten, maar meestal krijgt het een gedeeltelijke implementatie, of helemaal geen implementatie. Gemeenschappelijke functionaliteit wordt zo geencapsuleerd voor de ervende klassen.

Wanneer u een klasse hebt met methoden zonder implementatie (zogenaamde abstracte methoden), dan wordt ook de klasse abstract genoemd. Een abstracte methode is dan puur een aanduiding voor een methode die in een ervende klasse geïmplementeerd wordt. Abstracte methoden hebben tot doel om iedere klasse die erft van de abstracte klasse een implementatie af te dwingen. Als de ervende klasse dat niet doet, wordt ook deze klasse weer abstract genoemd en kan die dus niet geïnstantieerd worden.

We declareren een abstracte klasse met het sleutelwoord `abstract`.

- Code

```
abstract class LogBase
{
    abstract public void Write(string msg);
    public string Target { get; set; }
}
```

Een klasse die erft van de abstracte klasse `LogBase` ziet als volgt uit:

- Code

```
class SimpleLog : LogBase
{
    public override void Write(string msg)
    {
        FileInfo fi = new FileInfo(Target);
        StreamWriter sw = fi.AppendText();
        sw.WriteLine(DateTime.Now + " " + msg);
        sw.Close();
    }
}
```

De volledige code ziet er zo uit:

- Code

```
using System;
using System.IO;

namespace Example
{
    abstract class LogBase
    {
        abstract public void Write(string msg);
        public string Target { get; set; }
    }

    class SimpleLog : LogBase
    {
        public override void Write(string msg)
        {
            FileInfo fi = new FileInfo(Target);
            StreamWriter sw = fi.AppendText();
            sw.WriteLine(DateTime.Now + " " + msg);
            sw.Close();
        }
    }

    class Program
    {
        static void Main(string[] args)
        {
            SimpleLog log;
            log = new SimpleLog();
            log.Target = "C:\\temp\\test.log";
            log.Write("Tekstlogger via interface");
        }
    }
}
```

```
}
```

Het voordeel van zo'n abstracte klasse is dat we nu alleen maar de write-methode hoeven te implementeren. De eigenschap target is automatisch beschikbaar omdat SimpleLog erft van LogBase.

21.3.4 Klasse ontwerpen

Nu kent u het verschil tussen een interface en een abstracte klasse, maar waarom zou u een abstracte klasse willen gebruiken? In een klasse-ontwerp komt deze goed van pas. Wanneer u programmeert, maakt u namelijk geen gebruik van een algemene log-klasse, maar een specifieke implementatie ervan, bijvoorbeeld eentje die berichten in een tekstbestand plaatst. Maar het kan er ook een zijn die berichten in een database opslaat. Alles wat beide soorten log-klassen gemeenschappelijk hebben, kunnen we

in een abstracte klasse plaatsen, maar zo'n abstracte klasse kan niet alles. Het scheelt ook in de overzichtelijkheid van uw code wanneer u de benodigde functies in stukjes deelt. Doet u dit niet, dan loopt u het risico een veelkoppig monster te creëren.

Abstracte klassen zijn vooral handig wanneer u nog niet weet wat een klasse allemaal moet kunnen en op welke verschillende manieren de klasse zal worden gebruikt. U kunt dan de verwachte methoden al vastleggen, maar de implementatie van deze en andere methoden achterwege laten tot een specifieke implementatie nodig is.

Als er extra functionaliteit in de ervende klasse nodig is, kunnen we die toevoegen aan de basisklasse (baseclass) zonder de code te doen breken. Een klasse die erft van een abstracte klasse kan evenwel nog steeds interfaces implementeren.

Wanneer u een implementatie maakt van een abstracte klasse, moet u iedere abstracte methode in deze klasse implementeren. Iedere geïmplementeerde methode moet dezelfde, in aantal en type, parameters kennen. Ook moeten ze hetzelfde type teruggeven als gedeclareerd staat in de abstracte klasse. Visual C# biedt u overigens een snelle methode om een abstracte klasse te implementeren. Wanneer u de klasse declaratie begint, bijvoorbeeld zo:

- Code

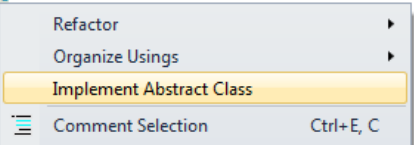
```
class SimpleLog : LogBase
```

dan kunt u met de rechtermuisknop op de tekst LogBase het contextmenu activeren en voor Implement Abstract Class kiezen. De verplichte methoden worden dat direct voor u gecreëerd. De invulling is uiteraard aan u.

```
abstract public void Write(string msg);
public string Target { get; set; }
}

class SimpleLog : LogBase
{
}

class Program
{
    static void Main(str
```



21.3.5 Wanneer Interfaces en wanneer Abstracte klassen?

- Als u verwacht dat er meerdere versies van een objecttype gemaakt zullen worden, creëer dan een abstracte klasse. Abstracte klassen bieden een eenvoudiger versiebeheer. Door het bijwerken van de basisklasse worden alle afgeleide klassen automatisch ook bijgewerkt. Interfaces daarentegen

kunnen niet aangepast worden nadat ze eenmaal zijn gemaakt. Als u een nieuwe versie van een interface wilt maken, moet u een geheel nieuwe interface maken.

