

Zie Scherp

Een C#-cursus

Table of Contents

| | |
|----------------|-------|
| FrontMatter | 1.1 |
| Inleiding | 1.2 |
| Benodigdheden | 1.2.1 |
| Nuttige extras | 1.2.2 |

Semester 1

| | |
|-----------------------------------|-------|
| H0: Werken met Visual Studio | 2.1 |
| Introductie tot C# | 2.1.1 |
| Werken met Visual Studio | 2.1.2 |
| Je eerste programma | 2.1.3 |
| De essentie van C# | 2.1.4 |
| H1: Variabelen en datatypes | 2.2 |
| Datatypes | 2.2.1 |
| Input verwerken met ReadLine | 2.2.2 |
| Strings en chars | 2.2.3 |
| Strings samenvoegen | 2.2.4 |
| Oefeningen | 2.2.5 |
| H2: Werken met data | 2.3 |
| Expressies | 2.3.1 |
| Operators | 2.3.2 |
| Casting, conversie en parsing | 2.3.3 |
| Random | 2.3.4 |
| Oefeningen | 2.3.5 |
| H3: Beslissingen | 2.4 |
| Beslissingen intro | 2.4.1 |
| If | 2.4.2 |
| Logische en relationele operators | 2.4.3 |
| Switch | 2.4.4 |
| Enums | 2.4.5 |
| Oefeningen | 2.4.6 |
| H4: Loops | 2.5 |
| Loops intro | 2.5.1 |
| While en Do While | 2.5.2 |
| For | 2.5.3 |
| Foreach en var | 2.5.4 |
| Programma flow analyse | 2.5.5 |
| Oefeningen | 2.5.6 |
| H5: Methoden | 2.6 |
| Methoden intro | 2.6.1 |
| Bibliotheken (Math.Pow) | 2.6.2 |
| Out en Ref parameters | 2.6.3 |
| Geavanceerde methoden | 2.6.4 |
| Oefeningen | 2.6.5 |
| H6: Arrays | 2.7 |
| Array intro | 2.7.1 |
| Array principles | 2.7.2 |

| | |
|-----------------------------------|-------|
| Werken met arrays | 2.7.3 |
| Arrays en methoden | 2.7.4 |
| N-dimensionale arrays | 2.7.5 |
| Jagged arrays | 2.7.6 |
| Oefeningen | 2.7.7 |
| All-In-Projecten | 2.8 |
| Overzicht | 2.8.1 |
| Console Matrix | 2.8.2 |
| Ascii filmpjes maken met loops | 2.8.3 |
| Ascii filmpjes maken met methoden | 2.8.4 |
| Tekst-gebaseerd Maze game | 2.8.5 |

Semester 2

| | |
|--------------------------------|-------|
| H7: Klassen en objecten | 3.1 |
| OOP Intro | 3.1.1 |
| Constructors | 3.1.2 |
| Properties | 3.1.3 |
| Autoproperties | 3.1.4 |
| Static | 3.1.5 |
| Expression bodied members | 3.1.6 |
| Labo-oefeningen 1 | 3.1.7 |
| Labo-oefeningen 2 | 3.1.8 |
| Labo-oefeningen 3 | 3.1.9 |
| H8: Geheugennmanagement | 3.2 |
| Stack en Heap | 3.2.1 |
| Arrays van objecten | 3.2.2 |
| Labo-oefeningen | 3.2.3 |
| H9: Overerving | 3.3 |
| Overerving intro | 3.3.1 |
| Virtual en override | 3.3.2 |
| Base keyword | 3.3.3 |
| Constructors bij overerving | 3.3.4 |
| System.Object | 3.3.5 |
| Abstract | 3.3.6 |
| Labo-oefeningen | 3.3.7 |
| H11: Compositie | 3.4 |
| Compositie intro | 3.4.1 |
| Labo-oefeningen | 3.4.2 |
| H12: Polymorfisme | 3.5 |
| Intro polymorfisme | 3.5.1 |
| Labo-oefeningen | 3.5.2 |
| H13: Interfaces | 3.6 |
| Interface intro | 3.6.1 |
| Interfaces in de praktijk | 3.6.2 |
| Labo-oefeningen | 3.6.3 |
| H14: Generics | 3.7 |
| Generics methoden en types | 3.7.1 |
| Generic classes en constraints | 3.7.2 |
| Collections | 3.7.3 |
| Labo-oefeningen | 3.7.4 |

| | |
|--|-------|
| H15: Is en As keyword | 3.8 |
| Is en As keywords | 3.8.1 |
| Polymorfisme en interfaces | 3.8.2 |
| Objecten vergekijken: alles komt samen | 3.8.3 |
| Labo-oefeningen | 3.8.4 |
| All-In-Projecten | 3.9 |
| Overzicht | 3.9.1 |
| OO Textbased Game | 3.9.2 |
| War Simulator | 3.9.3 |
| Map Maker | 3.9.4 |

Appendix

| | |
|--------------------------|-----|
| Ea-ict coding guidelines | 4.1 |
|--------------------------|-----|

Zie scherp

Deze cursus is nog **zwaar onder opbouw** en zal de komende weken en maanden (hopelijk) steeds meer vorm krijgen. De cursus zal gebruikt worden als handboek binnen de opleiding professionele bachelor elektronica-ict van de AP Hogeschool. Concreet zal dit het handboek voor de eerste 2 semesters omtrent 'leren programmeren met C#' worden:

- Deel 1: Programming Principles, eerste semester
- Deel 2: Object Oriented Programming, tweede semester

In de eerste fase is deze cursus een verzameling van eerder geschreven teksten (grotendeels van m'n voormalige "cursus-omgeving" [Code van 1001 nacht](#)) die ik nu stelselmatig zal samenvoegen tot een meer coherent geheel.

Indien bepaalde hoofdstukken of onderdelen niet duidelijk zijn of fouten bevatten, aarzel dan niet om me te contacteren. Alle feedback is **zéér welkom**.

De cursus is volledig 'opensource' en je kan deze dus ook zelf forken via [github](#), maar indien je briljante uitbreidingen schrijft verwacht ik uiteraard wel dat je me vermeld en een epic merge requests stuurt ;)

Veel leer-en leesplezier, Tim Dams

PS Besef dat goed kunnen programmeren enkel kan (aan)geleerd worden indien je ook effectief veel programmeert. Je kan ook niet leren fietsen door enkel een handboek "Fietsen met Merckx" te lezen, je zal op de fiets moeten springen! En vallen...véél vallen.

Todo's voor de auteur:

- Nieuwe => syntax voor properties (uiteleggen bij props én override bij interface en virtual)

Benodigheden

In alle lessen (hoorcollege en practica) hebben we 2 zaken nodig:

1. Deze cursus
2. Een laptop met daarop Visual Studio 2017 Enterprise edition geïnstalleerd. Enterprise (aanbevolen) kan je via [Dreamspark](#) downloaden.

Opgelet: VERGEET NIET DE KEY TE BEWAREN BIJ HET (GRATIS) BESTELLEN.

Leren programmeren door enkele de opdrachten in deze cursus te maken zal je niet ver (genoeg) brengen. Onze dikke vriend het Internet heeft echter tal van schitterende bronnen. Hier een overzicht.

Cheat sheet

Volgende document bevat een overzicht van de basis C# zaken van het eerste en (deel van het) tweede semester: [download hier](#)

Game-based programmeren

Ideale manier om programmeren meer in de vingers te krijgen:

- [Code Combat](#) (aanrader!)
 - [Pex For Fun](#) (specifiek voor C#!)
 - [Code Academy](#)
 - [RPG Game in C#](#) (behandelt leerstof van volledig eerste jaar en meer)

Tutorials

- [Online video c# cursus](#): Zeer aan te raden indien je een bepaald concept uit de les niet begrijpt.
 - [C-sharp.be](#) : Nederlandstalige cursus met veel toffe oefeningen waarvan je sommige zelfs in deze cursus zal terugvinden.
 - [Microsoft Virtual Academy](#): Microsoft heeft een virtual academy cursus "C# fundamentals" uitgebracht. Ik kan deze cursus zeer erg aanbevelen.
 - [Rob Miles's The C# Programming Yellow book](#): Zeer vermakelijk, vlot geschreven C# boek(je)
 - [Code van 1001 Nacht](#) : Hier plaats ik geregeld korte tutorials over een specifiek probleem/onderwerp omtrent C#. De meeste content daar werd reeds in deze cursus verwerkt.

• Oefenvragen

- Een lijst met oude oefenvragen uit 2010: nog steeds relevant.
 - Een dagelijkse programmeeruitdaging op reddit
 - Pittige vragen van de jaarlijkse Vlaamse Programmeerwedstrijd
 - 2013
 - 2014
 - 2015

Het algoritme

De essentie van een computerprogramma is het zogenaamde **algoritme** (het "recept" zeg maar). Dit is de reeks instructies die het programma moet uitvoeren telkens het wordt opgestart. Het algoritme van een programma moet je zelf verzinnen. De volgorde waarin de instructies worden uitgevoerd zijn uiteraard zeer belangrijk. Dit is exact hetzelfde als in het echte leven: een algoritme om je fiets op te pompen kan zijn:

1. Haal dop van het ventiel
2. Plaats pomp op ventiel
3. Begin te pompen

Eender welke andere volgorde van bovenstaande algoritme zal vreemde (en soms fatale) fouten geven.

C-Sharp

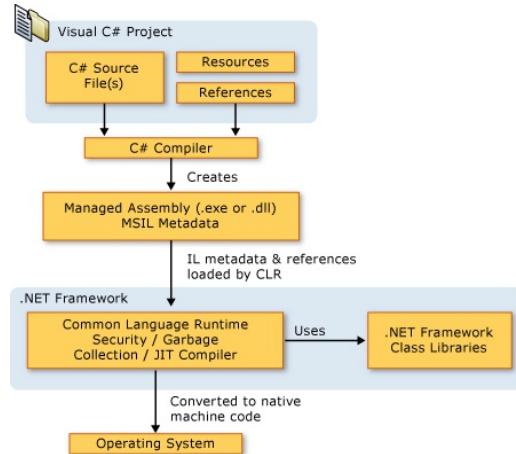
Om een algoritme te schrijven dat onze computer begrijpt dienen we een programmeertaal te gebruiken. Net zoals er ontelbare spreektaLEN in de wereld zijn, zijn er ook vele programmeertalen. **C# ('siesjarp')** is er een van de vele. In tegenstelling tot onze spreektaLEN moet een computertaAL 'exact' zijn en moet het op ondubbelzinnige manier door de computer verstaan worden. C# is een taal die deel uitmaakt van de .NET omgeving die meer dan 15 jaar geleden door Microsoft werd ontwikkeld (juli 2000).

De geschiedenis en de hele .NET-wereld vertellen zou een cursus op zich betekenen en gaan we hier niet doen. Het is nuttig om weten dat er een gigantische bron aan informatie over .NET en C# online te vinden is, beginnende met docs.microsoft.com

Microsoft .NET

Bij de geboorte van .NET in 2000 zat dus ook de taal C#. .NET is een zogenaamd **framework**. Dit framework bestaat uit een grote groep van bibliotheken (**class libraries**) en een *virtual execution system* genaamd de **Common Language Runtume (CLR)**. De CLR zal ervoor zorgen dat C#, of andere .NET talen (F#, VB.NET,etc), kunnen samenwerken met de vele bibliotheken.

Om een uitvoer bestand te maken (**executable**, vandaar de extensie .exe bij uitvoerbare programma's in windows) zal de broncode die je hebt geschreven in C# worden omgezet naar een **Intermediate Language (IL)** code. Op zich is deze IL code nog niet uitvoerbaar, maar dat is niet ons probleem. Wanneer een gebruiker een in IL geschreven bestand zal willen uitvoeren dan zal, achter de schermen, deze code ogenblikkelijk door de CLR naar machine code omzetten (**Just-In-Time** of JIT compilatie) en uitvoeren. De gebruiken zal dus nooit dit proces merken (tenzij hij geen .NET framework heeft geïnstalleerd).



(Bron afbeelding)

Merk op dat we veel details van de compiler achterwege laten hier. De compiler is een uitermate complex element , maar in deze fase van je (prille) programmeursleven hoeven we enkel de kern van de compiler te begrijpen: **het omzetten van C# code naar een uitvoerbaar bestand geschreven in IL code.**

Deze cursus heeft als doel om je de programmeertaal C# aan te leren. Terwijl we dit doen zullen we ook geregeld andere .NET framework gerelateerde zaken aanraken.

De compiler en Visual Studio

Jouw taak als **programmeur** in deze cursus is algoritmes in C# taal uitschrijven. We zouden dit in een eenvoudige tekstverwerker kunnen doen, maar dan maken we het onszelf lastig. Net zoals je tekst in notepad kunt schrijven, is het handiger dit in bijvoorbeeld Word te doen: je krijgt een spellingchecker en allerlei handige extra's. Ook voor het schrijven van computer code is het handiger om een zogenaamde IDE te gebruiken, een omgeving die ons zal helpen foutloze C# code te schrijven. Visual Studio is zo een IDE (Integrated Development Environment) die we in het vorige hoofdstuk reeds hebben besproken.

Het hart van Visual Studio bestaat uit de **compiler**. De compiler zal je C# code omzetten naar de ILcode zodat jij (of anderen) je applicatie op een computer (of ander apparaat) kan gebruiken. Zolang de C# niet exact voldoet aan de C# syntax (zie verder) zal de compiler het vertikken een uitvoerbaar bestand voor je te genereren.

Kennismaken met C# en Visual Studio

Het grootste deel van het leven van een beginnende software-ontwikkelaar ("programmeur") vindt plaats in een ontwikkelomgeving. Dit is de plek waar je je computercode zal schrijven, die vervolgens zal omgezet worden naar een uitvoerbare applicatie.

Visual Studio (VS) is een pakket dat een groot deel tools samenvoegt (debugger, code editor, compiler, etc) zodat je niet tientallen paketten moet gebruiken om software te schrijven.



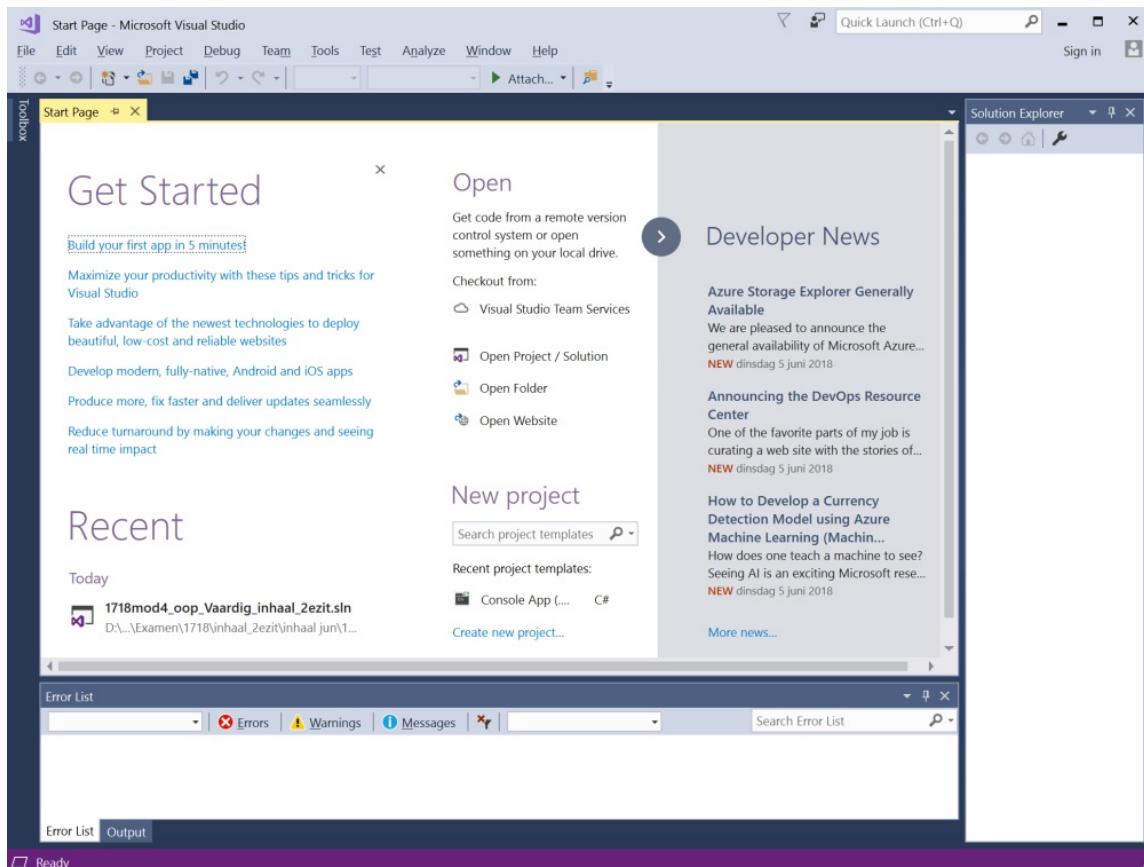
Visual Studio is een zogenaamde IDE("Integrated Development Environment") en is op maat gemaakt om C#.NET applicaties te ontwikkelen. Je bent echter verre van verplicht om enkel C# applicaties in VS te ontwikkelen, je kan gerust VB.NET, TypeScript, Python en andere talen gebruiken.

Visual Studio Code - de lightweight VS

Visual Studio vindt eindelijk ook z'n weg op andere platformen. Zoek je een lightweight versie dan moet je zeker eens [Visual Studio Code](#) eens proberen.

Visual studio opstarten

Na het opstarten van VS krijg je het startvenster te zien van waaruit je verschillende dingen kan doen, zoals eerder aangemaakte projecten opstarten, informatie opzoeken of een totaal nieuw project starten.



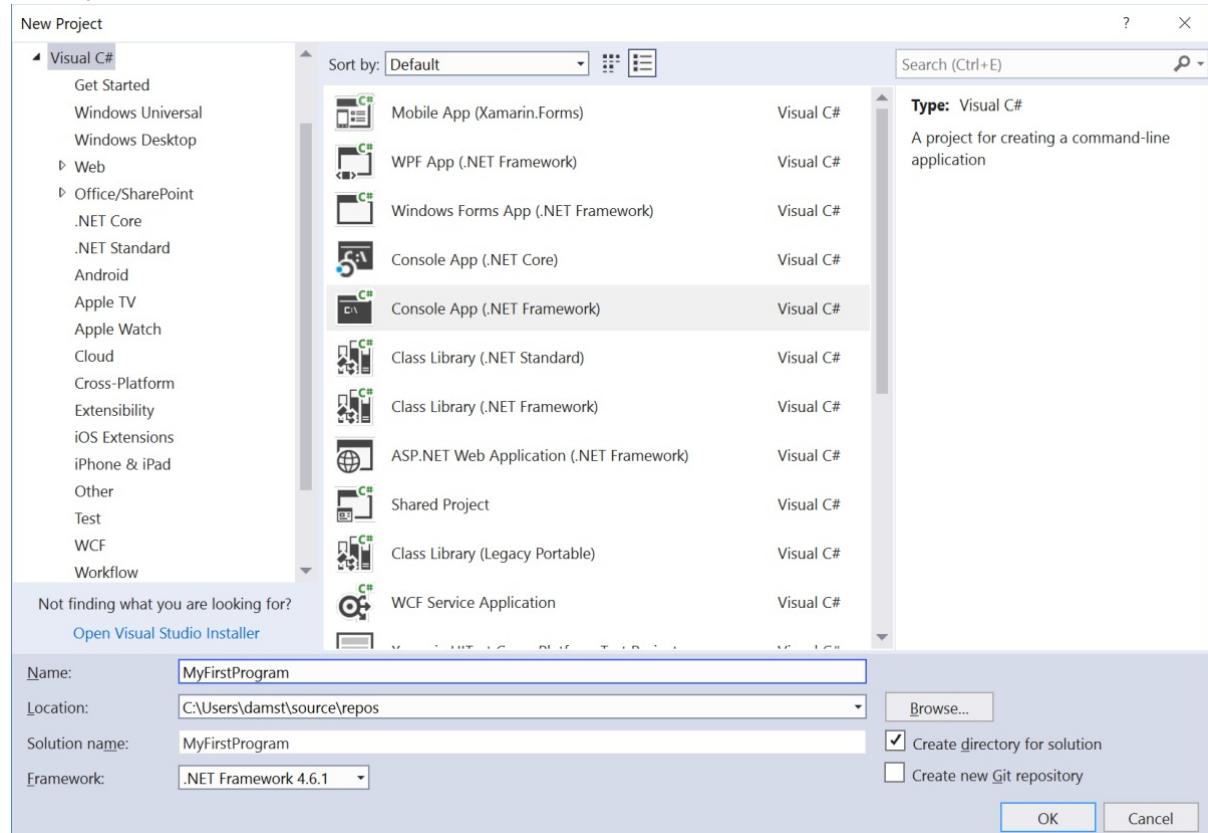
Een nieuw project aanmaken

We zullen nu een nieuw project aanmaken. Dit kan op verschillende manieren. De eenvoudigste manier is door te klikken op File -> New -> Project...

Het "New Project" venster dat nu verschijnt geeft je hopelijk al een glimsp van de veelzijdigheid van VS. In het linkerdeel zie je bijvoorbeeld alle Project Types staan. M.a.w. dit zijn alle soorten programma's die jan kan maken in VS. Naargelang de geïnstalleerde opties en bibliotheken zal deze lijst groter of kleiner zijn.

In ons geval kiezen we als Project Type **'Visual C#'** en als template **Console Application (.NET Framework)**. Onderaan kan je een naam geven voor je project alsook de locatie op de harde schijf waar het project dient opgeslagen te worden. **Onthoudt waar je je project aanmaakt zodat je dit later terugvindt.**

Geeft volgende informatie op:



Druk ok.

VS heeft nu reeds een aantal bestanden aangemaakt die je nodig hebt om een 'Console Applicatie' te maken. Een console applicatie is een programma dat alle uitvoer naar een zogenaamde 'console' stuurt, een shell. Maw, je kan enkel tekst (ASCII/Unicode) als uitvoer genereren en dus geen multimedia elementen zoals afbeeldingen, geluid, etc.

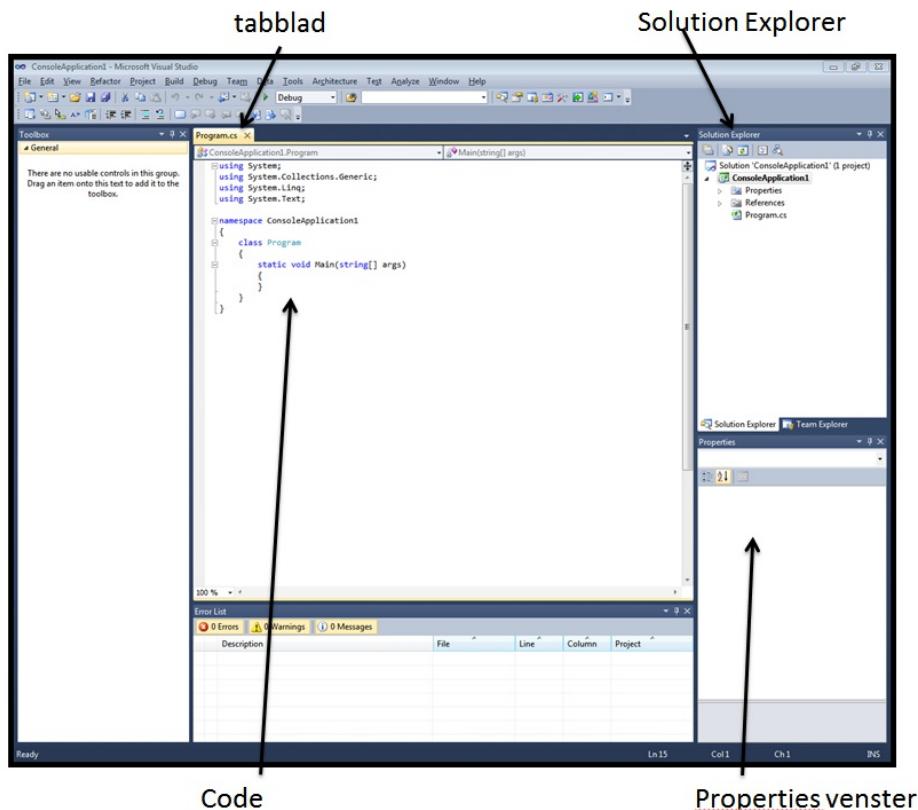
Goed nakijken

Kijk steeds goed volgende zaken na voor je je nieuwe project aanmaakt:

- Kies **Visual C#** als taal (en niet Visual Basic ofzo)
- Kies **Console Application** (en niet Windows Form Application ofzo)
- Zet al je projecten op een vaste locatie op je hd (of nog beter op je dropbox o.i.d.)

IDE Layout

We zullen nu eerst eens bekijken wat we allemaal zien in VS na het aanmaken van een nieuw programma.

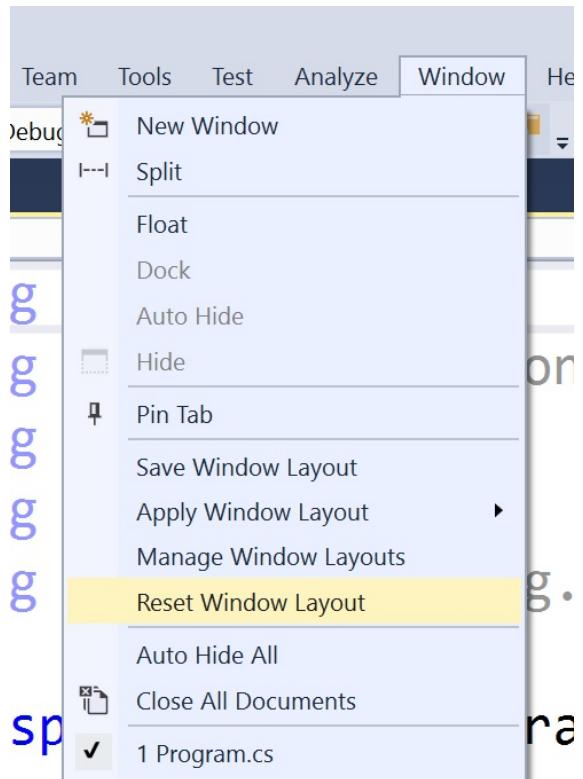


- Je kan meerdere bestanden tegelijkertijd openen in VS. Ieder bestand zal z'n eigen tab krijgen. De actieve tab is het bestand wiens inhoud je in het hoofdgedeelte eronder te zien krijgt.
- De "solution explorer" toont alle bestanden en elementen die tot het huidige project behoren. Als we dus later nieuwe bestanden toevoegen dan kan je die hier zien (en openen).
- Het **properties** venster (eigenschappen) is een belangrijk venster. Hier komen alle eigenschappen van het huidige geselecteerde element. Selecteer bijvoorbeeld maar eens Program.cs in de solution explorer en merk op dat er allerlei eigenschappen getoond worden. Onderaan het Properties venster wordt steeds meer informatie getoond over de huidig geselecteerde eigenschap.

Layout kapot/kwijt/vreemd

Het gebeurt al eens dat je layout een beetje rommelig is.

- Voor eenvoudige venster terug te krijgen, bijvoorbeeld het properties window of de solution explorer: klik bovenaan in de menubalk op "View" en kies dan het gewenste venster (soms staat dit in een submenu)
- Je kan ook altijd je layout in z'n geheel resetten: Ga naar "Window" en kies "Reset window layout".



Je programma starten

De code die VS voor je heeft gemaakt is reeds een werkend programma. Je kan de code compileren naar een uitvoerbaar bestand door te klikken op Build->Build Solution. Als dit gelukt is zal er onderaan VS in de statusbar 'Build succeeded' verschijnen.

Je kan nu je gecompileerde bestand uitvoeren door te kiezen voor Debug->Start without debugging (of door te drukken op ctrl+F5).

```
C:\Windows\system32\cmd.exe
Press any key to continue . . .
```

Veel doet je programma nog niet natuurlijk, dus sluit dit venster maar terug af door een willekeurige toets in te drukken.

Console-applicaties

Een console-applicatie is een eenvoudig programma dat zijn in- en uitvoer via een klassiek commando-scherm toont. Via 2 belangrijke methoden `Console.WriteLine` en `Console.ReadLine` kunnen we enerzijds tekst op het scherm tonen en anderzijds tekst dat de gebruiker intypt inlezen.

Een console-applicatie draait dus in dezelfde omgeving als wanneer we in Windows een command-prompt openen (via Start-> Uitvoeren-> cmd [enter])

Je eerste programma

Maak een nieuw console-project aan (noem het Demo1) en open het Program.cs bestand (indien het niet open).