- Als er functionaliteit van toepassing is op een hele reeks van diverse objecten, gebruik dan een interface. Abstracte klassen worden vooral gebruikt voor objecten die een nauwe relatie met elkaar hebben. Met interfaces ontwerpt u daarentegen gemeenschappelijke functionaliteit in een set van ongerelateerde klassen.
- Als u kleine, afgebakende functionaliteit ontwerpt, gebruik dan interfaces. Ontwerpt u eerder grotere functionele eenheden, gebruik dan liever een abstracte klasse.
- Als u wilt zorgen voor implementatie van gemeenschappelijke functionaliteit in alle implementaties van een type, kies dan voor een abstracte klasse. Met een abstracte klasse kunt u immers een klasse al deels implementeren, terwijl een interface voor geen enkele methode of eigenschap een implementatie bevat.

Een ezelsbruggetje dat kan helpen bij het maken van de juiste keuze is uzelf de vraag stellen of een klasse iets is of iets kan. Een klasse die een algemene vorm van iets is maakt u als een abstracte klasse. Wanneer u klassen maakt die iets moeten kunnen, dan implementeert u een interface.



DE WANDELENDE TAK...

22 ICOMPARABLE INTERFACE

22.1 SYSTEM.ICOMPARABLE

In de FCL (Framework Class Library) zitten reeds tal van voorgedefiniëerde interfacetypes. Deze zijn dus meteen klaar voor gebruik en worden zelfs al in andere types van de FCL gebruikt.

Een voorbeeld daarvan is de `IComparable`-interface uit de `System`-namespace. Deze wordt ondermeer gebruikt in de klassen `System.Array` en `System.Collections.ArrayList` waar men beschikt over `Sort`-methods.

Deze `Sort`-methods gaan een bepaald sorteringsprincipe toepassen op de elementen van de collectie. Bij numerieke inhoud van een `Array` of `ArrayList` zal `Sort` bijvoorbeeld de waarden van klein naar groot sorteren, bij `Dates` zal dit van oud naar nieuw zijn,

- Code

```
using System;
using System.Collections;

namespace Example
{
    class Program
    {
        static void Main(string[] args)
        {
            int[] values = { 3, 2, 1 };
            Array.Sort(values);
            foreach (int element in values)
            {
                Console.WriteLine(element.ToString() + " ");
            }
            //
            ArrayList arrayList1 = new ArrayList();
            arrayList1.Add(new DateTime(2013, 9, 3));
            arrayList1.Add(new DateTime(2013, 9, 2));
            arrayList1.Add(new DateTime(2013, 9, 1));
            arrayList1.Sort();
            foreach (DateTime element in arrayList1)
            {
                Console.WriteLine(element.ToString() + " ");
            }
            //
            Console.ReadLine();
        }
    }
}
```

- Output

```
1
2
3
1/09/2013 0:00:00
2/09/2013 0:00:00
3/09/2013 0:00:00
```

De manier waarop de elementen van een collectie gesorteerd worden is dus afhankelijk van het elementtype. Elk type die je definieert kan de `IComparable`-interface implementeren, om zo zelf te bepalen hoe elementen van dat type met elkaar vergeleken moeten worden.

Als we elementen van onze eigen user-defined types wensen te sorteren in bijvoorbeeld ondermeer een `Array` of `ArrayList`. Dan kunnen we dit doen door de `IComparable`-interface in ons type te implementeren.

Hiervoor hoeven we enkel een implementatie te voorzien voor: `int CompareTo(object obj)`.

Dit is een methode die een negatieve waarde, nul of positieve waarde oplevert naargelang het doorgegeven argument respectievelijk "groter", "gelijk" of "kleiner" is dan het object zelf (waarop de CompareTo(...) method wordt aangeroepen).

Door deze implementatie te voorzien kunnen methods als Sort (uit bijvoorbeeld de types Array, ArrayList, ...) twee elementen met elkaar vergelijken gebaseerd op de terugkeerwaarde van deze CompareTo(...).

Om elementen in een verzameling te sorteren, moet je immers in staat zijn twee elementen onderling met elkaar te vergelijken om te weten of het ene element voor of achter het andere element geplaatst moet worden.