Voeg binnen de accolades van `Main` volgende zin toe:

```
System.Console.WriteLine("Hoi, ik ben het!");
```

Zodat je dus volgende code krijgt (:

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace Demo1
{
    class Program
    {
        static void Main(string[] args)
        {
            Console.WriteLine("Hoi, ik ben het!");
            Console.ReadKey();
        }
    }
}
```

Compileer deze code en voer ze uit: druk hiervoor op het groene driehoekje met "Start" naast. Of via het menu Debug en dan Start Debugging.

Analyse van de code

We gaan nu iedere lijn code kort bespreken. Sommige lijnen code zullen lange tijd niet belangrijk zijn. Onthoud nu alvast dat **alle belangrijke code staat tussen de accolades onder de lijn `static void Main(string[] args)` !**

- `using System;` : Alle `Console`-commando's die we verderop gebruiken zitten in de `System` bibliotheek. Als we deze lijn (een zogenaamde **directive**) niet zouden schrijven dan moesten we `System.Console.WriteLine` i.p.v. `Console.WriteLine` schrijven verderop in de code.
- Andere `using` directives: standaard zet Visual Studio de meest gebruikte bibliotheken reeds klaar. Veeg ze gerust weg als je zeker bent dat je bijvoorbeeld niets met Linq gaat doen. Bij twijfel: laten staan!
- `namespace Demo1` : Dit is de unieke naam waarbinnen we ons programma zullen steken, en het is niet toevallig de naam van je project. Verander dit nooit tenzij je weet wat je aan het doen bent.
- `class Program{}` : Hier start je echte programma. Alle code binnen deze Program accolades zullen gecompileerd worden naar een uitvoerbaar bestand.
- `static void Main(string[] args)` : Het startpunt van iedere console-applicatie. Wat hier gemaakt wordt is een **methode** genaamd `Main`. Je programma kan meerdere methoden (of functies) bevatten, maar enkel degene genaamd `Main` zal door de compiler als het startpunt van het programma gemaakt worden.
- `Console.WriteLine("Hoi, ik ben het!");` : Dit is een **statement** dat de `WriteLine`-methode aanroeft van de `Console`-klasse. Het zal alle tekst die tussen de aanhalingstekens staat op het scherm tonen.
- `Console.ReadLine();` : Een truukje (zie hierna) dat het programma zal pauzeren tot de gebruiker op een toets heeft gedrukt.

LEES MIJ! (veel voorkomende fout)

De lijn met `ReadKey` is een klein truukje: ReadKey verwerkt input van de gebruiker maar in het voorgaande stuk code doen we niets met die input. Test echter eens wat er gebeurt als je de lijn `Console.ReadKey();` weglaat en dan probeert te starten.

Inderdaad het programma sluit zich ogenblikkelijk af en je hebt amper tijd om te lezen wat er op het scherm verscheen. Dankzij ReadKey bouwen we dus een "pauze/wachtpunt" in aan het einde van ons programma. Dit is een goede gewoonte als je console-applicaties aan het debuggen bent.

WriteLine: Tekst op het scherm

De WriteLine-methode is een veelgebruikte methode in Console-applicaties. Het zorgt ervoor dat we tekst op het scherm kunnen tonen.

Voeg volgende lijn toe na de vorige WriteLine-lijn in je project:

```
Console.WriteLine("Wie ben jij?");
```

De WriteLine methode zal alle tekst tonen die tussen de " " staan tussen de haakjes van de methode. **De aanhalingstekens aan het begin en einde van de tekst zijn uiterst belangrijk! Alsook het kommapunt helemaal achteraan.**

Je programma is nu:

```
namespace Demo1
{
    class Program
    {
        static void Main(string[] args)
        {
            Console.WriteLine("Hoi, ik ben het!");
            Console.WriteLine("Wie ben jij?!");
            Console.ReadLine();
        }
    }
}
```

Oh boy...Wat was dit allemaal?! We hebben al aardig wat vreemde code zien passeren en het is niet meer dan normaal dat je nu denkt "dit ga ik nooit kunnen". Wees echter niet bevreesd: je zal sneller dan je denkt bovenstaande code als 'kinderspel' gaan bekijken. Een tip nodig? Test en experimenteer met wat je al kunt!

Statements en de C# syntax

Om een werkend C#-programma te maken moeten we de C#-taal beheersen. Net zoals iedere taal bestaat ook C# uit enerzijds grammatica, in de vorm van de **C# syntax** en de te gebruiken keywords.

Een C#-programma bestaat uit een opeenvolging van instructies ook wel **statements** genoemd. **Deze eindigen steeds met een kommapunt (;)** (zoals ook in het Nederlands een zin eindigt met een punt).

De volgorde van de woorden (keywords, variabelen, etc.) zijn niet vrijblijvend en moeten aan (grammaticale) regels voldoen. Enkel indien alle statements correct zijn zal het programma gecompileerd worden naar een werkend en uitvoerbaar programma (zoals in vorige sectie besproken).

Enkele belangrijke regels van C#:

- **Hoofdletter-gevoelig:** C# is hoofdlettergevoelig. Dat wil zeggen dat hoofdletter `T` en lowercase `t` totaal verschillende zaken zijn voor C#.
- **Statements afsluiten met kommapunt:** Ieder C# statement wordt afgesloten moet een kommapunt `;`.
- **Witruimtes:** Spaties, tabs en enter worden door de C# compiler genegeerd. Je kan ze dus gebruiken om de layout van je code (*bladspiegel* zeg maar) te verbeteren.
- **Commentaar toevoegen kan:** met behulp van `//` voor een enkele lijn en `/* */` voor meerdere lijnen commentaar. Alles dat in commentaar staat zal door de compiler genegeerd worden.

Keywords: de woordenschat

C# bestaat zoals gezegd uit enkel grammaticale regels. Grammatica zonder woordenschat is nutteloos. Er zijn binnen C# dan ook 80 woorden, zogenaamde **reserved keywords** gereserveerd die de woordenschat voorstellen. In deze cursus zullen we stelselmatig deze keywords leren kennen en gebruiken op een correcte manier om zo werkende code te maken.

Deze keywords zijn: `abstract|as|base|bool|break|byte|case|catch|char|checked|class|const|continue|decimal|default|delegate|do|double|else|enum|event|explicit|extern|false|finally|fixed|float|for|foreach|goto|if|implicit|in|int|interface|internal|is|lock|long|namespace|new|null|object|operator|out|override|params|private|protected|public|readonly|ref|return|sbyte|sealed|short|sizeof|stackalloc|static|string|struct|switch|this|throw|true|try|typeof|uint|ulong|unchecked|unsafe|ushort|using|using static|virtual|void|volatile|while|`

De keywords in vet zijn keywords die we dit semester zullen kennen. Die in cursief in het tweede semester.

Variabelen, identifiers en naamgeving

We hebben variabelen nodig om (tijdelijke) data in op te slaan. Wanneer we een statement schrijven dat bijvoorbeeld input van de gebruiker moet vragen, dan willen we ook die input bewaren, zodat we verderop in het programma (het algoritme) iets met deze data kunnen doen. We doen hetzelfde in ons hoofd wanneer we bijvoorbeeld zeggen "tel 3 en 4 op en vermenigvuldig dat resultaat met 5". Eerst zullen we het resultaat van $3+4$ in een variabele moeten bewaren. Vervolgens zullen we de inhoud van die variabele vermenigvuldigen met 5 en dat nieuwe resultaat ook in een nieuwe variabele opslaan (om vervolgens bijvoorbeeld naar het scherm te sturen).

Wanneer we een variabele aanmaken zal deze moeten voldoen aan enkele afspraken. Zo moeten we minstens 2 zaken meegeven:

- Het type van de variabele: het **datatype** dat aangeeft wat voor data we wensen op te slaan (tekst, getal, afbeelding, etc)
- De naam van de variabele: de **identifier** waarmee we snel aan de variabele-waarde aankunnen

De verschillende datatypes bespreken we in een volgende hoofdstuk.

Regels voor identifiers

De code die we gaan schrijven moet voldoen aan een hoop regels. Wanneer we in onze code zelf namen (**identifiers**) moeten geven aan **variabelen** (en later ook methoden, objecten, etc) dan moeten we een aantal regels volgen:

- Hoofdlettergevoelig: de identifiers `tim` en `Tim` zijn verschillend. Je mag dus perfect twee verschillende variabelen aanmaken met deze name.
- Geen keywords: identifiers mogen geen gereserveerd C# keyword zijn. De keywords van hierboven mogen dus niet. Varianten wel: denk maar aan `goto` (`goto` is een gereserveerd keyword, maar dankzij de hoofdlettergevoeligheid is dit dus toegelaten) en `int`

- (tegenover keyword `int`)
- Eerste karakter-regel: het eerste karakter van de identifier mag enkel zijn:
 - kleine of grote letter
 - liggend streepje (`_`)
 - Alle andere karakters: de overige karakters moeten mogen enkel zijn:
 - kleine of groter| letter
 - liggend streepje
 - een cijfer (`1` tot en met `9` en `0`)
 - Lengte: een legale identifier mag zo lang zijn als je wenst, maar hou het best leesbaar

Naamgeving afspraken

Er zijn geen vaste afspraken over hoe je je variabelen moet noemen toch hanteren we enkele **coding guidelines** die doorheen je opleiding moeten gevuld worden. Naarmate we meer C# leren zullen er extra guidelines bijkomen (zie [deze appendix voor alle guidelines van de opleiding](#)).

- **Duidelijke naam:** de identifier moet duidelijk maken waarvoor de identifier dient. Schrijf dus liever `gewicht` of `leeftijd` in plaats van `a` of `meuh`.
- **Camel casing:** gebruik camel casing indien je meerdere woorden in je identifier wenst te gebruiken. Camel casing wil zeggen dat ieder nieuw woord terug met een hoofdletter begint. Een goed voorbeeld kan dus `zijnLeeftijdTimDams` of `aantalLeerlingenKlas1EA`. Merk op dat we liefst het eerste woord met kleine letter starten.

Commentaar

Soms wil je misschien extra commentaar bij je code zetten. Als je dat gewoon zou doen (bv `Dit deel zal alles verwijderen`) dan zal je compiler niet begrijpen wat die zin doet. Hij verwacht namelijk C# syntax en niet een Nederlandstalige zin. Om dit op te lossen kan je in je code op twee manieren aangeven dat een stuk tekst gewoon commentaar is en mag genegeerd worden door de compiler:

Enkele lijn commentaar met //

Eén lijn commentaar geef je aan door de lijn te starten met twee voorwaartse slashes `//`. Uiteraard mag je ook meerdere lijnen op deze manier in commentaar zetten. Zo wordt dit ook vaak gebruikt om tijdelijk een stuk code "uit te schakelen". Ook mogen we commenaar achter een stuk C# code plaatsen (zie onderaan voorbeeld)

```
//De start van het programma
int getal=3;
//Nu gaan we rekenen
int result = getal * 5;
// result= 5* 4;
Console.WriteLine(result); //We tonen resultaat op scherm
```

Blok commentaar met / en /

We kunnen een stuk tekst als commentaar aangeven door voor de tekst `/*` te plaatsen en achteraan de blok tekst `*/`. Een voorbeeld:

```
/*
Veel commentaar.
Een heel verhaal
Mooi he.
Is dit een haiku?
*/
int leeftijd= 0;
leeftijd++;
```

Typen en variabelen

Een variabele is een soort container in een C# programma waarin we een data waarde kunnen opslaan in het computergeheugen. De bewaarde waarde kan aangeroepen worden via de naam van de variabele.

Een essentieel onderdeel van C# is kennis van datatypes. Binnen C# zijn een aantal types gedefinieerd die je kan gebruiken om data in op te slaan. Wanneer je data wenst te bewaren in je applicatie dan zal je je moeten afvragen wat voor soort data het is. Gaat het om een getal, een geheel getal, een kommagetal, een stuk tekst of misschien een binaire reeks? Ieder datatype in C# kan één welbepaald soort data bewaren, en dit zal telkens een bepaalde hoeveelheid geheugen vereisen.

De data zelf bewaren we in **variabelen** van een **bepaald type**. Een variabele is een plekje in het geheugen dat in je programma zal gereserveerd worden om daarin data te bewaren van het type dat je aan de variabele hebt toegekend. Een variabele zal intern een geheugenadres hebben (waar de data in het geheugen staat) maar dat zou lastig programmeren meer, daarom moeten we ook steeds een naam oftewel **identifier** aan de variabele geven.

De naam (identifier) van de variabele moet voldoen aan de identifier regels uit het vorige hoofdstuk.

Om een variabele te maken moeten we deze **declareren**, door een type en naam te geven. Een variabele declaratie heeft als syntax:

```
datatype variabelenaam;
```

Je mag ook meerdere variabele van hetzelfde datatype in 1 enkele declaratie aanmaken door deze met komma's te scheiden:

```
datatype variabelenaam1, variabelenaam2, variabelenaam3;
```

Dit alles bespreken we in de komende secties.

Datatypes

Basistypen voor getallen

De belangrijkste basistypen (zogenaamde primitieve types) van C# om getallen weer te geven zijn

- Voor gehele getallen: `sbyte, byte, short, ushort, int, uint, long`
- Voor kommagetallen: `double, float, decimal`

Bereik van datatypes

Deze datatypes hebben allemaal een bepaald bereik, wat een rechtstreeks gevolg is van de hoeveelheid geheugen die ze innemen.

Voor de gehele getallen (voordeel in vet):

| Type | Geheugen | Range |
|---------------------|----------|-----------------------------|
| <code>sbyte</code> | 8 bits | -128 tot 127 |
| <code>byte</code> | 8 bits | 0 tot 255 |
| <code>short</code> | 16 bits | -32768 tot 32767 |
| <code>ushort</code> | 16 bits | 0 tot 65535 |
| <code>int</code> | 32 bits | -2147483658 tot +2147483657 |
| <code>uint</code> | 32 bits | 0 tot 4294967295 |
| <code>long</code> | 64 bits | Zie boek |
| <code>char</code> | 16 bits | 0 tot 65535 |

Voor de kommagetallen:

| Type | Geheugen | Range | Precisie |
|----------------------|----------------|---|---------------------|
| <code>float</code> | 32 bits | $1,5 \cdot 10^{-45}$ tot $3,4 \cdot 10^{48}$ | 7 digits |
| <code>double</code> | 64 bits | $5 \cdot 10^{-324}$ tot $1,7 \cdot 10^{308}$ | 15 digits |
| <code>decimal</code> | 128 bits | $1 \cdot 10^{-28}$ tot $7,9 \cdot 10^{28}$ | 28-29 digits |

Bereik in code weten

Het bereik van deze typen is weliswaar opgegeven in het handboek. Maar het is belangrijk om weten dat deze ook in de compiler gekend is. Het volgende voorbeeld toont dit aan:

```
string zinnetje = "Het bereik van het type double is:";  
Console.WriteLine(zinnetje + double.MinValue + " en " + double.MaxValue);
```

Je kan met andere woorden met `int.MaxValue` en `int.MinValue` het minimum- en maximumbereik van het type int verkrijgen. Wil je dit van een double, dan gebruik je `double.MaxValue` etc.

Variabelen aanmaken en gebruiken

Telkens wanneer je een variabele wenst aan te maken, om te gebruiken als werkgeheugen in je programma, dan dien je eerst deze variabele te definiëren.

Hiervoor dien je minstens op te geven:

1. Het datatype (bv int, double)
2. Een identifier zodat de variabele uniek kan geïdentificeerd worden ([volgens de naamgevingsregel van C#](#))
3. (optioneel) Een beginwaarde die de variabele krijgt bij het aanmaken ervan

Voorbeeld:

```
int mijnLeeftijd;
```

We maken variabele van het type 'int' (geheel getal) dat als identifier 'eenVariabele' heeft.

Indien je reeds weet wat de beginwaarde moet zijn van de variabele dan mag je de variabele ook reeds deze waarde toekennen bij het aanmaken:

```
int mijnLeeftijd = 37;
```

Waarden toekennen aan variabelen

Vanaf dit punt kunnen we dus ten allen tijde deze variabele gebruiken om een waarde aan toe te kennen, de bestaande waarde te overschrijven, of de waarde te gebruiken:

- Waarde toekennen: `mijnGetal= 15;`
- Waarde tonen op scherm: `Console.WriteLine(mijnGetal);`

Met de **toekenningsoperator (=)** kan je een waarde toekennen aan een variabele. Hierbij kan je zowel een literal toekennen oftewel het resultaat van een expressie.

Je kan natuurlijk ook een waarde uit een variabele uitlezen en toewijzen aan een andere variabele:

```
int eenAndereLeeftijd= mijnLeeftijd;
```

Literal toewijzen

Literals zijn expliciet ingevoerde waarden in je code. Als je in je code expliciet de waarde 4 wilt toekennen aan een variabele dan is het getal 4 in je code een zogenaamde literal. Wanneer we echter data bijvoorbeeld eerst uitlezen of berekenen (via bijvoorbeeld invoer van de gebruiker of als resultaat van een berekening) en het resultaat hiervan toekennen aan een variabele dan is dit geen literal.

Voorbeelden van een literal toekennen:

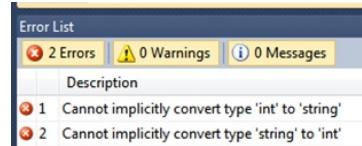
```
int temperatuurGisteren = 20;  
int temperatuurVandaag = 25;
```

Het is belangrijk dat het type van de literal overeenstemt met dat van de variabele waaraan je deze zal toewijzen. Een string-literal stel je voor door aanhalingstekens. Volgende code zal dan ook een compiler-fout genereren, daar je een string-literal aan een int-variabele wil toewijzen, en vice versa.

```
string eenTekst;
int eenGetal;

eenTekst = 4;
eenGetal = "4";
```

Als je bovenstaande probeert te compileren dan krijg je volgende error-boodschappen krijgen:



Nieuwe waarden overschrijven oude waarden

Wanneer je een reeds gedeclareerde variabele een **nieuwe waarde toekent** dan zak de oude waarde in die variabele onherroepelijk verloren zijn. Probeer dus altijd goed op te letten of je de oude waarde nog nodig hebt of niet. Wil je de oude waarde ook nog bewaren dan zal je een nieuwe, extra variabele moeten aanmaken en daarin de nieuwe waarde moeten bewaren:

```
int temperatuurGisteren = 20;
temperatuurGisteren = 25;
```

In dit voorbeeld zal er dus voor gezorgd worden dat de oude waarde van temperatuurGisteren, 20, zal worden overschreven met 25.

Volgende code toont hoe je bijvoorbeeld eerst de vorige waarde kunt bewaren:

```
int temperatuurEerGisteren= temperatuurGisteren;
temperatuurGisteren = 25;
```

First program verbeteren

We nemen terug ons eerste programma erbij en gaan hier aan verder werken:

```
namespace Demo1
{
    class Program
    {
        static void Main(string[] args)
        {
            Console.WriteLine("Hoi, ik ben het!");
            Console.WriteLine("Wie ben jij?!");
            Console.ReadKey();
        }
    }
}
```

ReadLine: Input van de gebruiker verwerken

Met de console kan je met een handvol methoden reeds een aantal interessante dingen doen.

Zo kan je bijvoorbeeld input van de gebruiker inlezen en bewaren in een variabele als volgt:

```
string result;
result = Console.ReadLine();
```

Bespreking van deze code:

```
string result;

• Concreet zeggen we hiermee aan de compiler: maak in het geheugen een plekje vrij waar enkel data van het type string in mag bewaard worden;
• Noem deze geheugenplek result zodat we deze later makkelijk kunnen in en uitlezen.

result = Console.ReadLine();

• Vervolgens roepen we de ReadLine methode aan. Deze methode zal de invoer van de gebruiker uitlezen tot de gebruiker op enter drukt.
• Het resultaat van de ingevoerde tekst wordt bewaarde in de variabele result.
```

Je programma zou nu moeten zijn:

```
namespace Demo1
{
    class Program
    {
        static void Main(string[] args)
        {
            Console.WriteLine("Hoi, ik ben het!");
            Console.WriteLine("Wie ben jij?!");
            string result;
            result = Console.ReadLine();
            Console.ReadLine();
        }
    }
}
```

Start nogmaals je programma. Je zal merken dat je programma nu een cursor toont en wacht op invoer. Als je eender welke tekst intypt en drukt op enter dan zal je programma stoppen met 'Druk op een toets om door te gaan...').

Meer input vragen

Als je meerdere inputs van de gebruiker tegelijkertijd wenst te bewaren dan zal je meerdere geheugenplekken nodig hebben om de invoer te bewaren. Bijvoorbeeld:

```
string leeftijd;
Console.WriteLine("Geef leeftijd");
result = Console.ReadLine();
```

```
Console.WriteLine("Geef adres");
string adres;
result = Console.ReadLine();
```

Input gebruiker verwerken en gebruiken

We kunnen nu invoer van de gebruiker, dat we hebben bewaard in de variabele `result` gebruiken en tonen op het scherm.

```
Console.WriteLine("Dag ");
Console.WriteLine(result);
Console.WriteLine(" hoe gaat het met je?");
```

In de tweede lijn hier gebruiken we de variabele `result`, waar de invoer van de gebruiker in bewaard wordt als parameter in de `WriteLine`-methode. Met andere woorden: de `WriteLine` methode zal op het scherm tonen wat de gebruiker even daarvoor heeft ingevoerd.

Je volledige programma ziet er dus nu zo uit:

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

namespace Demo1
{
    class Program
    {
        static void Main(string[] args)
        {

            Console.WriteLine("Hoi, ik ben het!");
            Console.WriteLine("Wie ben jij?!");

            string result;
            result = Console.ReadLine();

            Console.WriteLine("Dag");
            Console.WriteLine(result);
            Console.WriteLine("hoe gaat het met je?");
            Console.ReadLine();
        }
    }
}
```

Test het programma en voer je naam in wanneer de cursor knippert.

Aanhalingsteken of niet?

Wanneer je de inhoud van een variabele wil gebruiken in een methode zoals `WriteLine()` dan plaats je deze zonder aanhalingsteken! Bekijk zelf eens wat het verschil wordt wanneer je volgende lijn code `Console.Write(result);` vervangt door `Console.WriteLine("result");`.

Write en WriteLine

De `WriteLine`-methode zal steeds een line break ([enter] zeg maar) aan het einde van de lijn zetten zodat de cursor naar de volgende lijn springt.

De `Write`-methode zal geen enter aan het einde van de lijn toevoegen. Als je dus vervolgens iets toevoegt (met een volgende `Write` of `WriteLine`) dan zal dit aan dezelfde lijn toegevoegd worden.

Vervang daarom eens de laatste 3 lijnen code in je project door:

```
Console.Write("Dag");
Console.Write(result);
Console.WriteLine("hoe gaat het met je?");
```

Voer je programma uit en test het resultaat.

Wat is er verkeerd gelopen? Al je tekst van de laatste lijn plakt zo dicht bij elkaar? Inderdaad, we zijn spaties vergeten toe te voegen! Spaties zijn ook tekens die op scherm moeten komen (ook al zien we ze niet) en je dient dus binnen de aanhalingstekens spaties toe te voegen.

Opletten met spaties

Spaties zijn ook tekens die op scherm moeten komen (ook al zien we ze niet) en je dient dus binnen de aanhalingstekens spaties toe te voegen. Indien je deze erbuiten plaats dan heeft dit geen effect. *In volgend voorbeeld zijn de spaties aangegeven als liggende streepjes (_).*

Fout:

```
Console.WriteLine("Dag_");
Console.WriteLine(result_);
Console.WriteLine("hoe gaat het met je?");
```

Correct:

```
Console.WriteLine("Dag_");
.Console.WriteLine(result_);
Console.WriteLine("_hoe gaat het met je?");
```

Zinnen aan elkaar plakken

We kunnen dit hele verhaal een pak koper tonen. De plus-operator (+) in C# kan je namelijk gebruiken om variabelen van het type string aan elkaar te plakken. De laatste 3 lijnen code kunnen koper schrijven als volgt

```
Console.WriteLine("Dag" + result + " hoe gaat het met je?");
```

Merk op dat result dus NIET tussen aanhalingstekens staat, in tegenstelling de andere stukken zin. Waarom is dit? Aanhalingsstekens in C# duiden aan dat een stuk tekst moet beschouwd worden als tekst van het type string. Als je geen aanhalingssteken gebruikt dan zal C# de tekst beschouwen als een variabele met die naam.

Bekijk zelf eens wat het verschil wanneer je volgende lijn code vervangt door de lijn er onder:

```
Console.WriteLine("Dag " + result + " hoe gaat het met je?");
Console.WriteLine("Dag " + "result" + " hoe gaat het met je?");
```

Char

Een enkele letter of getal als 'tekst' opslaan kan je doen door het char-type te gebruiken. Zo kan je bijvoorbeeld een enkel karakter als volgt tonen:

```
char eenLetter = 'X';
Console.WriteLine("eenLetter=" + eenLetter);
```

Het is belangrijk dat je de apostrof ('') niet vergeet voor en na het karakter dat je wenst op te slaan!

Je kan eender welk ASCII-teken in een char-bewaren, namelijk letters, cijfers en speciale tekens (% , \$, * , #, etc.) Merk dus op dat volgende lijn: char eenGetal = '7'; Weliswaar een getal als teken opslaan, maar dat intern de compiler deze variabele steeds als een character zal gebruiken. Als je dit cijfer zou willen gebruiken als effectief cijfer om wiskundige bewerkingen op uit te voeren, dan zal je dit eerst moeten converteren naar een getal ([zie Convert en Casting](#)).

String

Een string is een reeks van 0, 1 of meerdere chars, zoals je ook kan zien als je even met je muis boven een string keyword *hovert* in je code:

```
string eenZin = "Wat een mooie zin";
class System.String
Represents text as a series of Unicode characters.
```

Alles van het type string uitleggen zou ons voorlopig te ver nemen; het is belangrijk om te beseffen nu wat het verschil is tussen een string en een karakter.

Strings declareren

Merk op dat we bij een string literal gebruik maken van aanhalingstekens ("") terwijl bij chars we een apostrof ('') gebruiken (''). Dit is de manier om een string van een char te herkennen.

Volgende code geeft dus drie keer het cijfer 1 onder mekaar op het scherm, maar de eerste keer behelst het een char (enkelvoudig teken), dan een een string (reeks van tekens) en dan een int (effectief getal):

```
char eenKarakter = '1';
string eenString = "1";
int eenGetal = 1;

Console.WriteLine(eenKarakter);
Console.WriteLine(eenString);
Console.WriteLine(eenGetal);
```

Fout gebruik van strings en chars: Volgende code zal dus niet compileren!

```
char eenKarakter = "1";
string eenString = '1';
int eenGetal = '1';

Console.WriteLine(eenKarakter);
Console.WriteLine(eenString);
Console.WriteLine(eenGetal);
```

In de eerste toekenning proberen we dus een literal van het type string toe te kennen een variabele van het type char. In de tweede toekenning proberen we een literal van het type char toe te kennen een variabele van het type string. En in de laatste toekenning proberen we een literal van het type char toe te kennen een variabele van het type int.

Escape characters

Naast letters en tekens mogen in string en chars ook escape characters staan. Escape characters worden met een backslash () gestart, gevuld door het karakter dat we wensen te tonen. In C# hebben bepaalde tekens namelijk een speciale functie, zoals de dubbele aanhalingstekens ("") om het begin of einde van een string-literal aan te geven .

Zonder aan te geven dat we letterlijk dat teken willen tonen, en het niet in z'n C# functie gebruiken, zouden we problemen krijgen.

Denk bijvoorbeeld aan de apostrof... Volgende code zou de compiler verkeerd interpreteren, daar hij denkt dat we een leeg karakter wensen op te slaan:

```
char apostrof= '';
```

De juiste manier is om het teken dus door een backslash vooraf te laten gaan:

```
char apostrof= '\'';
```

Er zijn echter nog een heleboel andere escape characters die je gereeld zal moeten gebruiken, waaronder \n om een nieuwe lijn aan te geven en \t om een tab in de tekst te plaatsen.

Bekijk deze tabel voor alle toegelaten escape characters in C#

Escape characters in strings

We gebruiken vooral escape characters in strings, om bijvoorbeeld witregels en tabulaties aan te geven. Test bijvoorbeeld volgende lijn eens:

```
string eenString = "Eerst een zin.\t dan een zin na een tab \n Dan eentje op een nieuwe regel";
```

Het is belangrijk dat je vlot kan werken met escape characters in string, daar we dit gereeld zullen nodig hebben.

Optellen van char

Stel dat we volgende char-variabele aanmaken. Bij string mogen we de + operator gebruiken om 2 strings aan elkaar te plakken. Bij char mag dat niet!

```
char a = 'A';
char b = 'B';
Console.WriteLine(a + b);
```

Wanneer je deze code uitvoert dan krijg je een getal te zien?!

Had je dit verwacht? Herinner je dat het char-type z'n waarde als getallen bijhoudt, de zogenaamde UNICODE voorstelling van het karakter. Als de compiler het volgende ziet staan:

```
eenTab + eenLetter
```

dan zal de compiler deze twee waarden letterlijk optellen en het nieuw verkregen getal als resultaat geven.