- Code

```
using System;
using System.Collections;

namespace Example
{
    public class Temperature : IComparable
    {
        public int CompareTo(object obj)
        {
            Temperature otherTemperature = obj as Temperature;
            return this.value.CompareTo(otherTemperature.value);
        }

        public double value { get; set; }
    }

    class Program
    {
        static void Main(string[] args)
        {
            ArrayList temperatures = new ArrayList();
            // Initialize random number generator.
            Random rnd = new Random();

            // Generate 10 temperatures between 0 and 100 randomly.
            for (int ctr = 1; ctr <= 10; ctr++)
            {
                int degrees = rnd.Next(0, 100);
                Temperature temp = new Temperature();
                temp.value = degrees;
                temperatures.Add(temp);
            }

            // Before sort
            foreach (Temperature temp in temperatures)
                Console.Write(temp.value + " ");
            Console.WriteLine();

            // Sort ArrayList.
            temperatures.Sort();

            // After sort
            foreach (Temperature temp in temperatures)
                Console.Write(temp.value + " ");
            //
            Console.ReadLine();
        }
    }
}
```

- Output

```
21 7 96 87 97 19 61 94 41 66
7 19 21 41 61 66 87 94 96 97
```

In onderstaand voorbeeld gaan we de volgorde in een verzameling waarop elementen van het type `Figuur` moeten geplaatst worden baseren op de oppervlakte van deze figuren.

- Code

```
using System;
using System.Collections;

namespace Example
{
    abstract public class Figuur : IComparable
    {
        public int CompareTo(object obj)
        {
            Figuur other = obj as Figuur;
            return this.Oppervlakte().CompareTo(other.Oppervlakte());
        }

        abstract public double Oppervlakte();
    }

    public class Rechthoek : Figuur
    {
        public Rechthoek(int breedte, int hoogte)
        {
            this.Breedte = breedte;
            this.Hoogte = hoogte;
        }

        public int Breedte { get; set; }
        public int Hoogte { get; set; }

        public override double Oppervlakte()
        {
            return Breedte * Hoogte;
        }
    }

    public class Circle : Figuur
    {
        public Circle(int straal)
        {
            this.Straal = straal;
        }

        public int Straal { get; set; }

        public override double Oppervlakte()
        {
            return Straal * Straal * Math.PI;
        }
    }

    class Program
    {
        static void Main(string[] args)
        {
            Figuur fig1 = new Rechthoek(5, 3);
            Figuur fig2 = new Circle(6);
            Figuur fig3 = new Circle(9);
            Figuur fig4 = new Rechthoek(6, 2);
            Figuur fig5 = new Rechthoek(6, 2);

            Figuur[] figuren1 = { fig1, fig2, fig3, fig4 };

            foreach (Figuur fig in figuren1)
                Console.Write(fig.Oppervlakte() + " ");
            Console.WriteLine();

            Array.Sort(figuren1);

            foreach (Figuur fig in figuren1)
                Console.Write(fig.Oppervlakte() + " ");
        }
    }
}
```

```

        Console.WriteLine();

        Console.WriteLine(fig1.CompareTo(fig2));
        Console.WriteLine(fig2.CompareTo(fig1));
        Console.WriteLine(fig4.CompareTo(fig5));

        Console.ReadLine();
    }
}

```

- Output

```

15 113,097335529233 254,469004940773 12
12 15 113,097335529233 254,469004940773
-1
1
0

```

- Code

```

int CompareTo(Object obj) {
    // return Less than zero if this object
    // is less than the object specified by the CompareTo method.

    // return Zero if this object is equal to the object
    // specified by the CompareTo method.

    // return Greater than zero if this object is greater than
    // the object specified by the CompareTo method.
}

```

22.2 OEFENINGEN

22.2.1 Oefening 1

Zorg ervoor dat in een collectie van Counter-objecten, de objecten op huidige value kan gesorteerd worden.

22.2.2 Oefening 2

Creëer een Reservation-klasse met een StartDate en EndDate eigenschap.

Een array van objecten van het type Reservation zou sorteerbaar moeten zijn met de Array.Sort(...)-method.

Reservaties moeten gesorteerd worden eerst oplopend (van oud naar nieuw) volgens startdatum, is deze gelijk dan oplopend (van oud naar nieuw) volgens einddatum.

22.2.3 Oefening 3

Maak een systeem waarbij verschillende soorten "calculations" ("berekeningen") kunnen worden gemaakt, voorzie minimaal de mogelijkheid om een "division" ("deling") en een "multiplication" ("vermenigvuldiging") te creëren.

Alle berekeningen hebben gemeenschappelijk dat ze een bepaald resultaat ("quotient", "product", ...) kunnen opleveren en dat ze herhaald kunnen worden ("divide by", "multiply by", ...).