Strings samenvoegen

Je kan strings en variabelen samenvoegen tot een nieuwe string op verschillende manieren:

- +-operator
- \$ string interpolation
- String.Format()

Stel dat je 2 variabelen hebt `int age=13` en `string name="Finkelstein"`. We willen de inhoud van deze variabelen samenvoegen in een nieuwe `string result` die zal bestaan uit de tekst: `Ik ben Finkelstein en ik ben 13 jaar oud`

Volgende 3 manieren tonen hoe je steeds tot voorgaande string zal komen.

Manier 1 String samenvoegen met de +-operator

Je kan string en variabelen eenvoudig 'bij mekaar' optellen. Ze worden dan achter elkaar geplakt als het waren.

```
string result= "Ik ben "+ name + "en ik ben "+ age+ " jaar oud";
```

Let er op dat je tussen de aanhalingstekens (binnen de strings) spaties zet indien je het volgende deel niet 'tegen het vorige stringstuk' wilt plakken.

* string interpolation

In de oude dagen van C# gebruiken we `String.Format()` om meerdere string(s) en variabelen samen te voegen tot een string. Nu kan dat met string interpolation waarbij we het \$-teken gebruiken. Door het \$-teken VOOR de string te plaatsen geef je aan dat alle delen in de string die tussen accolades staan {} als code mogen beschouwd worden. Een voorbeeld maakt dit duidelijk:

```
string result= $"Ik ben {name} en ik ben {age} jaar oud";
```

Zoals je kan zien is dit veel leesbaarder dan de eerste manier.

Berekeningen doen

Je mag eender welk statement tussen de accolades zetten, denk maar aan:

```
string result= $"Ik ben {name} en ik ben {age+4} jaar oud";
```

String.Format()

`String.Format` is een methode die string-interpolatie toe laat op een iets minder intuïtieve manier:

```
string result= String.Format("Ik ben {0} en ik ben {1} jaar oud",name,age);
```

Het getal tussen de accolades geeft aan de hoeveelste parameter na de string hier in de plaats moet gezet worden (0= de eerste, 1= de tweede, enz).

Volgende code zal dus `Ik ben 13 en ik ben 13 jaar oud` als resultaat geven:

```
string result= String.Format("Ik ben {1} en ik ben {1} jaar oud",name,age);
```

Wens je meer informatie over `String.Format`, kijk dan [hier](#)

Rommel zin

Schrijf een applicatie met behulp van ReadLine en WriteLine()-methoden waarbij de computer aan de gebruiker om zijn favoriete kleur, eten, auto, film en boek. Het programma zal de antwoorden echter door elkaar halen waardoor de computer vervolgens toont Je favoriete kleur is [eten]. Je eet graag [auto]. Je lievelingsfilm is [boek] en je favoriete boek is [kleur].

Waarbij tussen de rechte haakjes steeds de invoer komt die de gebruiker eerder opgaf voor de bijhorende vraag.

BMI berekenaar

Maak een programma dat aan de gebruiker z'n lengte en gewicht vraagt en vervolgens z'n berekende BMI (Body Mass Index) toont. Bereken na met je rekenmachine of je uitkomst wel degelijk klopt!

Expressies

Zonder expressies is programmeren saai: je kan dan enkel variabelen aan elkaar toewijzen. Expressies zijn als het ware eenvoudige tot complexe sequenties van bewerkingen die resulteren tot 1 resultaat. De volgende code is bijvoorbeeld een expressie: `3+2`

Het resultaat van deze expressie is 5. [Meer informatie over expressies hier.](#)

Expressie-resultaat toewijzen

Meestal zul je expressies schrijven waarin je bewerkingen op en met variabelen uitvoert. Vervolgens zal je het resultaat van die expressie willen bewaren voor verder gebruik in je code.

Voorbeeld van **expressie**-resultaat toekennen:

```
int temperatuursVerschil = temperatuurGisteren - temperatuurVandaag;
```

Hierbij zal de temperatuur uit de rechtse 2 variabelen worden uitgelezen, van elkaar wordt afgetrokken en vervolgens bewaard worden in `temperatuursVerschil`.

De voorgaande code kan ook langer geschreven worden als:

```
int tussenResultaat= temperatuurGisteren - temperatuurVandaag;
int temperatuursVerschil = tussenResultaat;
```

Een ander voorbeeld van een expressie-resultaat toewijzen maar nu met literal (stel dat we `temperatuursVerschil` reeds hebben gedeclareerd eerder):

```
temperatuursVerschil = 21 - 25;
```

Uiteraard mag je ook combinaties van literals en variabelen gebruiken in je expressies:

```
int breedte = 15;
int hoogte = 20 * breedte;
```


Casting, conversie en parsing: data omzetten

Wanneer je de waarde van een variabele wil toekennen aan een variabele van een ander type mag dit niet zo maar. Je kan geen appelen in peren veranderen zonder magie: in het geval van C# zal je moeten converteren of casten. Parsing is iets anders: deze zal je enkel nodig hebben om data door de computer te 'interpretieren', vooral nuttig als je bijvoorbeeld tekst naar getallen wilt omzetten.

Casting

Hierbij dien je aan de compiler te zeggen: "Volgende variabele die van het type x is, moet aan deze variabele van het type y toegekend worden. Ik besef dat hierbij data verloren kan gaan, maar zet de variabele toch maar om naar het nieuwe type".

Stel dat temperatuurGisteren en temperatuurVandaag van het type int zijn, maar dat we nu de gemiddelde temperatuur willen weten. De formule voor gemiddelde temperatuur over 2 dagen is:

```
int temperatuurGemiddeld = (temperatuurGisteren + temperatuurVandaag)/2;
```

Test dit eens met de waarden 20 en 25. Wat zou je verwachten als resultaat? Inderdaad: $22,5$ ($20+25)/2 = 22,5$) *Nochtans krijg je 22 op scherm te zien en zal de variabele temperatuurGemiddeld ook effectief de waarde 22 bewaren en niet 22,5.*

Het probleem is dat het gemiddelde van 2 getallen niet noodzakelijk een geheel getal is. **Omdat de expressie enkel integers bevat (temperatuurGisteren en temperatuurVandaag) zal ook het resultaat een integer zijn.** In dit geval wordt alles na de komma gewoon weggegooid, vandaar de uitkomst. **Dit is narrowing.**

Hoe krijgen we de correctere uitslag te zien? Door temperatuurGemiddeld als kommagetal te declareren (bijvoorbeeld door het type double):

```
double temperatuurGemiddeld = (temperatuurGisteren + temperatuurVandaag)/2;
```

Als we dit testen zal nog steeds de waarde 22 aan temperatuurGemiddeld toegewezen worden. Inderdaad. De expressie rechts bevat enkel integers en de computer zal dus ook de berekening en het resultaat als integer beschouwen. We moeten dus ook de rechterkant van de toekenning als double beschouwen. *We doen dit door middel van casting zoals eerder vermeld*, als volgt:

```
double temperatuurGemiddeld = ((double)temperatuurGisteren + (double)temperatuurVandaag)/2;
```

Nu zal temperatuurGemiddeld wel de waarde 22,5 bevatten.

Wat is casting

Casting heb je nodig om een variabele van een bepaald type voor een ander type te laten doorgaan? Stel dat je een complexe berekening hebt waar je werkt met verschillende types (bijvoorbeeld int, double en float). Door te casten voorkom je dat je vreemde resultaten krijgt. Je gaat namelijk bepaalde types even als andere types gebruiken.

Het is belangrijk in te zien dat het casten van een variabele naar een ander type enkel een gevolg heeft TIJDENS het uitwerken van de expressie waarbinnen je werkt. De variabele in het geheugen zal voor eeuwig en altijd van het type zijn waarin het origineel gedeclareerd werd.

Casting voorbeelden

Narrowing

Casting doe je wanneer je een variabele wil toekennen aan een andere variabele van een ander type dat daar eigenlijk niet inpast. Bekijk eens volgende voorbeeld:

```
double var1;
int var2;

var1 = 20.4;
var2 = var1;
```

Dit zal niet gaan. Je probeert namelijk een waarde van het type double in een variabele van het type int te steken. Dat gaat enkel als je informatie weggooit. Je moet aan *narrowing* doen. Dit gaat enkel als je expliciet aan de compiler zegt: het is goed, je mag informatie weggooien, ik begrijp dat en zal er rekening mee houden. Dit proces van narrowing noemen we casting.

En je lost dit op door voor de variabele die TIJDELIJK dienst moet doen als een ander type, het nieuwe type, tussen ronde haakjes, te typen, als volgt:

```
double var1;
int var2;

var1 = 20.4;
var2 = (int)var1;
```

Het resultaat in var2 zal 20 zijn (alles na de komma wordt bij casting van een double naar een int weggegooid).

Widening

Casting is echter dus niet nodig als je aan *widening* doet (een kleiner type in een groter type steken), als volgt:

```
int var1;
double var2;

var1 = 20;
var2 = var1;
```

Deze code zal zonder problemen gaan. Var2 zal de waarde 20.0 bevatten. De inhoud van var1 wordt *verbreed* naar een double, eenvoudigweg door er een kommagetal van te maken. Er gaat **geen** inhoud verloren echter.

Conversie

Casting is de 'oldschool' manier van data omzetten die vooral zeer nuttig is daar deze ook werkt in andere C#-related programmeertalen zoals C, C++ en Java.

Echter, .NET heeft ook enkele ingebouwde conversie-methoden die je kunnen helpen om data van het ene type naar het andere te brengen. Al deze methoden zitten binnen de **Convert**-bibliotheek.

Het gebruik hiervan is zeer eenvoudig. Enkele voorbeelden:

```
int getal= Convert.ToInt32(3.2); //double to int
double anderGetal= Convert.ToDouble(5); //int to double
bool isWaar= Convert.ToBoolean(1); //int to bool
int userAge= Convert.ToInt32("19"); //string to int
```

De convert-klasse bevat tal van conversiemogelijkheden. Opgelet: de convert zal zelf zo goed mogelijk de data omzetten en dus indien nodig widening of narrowing toepassen. Zeker bij het omzetten van een string naar een ander type kijk je best steeds de documentatie na om te weten wat er intern juist zal gebeuren.

Je kan [alle conversie-mogelijkheden hier bekijken](#)

Parsing

Ieder ingebouwd type heeft ook een `.Parse()` methode die je kan aanroepen om strings om te zetten naar het gewenste type. Parsing zal je echter minder vaak nodig hebben. Gebruik deze enkel wanneer

1. Je een string hebt waarvan je weet dat deze altijd van een specifiek type zal zijn, bv een int, dan kan je `Int32.Parse()` gebruiken
2. Je input van de gebruiker vraagt (bv via `Console.ReadLine()`) en niet 100% zeker bent dat deze een getal zal bevatten, gebruik dan `Int32.TryParse()` [info](#)

Er zijn nog subtiele verschillen die we hier niet behandelen ([zie](#))

Random getallen genereren in je code kan leuk zijn om de gebruiker een interactievere ervaring te geven. Beeld je in dat je monsters steeds dezelfde weg zouden bewandelen of dat er steeds op hetzelfde tijdstip een orkaan op je stad neerdwaalt. SAAI!

Random generator

De `Random`-klasse laat je toe om eenvoudig willekeurige gehele en komma-getallen te maken. Je moet hiervoor twee zaken doen:

1. Maak **eenmalig** een Random-generator aan: `Random randomgen= new Random();`
2. Roep de `Next` methode aan telkens je een nieuw getal nodig hebt, bijvoorbeeld: `int mijnGetal= randomgen.Next()`
`Next()` zal een geheel getal genereren.

De eerste stap dien je dus maar 1 keer te doen. Vanaf dan kan je telkens aan de generator een nieuw getal vragen m.b.v. `Next`.

Volgende code toont bijvoorbeeld 3 random getallen op het scherm:

```
Random mygen= new Random();
int getal1= mygen.Next();
int getal2= mygen.Next();
int getal3= mygen.Next();
Console.WriteLine(getal1);
Console.WriteLine(getal2);
Console.WriteLine(getal3);
```

Next mogelijkheden

Je kan de `Next` methode ook 2 paramters meegeven, namelijk de grenzen waarbinnen het getal moet gegenereert worden. De 2e parameter is exclusief dit getal zelf. Wil je dus een getal tot en met 10 dan schrijf je 11, niet 10.

Enkele voorbeelden:

```
Random somegenerator = new Random();

int a= somegenerator.Next(0,10); //getal tussen 0 tot en met 9
int b= somegenerator.Next(5,10); //getal tussen 5 tot en met 10
int c= somegenerator.Next(0,b); //getal tussen 0 tot en met het getal dat de lijn ervoor werd gegenereert.
```

Genereer kommagetallen met `NextDouble`

Met de `NextDouble` methode kan je kommagetallen genereren tussen `0.0` en `1.0` (1.0 zal niet gegenereert worden).

Wil je een groter kommagetal dan zal je dit gegenereerde getal moeten vermenigvuldigen naar de range die je nodig hebt. Stel dat je een getal tussen 0.0 en 10.0 nodig hebt dan schrijf je:

```
Random myran= new Random();
double randomgetal= myran.NextDouble() * 10.0;
```

Euro naar dollar

Ontwerp een toepassing waarmee je een ingevoerd bedrag, inclusief komma-getallen in euro kan omrekenen naar dollar. Gebruik hierbij de huidige wisselkoers. Je hoeft niet af te ronden. Het resultaat in een label wordt als volgt weergegeven: `X EUR is gelijk aan Y USD`.

Feestkassa

De plaatselijke voetbalclub organiseert een mossselfestijn. Naast mosselen met frietjes (20 EUR) bieden ze voor de kinderen de mogelijkheid om een koninginnehapje (10 EUR) te kiezen. Verder is er een ijsje als nagerecht voorzien (3 EUR). Om het gemakkelijk te maken kosten alle dranken 2 EUR.

Ontwerp een applicatie zodat de vrijwilliger aan de kassa alleen maar de juiste aantallen moet ingeven ,lijn per lijn. (frietjes, koninginnehapje, ijsje, drank) om de totaal prijs te berekenen.

Het resultaat wordt als volgt weergegeven: `Het totaal te betalen bedrag is x EU`.

Systeem informatie

Deel 1

Maak een applicatie die de belangrijkste computer-informatie (geheugen, etc) aan de gebruiker toont.

Deze computer-informatie kan je verkrijgen mbv de Environment-klasse. Hier enkele voorbeelden (kijk zelf of er nog nuttige properties over je computer in staan):

```
bool is64bit = Environment.Is64BitOperatingSystem;
string pcname = Environment.MachineName;
int procCount = Environment.ProcessorCount;
string username = Environment.UserName;
long memory = Environment.WorkingSet; //bytes
```

Zoals je ziet wordt het geheugen in bytes teruggegeven. Zorg ervoor dat het geheugen steeds in mega of gigabytes op het scherm wordt getoond.

Formateer de informatie met behulp van de \$-notatie zodat deze deftig getoond wordt.

Deel 2

Ook informatie over de harde schijven kan je verkrijgen (in bits)

```
long cdriveinbytes = DriveInfo.GetDrives()[0].AvailableFreeSpace;
long totalsize = DriveInfo.GetDrives()[0].TotalSize;
```

De 0 tussen rechte haakjes is de index van welke schijf je informatie wenst. 0 is de eerste harde schijf, 1 de tweede, enzovoort. (Ter info: dit zijn arrays, zie later)

Vraag aan de gebruiker het nummer van de harde schijf waar meer informatie over moet getoond worden.

Opgelet: sta toe dat de gebruiker 1 voor de eerste harde schijf mag gebruiken, 2 voor de tweede, enzovoort. Je zal dus in code nog manueel moeten aftrekken van de invoer van de gebruiken. Bv:

```
int invoer = Convert.ToInt(Console.ReadLine()) - 1;
long totalsize = DriveInfo.GetDrives()[invoer].TotalSize;
```

Het Orakel van Delphi

Gebruik de random generator om een orakel/waarzegger te maken. Het programma zal aan de gebruiker vertellen hoe lang deze nog zal leven. Bijvoorbeeld: "Je zal nog 15 jaar leven."

Beslissingen/If statements

Nu we de elementaire zaken van C# en Visual Studio kennen is het tijd om onze programma's wat interessanter te maken. De ontwikkelde programma's tot nog toe waren stevast lineair van opbouw, ze werden lijn per lijn uitgevoerd zonder de mogelijkheid om de *flow* van het programma aan te passen.

In dit deel zullen we bekijken hoe we ons programma dynamischer kunnen maken met behulp van het if-statement.

If

De `if` uitdrukking is 1 van de elementairste uitdrukking in een programmeertaal. De syntax is als volgt:

```
if (boolean expression)
{
    // C# code to be performed when expression evaluates to true
}
```

Enkel indien de boolean expressie true als resultaat geeft zal de code binnen de accolades van het if-blok uitgevoerd worden. Indien de expressie false terugvalt dan wordt het blok overgeslagen en gaat het programma verder met de code eronder.

Een voorbeeld:

```
int x = 10;

if ( x > 9 )
{
    System.Console.WriteLine ("x is greater than 9!");
}
```

If/else

Met if/else kunnen we niet enkel zeggen welke code moet uitgevoerd worden als de conditie waar is maar ook welke specifieke code indien de conditie niet waar (false). Volgende voorbeeld geeft een typisch gebruik van een if/else structuur om 2 waarden met elkaar te vergelijken:

```
int x = 10;

if ( x > 9 )
{
    System.Console.WriteLine ("x is greater than 9!");
}
else
{
    System.Console.WriteLine ("x is less than 9!");
}
```

If/else if

Met een if/else if constructie kunnen we meerdere criteria opgeven die waar/niet waar moeten zijn voor een bepaald stukje code kan uitgevoerd worden. Sowieso begint men steeds met een if. Als men vervolgens een else if plaatst dan zal de code van deze else if uitgevoerd worden enkel en alleen als de eerste expressie (van de if) niet waar was en de expressie van deze else if wel waar is.

Een voorbeeld:

```
int x = 9;

if (x == 10)
{
    System.Console.WriteLine ("x is 10");
}
else if (x == 9)
{
    System.Console.WriteLine ("x is 9");
}
```

```

else if (x == 8)
{
    System.Console.WriteLine ("x is 8");
}

```

Relationele operators

Met de [relationele operators](#) (`++, !=, <, >, <= en >=`) kunnen we expressie schrijven die als uitkomst waar (true) of niet waar (false) geven.

Een voorbeeld:

```

int a, b, c;

a = 2;
b = 3;

if(a < b) Console.WriteLine("a is less than b");

// this won't display anything
if(a == b) Console.WriteLine("you won't see this");

Console.WriteLine();

c = a - b; // c contains -1

Console.WriteLine("c contains -1");
if(c >= 0) Console.WriteLine("c is non-negative");
if(c < 0) Console.WriteLine("c is negative");

Console.WriteLine();

c = b - a; // c now contains 1
Console.WriteLine("c contains 1");
if(c >= 0) Console.WriteLine("c is non-negative");
if(c < 0) Console.WriteLine("c is negative");

```

We kunnen ook meerdere expressie combineren zodat we complexere uitdrukkingen kunnen maken. Hierbij kan je gebruik maken van de [logische operators AND \(&&\) en OR \(||\)](#)* operators. .

Een voorbeeld:

```

int a = 5, b = 5, c = 10;

if (a == b)
    Console.WriteLine(a);

if ((a > c) || (a == b))
    Console.WriteLine(b);

if ((a >= c) && (b <= c))
    Console.WriteLine(c);

```

Nesting

We kunnen met behulp van nesting ook complexere programma flows maken. Hierbij gebruiken we de accolades om het blok code aan te duiden dat bij een if/else/if else hoort. Binnen dit blok kunnen nu echter opnieuw if/else/if else structuren worden aangemaakt.

Volgende voorbeeld toont dit aan (bekijk wat er gebeurt als je emergencyValve aan `closed` gelijkstelt):

```

int reactorTemp = 1500;
string emergencyValve = " ";

if (reactorTemp < 1000)
{
    System.Console.WriteLine("Reactor temperature normal");
}
else
{
    System.Console.WriteLine("Reactor temperature too high!");
    if (emergencyValve == "closed")
    {
        System.Console.WriteLine("Reactor meltdown in progress!");
    }
}

```

}

Relationele operators

| C#-syntax | Betekenis |
|-----------|---------------------------|
| > | groter dan |
| < | kleiner dan |
| == | gelijk aan |
| != | niet gelijk aan |
| <= | kleiner dan of gelijk aan |
| >= | groter dan of gelijk aan |

Logische operator

De logische EN , OF en NIET-operators die je kent van de booleaanse algebra kan je ook gebruiken in C#:

| C#-syntax | Betekenis |
|-----------|---------------|
| && | en-operator |
| | of-operator |
| ! | niet-operator |

Je kan de niet-operator voor een expressie zetten om het resultaat hiervan om te draaien. Bijvoorbeeld:

```
!(0==2) //zal true geven
```

Switch

Een switch statement is een program-flow element om een veelvoorkomende constructie van if/if else.else elementen eenvoudiger te tonen. Vaak komt het voor dat we bijvoorbeeld aan de gebruiker vragen om een keuze te maken (bijvoorbeeld een getal van 1 tot 10, waarbij ieder getal een ander menu-item uitvoert van het programma), zoals:

```
int option;
Console.WriteLine("Kies 1 voor afbreken, 2 voor opslaan, 3 voor laden:");
option = int.Parse(Console.ReadLine());

if (option == 1)
    Console.WriteLine("Afbreken gekozen");
if (option == 2)
    Console.WriteLine("Opslaan gekozen");
if (option == 3)
    Console.WriteLine("Laden gekozen");
```

Met een switch kan dit eenvoudiger. De syntax van een switch is een beetje speciaal dan de andere programma flow-elementen (if, while, etc), namelijk als volgt:

```
switch (value)
{
    case constant:
        statements
        break;
    case constant:
        statements
        break;
    default:
        statements
        break;
}
```

Value is de waarde of variabele (beide mogen) die wordt gebruikt als test in de switch. Iedere case begint met het case keyword gevolgd door de waarde die value moet hebben om in deze case te *springen*. Na het dubbelpunt volgt vervolgens de code die moet uitgevoerd worden in deze case. De case zelf mag eender welke code bevatten (methoden, nieuwe program flow elementen, etc), maar moet zeker afgesloten worden met het break keyword.

Tijdens de uitvoer zal het programma value vergelijken met iedere case constant van boven naar onder. Wanneer een gelijkheid wordt gevonden dan wordt die case uitgevoerd. Indien geen case wordt gevonden die gelijk is aan value dan zal de code binnen de default-case uitgevoerd worden.

Opgelet:

De case waarden moeten constanten zijn en mogen dus geen variabelen zijn. Constanten zijn de welbekende *literals* (1, "1", 1.0, 1.d, '1' , etc.)

Het voorbeeldprogramma bovenaan wordt dan:

```
int option;
Console.WriteLine("Kies 1 voor afbreken, 2 voor opslaan, 3 voor laden:");
option = int.Parse(Console.ReadLine());

switch(option)
{
    case 1:
        Console.WriteLine("Afbreken gekozen");
        break;
    case 2:
        Console.WriteLine("Opslaan gekozen");
        break;
    case 3:
        Console.WriteLine("Laden gekozen");
        break;
    default:
        Console.WriteLine("Ongeldige optie");
        break;
}
```


Enum datatypes

Deel van dit hoofdstuk uit Visual C# 2012 - De basis (Sander Gerz)

De bestaansreden voor enums

Stel dat je een programma moet schrijven dat afhankelijk van de dag van de week iets anders moet doen. In een wereld zonder enums (enumeraties) zou je dan kunnen schrijven op 2 zeer foutgevoelige manieren:

1. Met een int die een getal van 1 tot en met 7 kan bevatten
2. Met een string die de naam van de dag bevat

Slechte oplossing 1: Met ints

De waarde van de dag staat in een variabele `int dagKeuze`. We bewaren er 1 voor Maandag, 2 voor dinsdag, enzovoort.

```
if(dagKeuze==1)
{
    Console.WriteLine("We doen de maandag dingen");
}
else
if (dagKeuze==2)
{
    Console.WriteLine("We doen de dinsdag dingen");
}
else
if //enz..
```

Deze oplossing heeft 2 grote nadelen:

- Wat als we per ongeluk `dagKeuze` een niet geldige waarde geven, zoals 9, 2000, -4, etc. ?
- De code is niet erg leesbaar. `dagKeuze==2`? Was dat nu dinsdag of woensdag (want misschien was maandag 0 i.p.v. 1)

Slechte oplossing 2: Met strings

De waarde van de dag bewaren we nu in een variabele `string dagKeuze`. We bewaren de dagen als "maandag", "dinsdag", etc.

```
if(dagKeuze=="maandag")
{
    Console.WriteLine("We doen de maandag dingen");
}
else
if (dagKeuze=="dinsdag")
{
    Console.WriteLine("We doen de dinsdag dingen");
}
else
if //enz..
```

De code wordt nu wel leesbaarder dan met 1, maar toch is ook hier 1 groot nadeel:

- De code is veel foutgevoeliger voor typfouten. Wanneer je "Maandag" i.p.v. "maandag" bewaard dan zal de if al niet werken. Iedere schrijffout of variant zal falen.

Enumeraties: het beste van beide werelden.

Enumeraties (`enum`) zijn een C# syntax die bovenstaand probleem oplost en het beste van beide samenvoegt. Het keyword `enum` geeft aan dat we een nieuw type maken dat maar enkele mogelijke waarden kan hebben. De variabele die we dan later maken zal van dit type zijn en enkel de opgegeven waarden mogen bevatten. Ook zal IntelliSense je de mogelijke waarden helpen invullen.

In C# zitten al veel Enum-types ingebouwd. Denk maar aan `ConsoleColor`: wanneer je de kleur van het lettertype van de console wilt veranderen gebruiken we een enum-type. Er werd reeds gedefinieerd wat de toegelaten waarden zijn: `Console.ForegroundColor= ConsoleColor.Red;`

Zelf een `enum` type maken gebeurt in 2 stappen:

1. Het type en de mogelijke waarden definiëren
2. Variabele(n) van het nieuwe type aanmaken gebruiken in je code

Stap 1: het type definiëren

We maken eerst een enum type aan. In je console-applicaties moet dit binnen `class Program` gebeuren, maar buiten de methode daar binnenvinden:

```
enum Weekdagen {Maandag, Dinsdag, Woensdag, Donderdag, Vrijdag, Zaterdag, Zondag}
```

Vanaf nu kan je variabelen van het type `Weekdagen` aanmaken.

Stap 2: variabelen van het type aanmaken en gebruiken.

We maken een variabele(n) aan:

```
Weekdagen dagKeuze;  
Weekdagen andereKeuze;
```

En vervolgens kunnen we waarden aan deze variabele(n) toewijzen als volgt

```
dagKeuze = Weekdagen.Donderdag;
```

Ook de beslissingsstructuren worden leesbaarden:

```
if(dagKeuze== Weekdagen.Woensdag)
```

of een switch:

```
switch(dagKeuze)  
{  
    case Weekdagen.Maandag:  
        Console.WriteLine("It's monday!");  
        break;  
    case Weekdagen.Dinsdag:  
        //etc.  
}
```

Conversie

Intern worden de enum-variabelen als ints bewaard. In het geval van de `Weekdagen` zal maandag de waarde 0 krijgen, dinsdag 1, etc.

Volgende conversies zijn dan ook perfect toegelaten:

```
int keuze= 3;  
  
Weekdagen dagKeuze = (Weekdagen)keuze;  
  
//dagKeuze zal de waarde Weekdagen.Donderdag hebben
```

Wil je dus bijvoorbeeld 1 dag bijtellen dan kan je schrijven:

```
Weekdagen dagKeuze= Weekdagen.Dinsdag;  
int extradag= (int)dagKeuze + 1;  
Weekdagen nieuweDag= (Weekdagen)extradag;  
//extraDag heeft de waarde Weekdagen.Woensdag
```

Andere interne waarde

Standaard worden de waarden dus genummerd intern beginnende bij 0, enz. Je kan dit ook manueel veranderen door bij het maken van de `enum` expliciet aan te geven wat de interne waarde moet zijn, als volgt:

```
enum WeekDagen {Maandag =1, Dinsdag,Woensdag, Donderdag, Vrijdag, Zaterdag, Zondag}
```

De dagen zullen nu vanaf 1 genummerd worden, dus `WeekDagen.Woensdag` zal de waarde 3 hebben.

We kunnen ook nog meer informatie meegeven, bijvoorbeeld:

```
enum WeekDagen {Maandag =1, Dinsdag,Woensdag, Donderdag, Vrijdag, Zaterdag=50, Zondag=60}
```

In dit geval zullen Maandag tot Vrijdag intern als 1 tot en met 5 bewaard worden, Zaterdag als 50, en Zondag als 60.

BMI met if

Pas je BMI-programma ([zie practica vorige hoofdstuk](#)) aan zodat je programma feedback aan de gebruiker geeft naargelang de berekende BMI.

- Onder de 18,5: ondergewicht.
- Tussen de 18,5 en de 24,9: normaal gewicht.
- Tussen de 25 en de 29,9: overgewicht. Je loopt niet echt een risico, maar je mag niet dikker worden.
- Tussen de 30 en de 39,9: Zwaarlijvigheid (obesitas). Verhoogde kans op allerlei aandoeningen zoals diabetes, hartaandoeningen en rugklachten. Je zou 5 tot 10 kg moeten vermageren.
- Boven de 40: ernstige zwaarlijvigheid. Je moet dringend vermageren want je gezondheid is in gevaar (of je hebt je lengte of gewicht in verkeerde eenheid ingevoerd)

Schoenverkoper

a) Maak een oefening die aan de gebruiker vraagt hoeveel paar schoenen hij wenst te kopen. Ieder paar schoenen kost steeds 20euro. Indien de gebruiker 10 paar koopt kosten ze maar 10euro. Toon aan de gebruiker de totale prijs.

b) Voeg nu toe dat het programma eerst aan de kassier vraagt vanaf hoeveel paar de klant korting zal krijgen. Als de gebruiker dus 5 indient zal reeds bij 5 paar de kost per paar 10euro zijn.

Quiz

Maak een quiz. Maak gebruik van het switch-statement om de input van de gebruiker (a,b,c of d) te verwerken en bij iedere vraag aan te geven of dit juist/fout is. Voorzie 3 multiple choice vragen. Houdt bij hoe vaak de speler juist antwoordde en geef op het einde de eindscore (Juist is +2 , fout is -1)

Zoek op hoe je de kleur van de letters en de achtergrond in een console-applicatie kunt aanpassen en pas dit toe op je quiz om deze er iets boeiender uit te laten zien. Toon iedere vraag op een nieuw scherm.

Herhalingen

Herhalingen (*Loops*) creëer je wanneer bepaalde code een aantal keer moet herhaald worden. Hoe vaak de herhaling moet duren is afhankelijk van de conditie die je hebt bepaald.

While

De syntax van een while loop is eenvoudig:

```
while (condition)
{
    // C# code to be performed while expression evaluates to true
}
```

Waarbij we, net als bij een if statement, de conditie uitgedrukt wordt als een booleaanse expressie met 1 of meerdere relationele operators.

Zolang de conditie true is zal de code binnen de accolades uitgevoerd worden. Indien dus de conditie reeds vanaf het begin false is dan zal de code binnen de while-loop niet worden uitgevoerd.

Een voorbeeld van een eenvoudige while loop:

```
int myCount = 0;

while (myCount < 100)
{
    myCount++;
    Console.WriteLine(myCount);
}
```

Zolang myCount kleiner is dan 100 ($myCount < 100$) zal myCount met 1 verhoogd worden en zal de huidige waarde van myCount getoond worden. Daar de test gebeurt aan het begin van de loop wil dit zeggen dat het getal 100 nog wel getoond zal worden. Test dit zelf!

Do while

In tegenstelling tot een while loop, zal een do-while loop sowieso **minstens 1 keer uitgevoerd worden**. Ongeacht de opgegeven conditie zal de do-while loop zijn code 1 keer uitvoeren. We herhalen deze zin uitdrukkelijk 2x zodat het verschil tussen beide type loops duidelijk blijft.

De syntax van een do-while is eveneens verraderlijk eenvoudig:

```
do{
    // C# code to be performed while expression evaluates to true
} while (condition);
```

Merk op dat achteraan de conditie een kommapunt na het ronde haakje staat. **Dit is een v  l voorkomende fout. Bij een while is dit niet!** Daar de test van een do-while achteraan de code van de loop gebeurt is het logisch dat een do-while dus minstens 1 keer wordt uitgevoerd Volgende eenvoudige aftelprogramma toont de werking van de do-while loop

```
int i = 10;
do
{
    i--;
    Console.WriteLine(i);
} while (i < 10);
```

Complexe condities

Uiteraard mag de conditie waaraan een loop moet voldoen complexer zijn door middel van de booleaanse logische operator. Volgende while bijvoorbeeld zal uitgevoerd worden zolang teller groter is dan 5 en de variabele naam van het type string niet gelijk is aan "tim":

```
while(teller > 5 && naam!="tim")
{
    //Keep repeating
}
```

Oneindige loops

Indien de loop-conditie nooit false wordt dan heb je een oneindige loop gemaakt. Soms is dit gewenst gedrag (bijvoorbeeld bij de gameloop) soms is dit een bug en zal je dit moeten debuggen.

Volgende twee voorbeelden tonen dit:

- Een bewuste oneindige loop:

```
while(true)
{
    //See you in infinity
}
```

- Een bug die een oneindige loop veroorzaakt:

```
int teller = 0;
while(teller<10)
{
    Console.WriteLine(teller);
    teller--;    //oops, dit had teller++ moeten zijn
}
```

Scope van variabelen in loops

Let er op dat de scope van variabelen bij loops zeer belangrijk is. Indien je een variabelen binnen de loop definieert dan zal deze steeds terug "gereset" worden wanneer de volgende cyclus van de loop start. Volgende code toont bijvoorbeeld **foutief** hoe je de som van de eerste 10 getallen ($1+2+3+\dots+10$) zou maken:

```
int teller=1;
while(teller<=10)
{
    int som=0;
    som= som+teller;
}
Console.WriteLine(som); //deze lijn zal fout genereren
```

De **correcte** manier om dit op te lossen is te beseffen dat de variabele som enkel binnen de accolades van de while-loop gekend is. Op de koop toe wordt deze steeds terug op 0 gezet en er kan dus geen som van alle teller-waarden bijgehouden worden:

```
int teller=1;
int som=0;
while(teller<=10)
{
    som= som+teller;
}
Console.WriteLine(som); //deze lijn zal fout genereren
```

For

Intro

Een veelvoorkomende manier van while-loops gebruiken is waarbij je een bepaalde teller bijhoudt die je telkens met een bepaalde waarde verhoogt. Wanneer de teller een bepaalde waarde bereikt moet de loop afgesloten worden.

Bijvoorbeeld volgende code om alle even getallen van 0 tot 10 te tonen:

```
int k = 0;
while(k<11)
{
    Console.WriteLine(k);
    k = k + 2;
}
```

For syntax

Met een for-loop kunnen we deze veel voorkomende code-constructie verkort schrijven, met volgende algemene syntax:

```
for (setup; finish test; update)
{
    // C# code to be performed while finish test evaluates to true
}
```

Gebruiken we deze kennis nu, dan kunnen we de eerder vermelde code om de even getallen van 0 tot en met 10 tonen als volgt:

```
for (int i = 0; i < 11; i=i+2)
{
    Console.WriteLine(i);
}
```

Voor de setup-variabele kiest men meestal i, maar dat is niet noodzakelijk. In de setup wordt dus een variabele op een start-waarde gezet. De finish test zal aan de start van iedere loop kijken of de finish test nog waar is, indien dat het geval is dan wordt een nieuwe loop gestart en wordt i met een bepaalde waarde, zoals in update aangegeven, verhoogd.

Lees zeker [deze for tutorial na](#) want er zijn nog enkele subtiliteiten in for-loops die we niet hier behandelen.

Foreach loops

Wanneer je geen indexering nodig hebt, maar toch snel over alle elementen in een array wenst te gaan, dan is het **foreach** statement een zeer nuttig is. Een foreach loop zal ieder element in de array een voor een in een tijdelijke variabele plaatsen (de **iteration variable**).

Volgende code toont de werking waarbij we een array van doubles hebben en alle elementen er in op het scherm willen tonen:

```
double[] killdeathRates= {1.2, 0.89, 3.15, 0.1};
foreach (double kdrate in killdeathRates)
{
    Console.WriteLine($"Kill/Death rate is {kdrate}");
}
```

De eerste keer dat we in de loop gaan zal het element `killdeathRates[0]` aan `kdrate` toegewezen worden voor gebruik in de loop-body, vervolgens wordt `killdeathRates[1]` toegewezen, enz.

Het voordeel is dat je dus geen teller/index nodig hebt en dat foreach zelf de lengte van de array zal bepalen.

Opgelet bij het gebruik van foreach loops

- De foreach iteration variable is *read-only*: je kan dus geen waarden in de array aanpassen, enkel uitlezen.
- De foreach gebruik je enkel als je alle elementen van een array wenst te benaderen. In alle andere gevallen zal je een ander soort loop (for, while, etc.) moeten gebruiken.

var keyword

C# heeft een `var` keyword. Je mag dit keyword gebruiken ter vervanging van het type (bv int) op voorwaarde dat de compiler kan achterhalen wat het type moet zijn.

```
var getal= 5; //var zal int zijn
var myArray= new double[20]; //var zal double[] zijn
var tekst= "Hi there handsome"; //var zal string zijn
```

Opgelet: het `var` keyword is gewoon een *lazy programmer syntax toevoeging* om te voorkomen dat je als programmer niet constant het type moet schrijven (vooral handig bij [klassen en objecten later](#)). Bij javascript heeft var een totaal andere functie: het zegt eigenlijk "het type dat je in deze variabele kan steken is...variabel", m.a.w. het kan de ene keer een string zijn, dan een int. Bij C# gaat dit niet: eens je een variabele aanmaakt dan zal dat type onveranderbaar zijn.

JavaScript is a dynamically typed language, while c# is (usually) a statically typed language (stackoverflow.com)

var en foreach

Wanneer je de Visual Studio [code snippet](#) voor foreach gebruikt `foreach [tab][tab]` dan zal deze code ook een var gebruiken voor de iteration variabele. De compiler kan aan de te gebruiken array zien wat het type van een individueel element in de array moet zijn. De foreach van zonet kan dus herschreven worden naar:

```
foreach (var kdrate in killdeathRates)
{
    Console.WriteLine($"Kill/Death rate is {kdrate}");
}
```

Programma flow analyse

Door een flowchart op te stellen is het vaak veel eenvoudiger om een programma ofwel te analyseren (van code naar idee) ofwel om een programma te schrijven (van idee naar code).

Een flowchart (letterlijk: *stroomkaart*) of stroomdiagram is een schematische beschrijving van een proces. Met een flowchart kan je vaak ingewikkelde stukken code visualiseren waardoor het geheel plots niet meer zo ingewikkeld is.

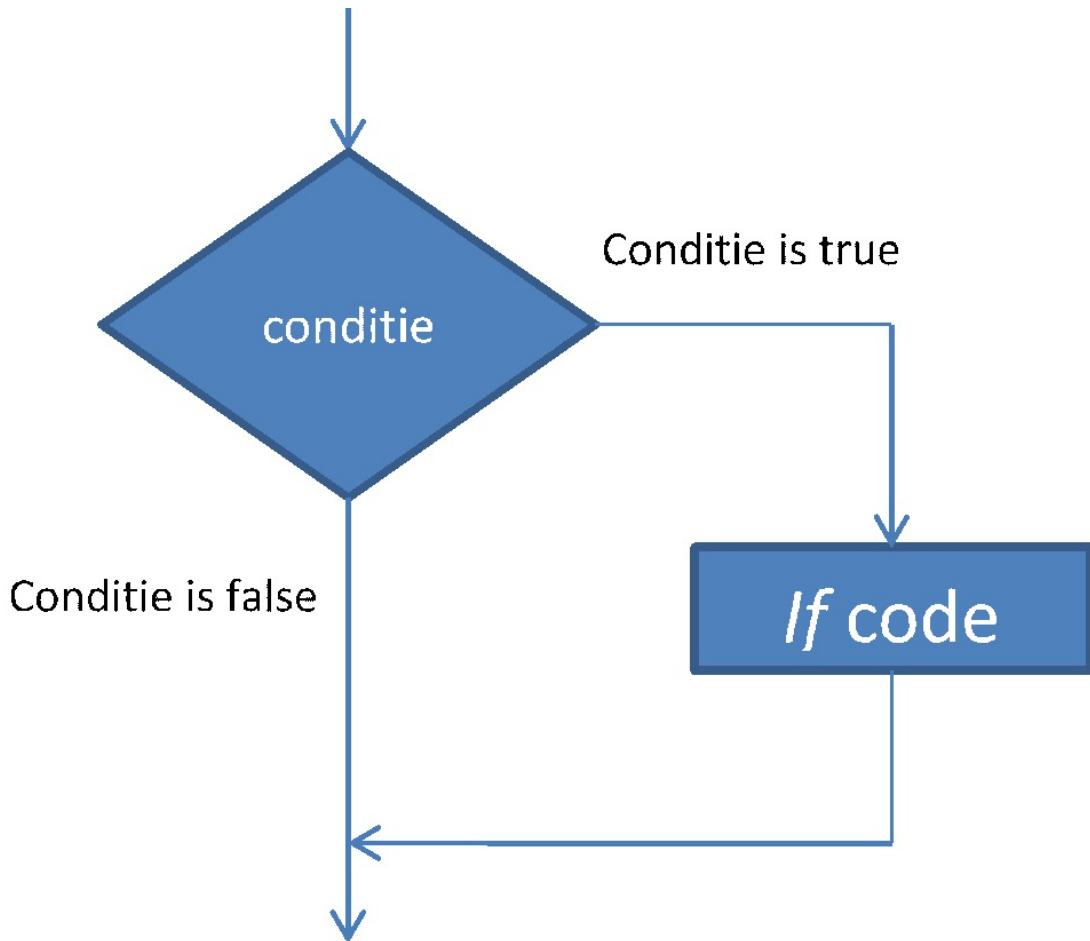
Een flowchart bestaat uit een aantal elementen:

- Pijl: een pijl geeft aan naar welk volgende blok wordt gegaan. Indien boven de pijl een bepaalde waarde staat wil dit zeggen dat deze pijl enkel wordt gevuld als de uitkomst van het vorige blok de getoonde waarde geeft.
- Start en einde: aangegeven met een cirkel met daarin de woorden "Start" of "Einde"
- Verwerk-stap: een statement zoals "Voeg 1 toe aan X" wordt in een rechthoek geplaatst. Alle code die geen invoer nodig heeft zet je in een rechthoek.
- Input/output: Een parallelogram gebruik je om in-of uitvoer van de gebruiker of scherm te tonen. Bv "Verkijg X van gebruiker" of "Toon volgende zin op het scherm".
- Condities en beslissingen: Een ruit wordt gebruikt wanneer een beslissing moet genomen worden. De condities van if en while-loops zet je dus in een ruit. De pijlen die eruit volgen geven aan welke pijl moet gevuld worden gegeven een bepaalde waarde van de conditie.

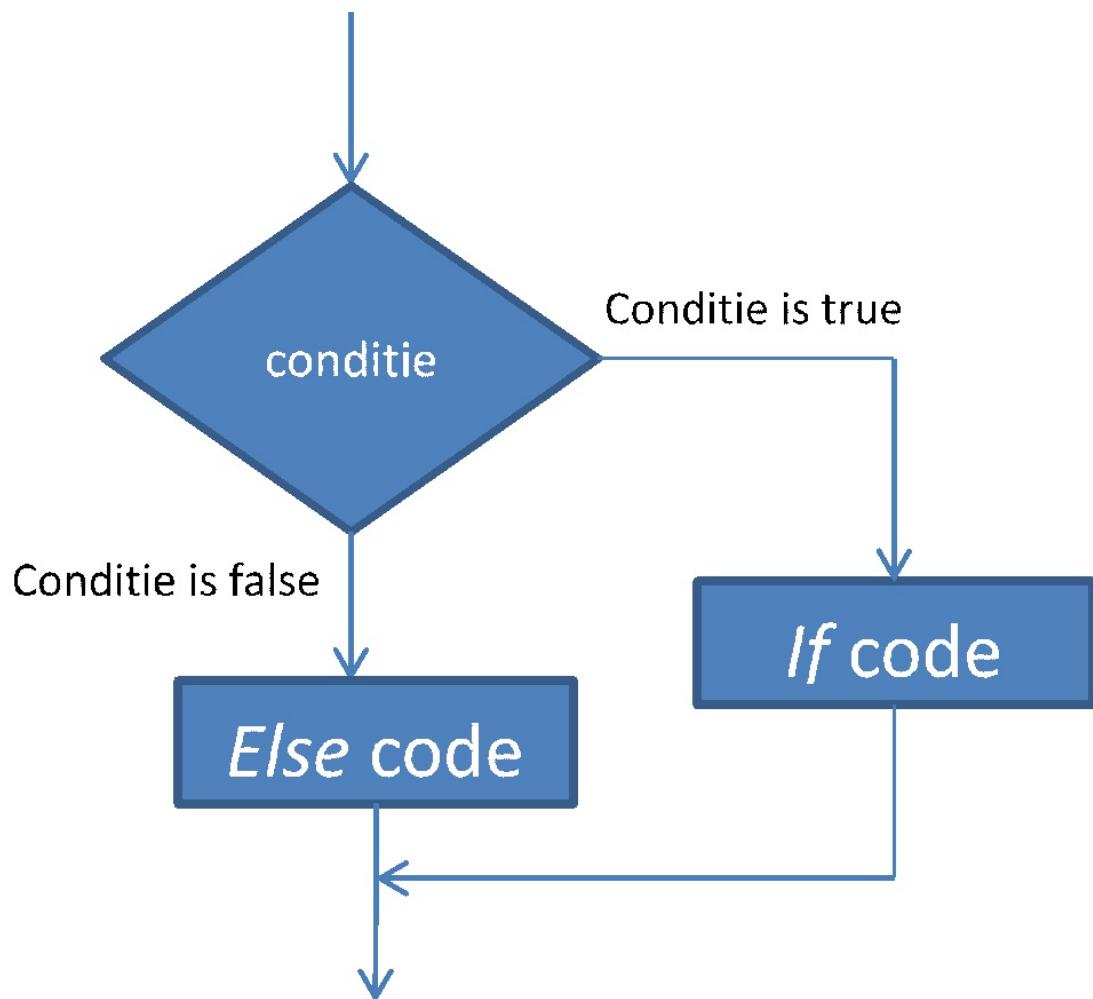
Flow-elementen

We tonen nu kort de verschillende program flow elementen en hoe ze in een flowchart voorkomen.

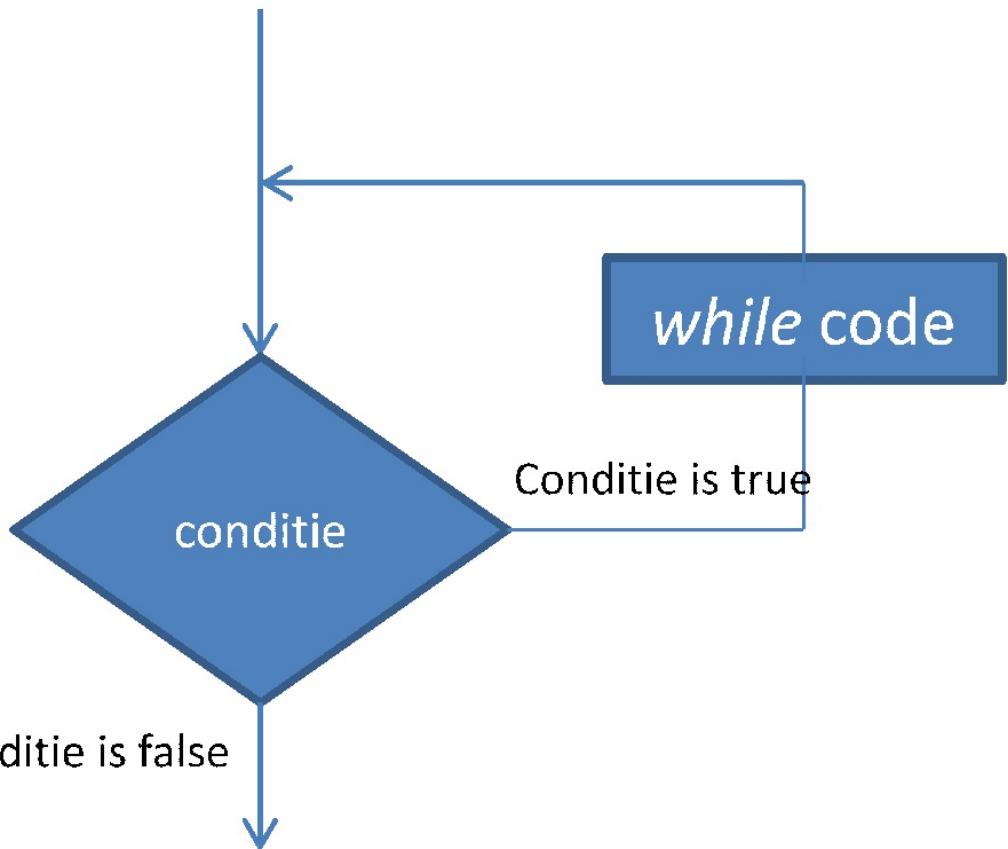
If-element



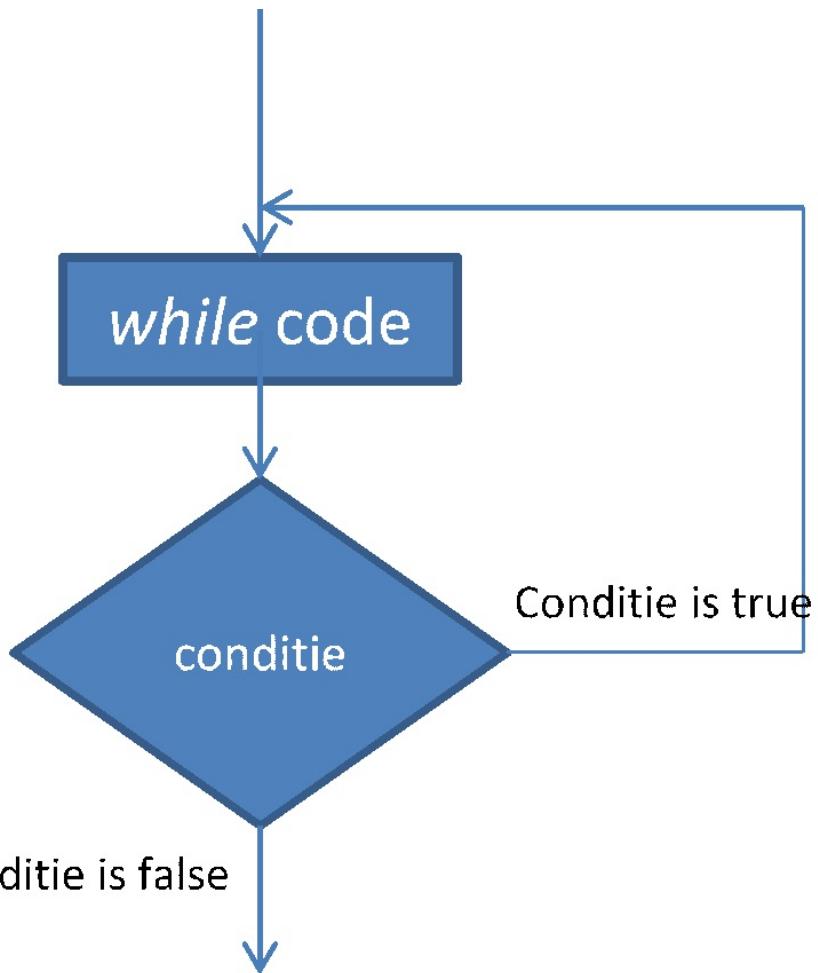
If-else element



While-element

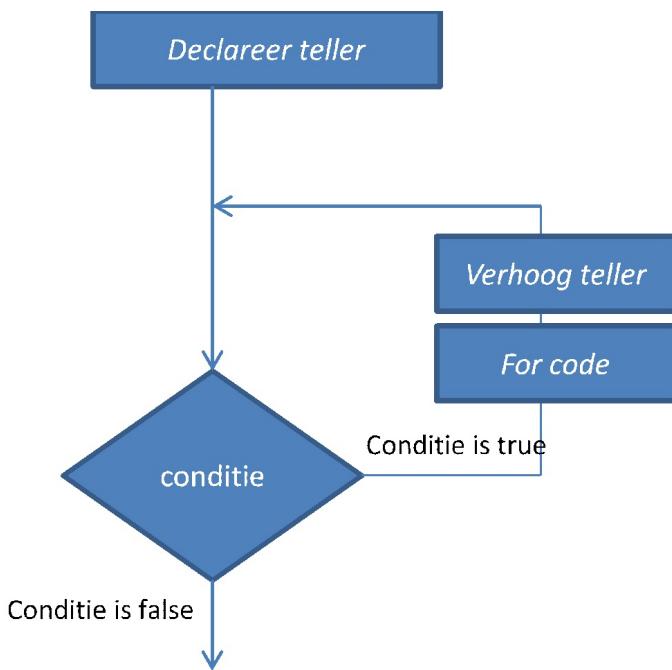


Do while-element



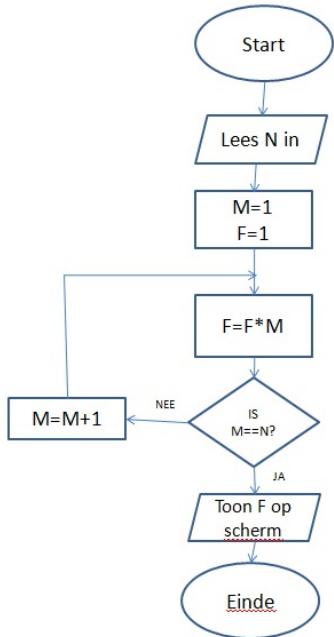
Merk op dat bij if en if-else de flow niet naar een eerder punt in de code gaat. Dit is dus de manier om een while/do while te herkennen: er wordt naar een eerder punt in de code gegaan, een punt waar we reeds geweest waren

For-element



Voorbeeld flowchart

Door de eerder beschreven elementen nu samen te voegen kunnen we een programma als een flowchart voorstellen. Stel dat we een programma "Faculteit" maken. Hierin is het de bedoeling om de faculteit van een gegeven getal N dat door de gebruiker wordt ingevoerd, te berekenen (bijvoorbeeld $N=5$ geeft dan $5! = 5 \times 4 \times 3 \times 2 \times 2 = 120$). De finale flowchart ziet er als volgt uit:



Zoals verteld kunnen we een flowchart in beide richtingen gebruiken. We hebben de flowchart hiervoor gemaakt, gebaseerd op onze oplossing van het vorige labo. Maar stel dat je deze flowchart krijgt, dan kan je dus ook deze chart rechtstreeks omzetten in C#.

Prijzen met foreach

Maak een array die tot 20 prijzen (double) kan bewaren. Vraag aan de gebruiker om 20 prijzen in te voeren en bewaar deze in de array. Doorloop vervolgens m.b.v. een foreach-lus de volledige array en toon enkel de elementen op het scherm wiens prijs hoger of gelijk is aan €5.00. Toon op het einde van het programma het gemiddelde van alle prijzen (dus inclusief de lagere prijzen).

For doordenker

Schrijf een programma dat de volgende output geeft , gegeven dat de gebruiker een maximum waarde invoert , dus als hij 4 ingeeft dan zal de driehoek maximum 4 breed worden. Gebruik enkel forloops!

```
*
```

```
**
```

```
***
```

```
****
```

```
***
```

```
**
```

```
*
```

For doordenker extra

Schrijf een programma dat de volgende output geeft (zie ook WhileDoordenker van vorige labo), gegeven dat de gebruiker een maximum waarde invoert die opgeeft uit hoeveel lijnen de boom bestaat. Maak enkel gebruik van for-loops.

```
*
```

```
***
```

```
*****
```

```
*****
```

```
*****
```

```
*****
```

```
*****
```

Euler project

Maak volgende opdracht van [projecteuler.net](#):

Indien we alle natuurlijke getallen van 0 tot en met 10 oplijsten die een meervoud van 3 of 5 zijn, dan krijgen we de getallen 3,5,6 en 9. De som van deze 4 getallen is 23. Maak nu een programma dat de som van alle veelvouden van 3 of 5 weergeeft onder van 0 tot 1000.

Grootste getal

Vervolledig deze code zodat ook getoond wordt welke de grootste waarde is die werd ingevoerd.

```
int x=0;
int y=0;
do
{
    y=y+x;
    Console.WriteLine("Voer gehele waarden in (32767=stop)");
    string instring= Console.ReadLine();
    x= Convert.ToInt32(instring);
    //.....
}while (x != 32767)
Console.WriteLine(y);
```

Boekhouder

1° Maak een 'boekhoud-programma': de gebruiker kan continu positieve en negatieve getallen invoeren. Telkens hij op enter duwt wordt de huidige invoer aan de balans bijgevoegd. Je houdt volgende zaken bij:

- De balans van alle ingevoerde getallen: dit is gewoon de som van de getallen. Als de gebruiker dus de getallen 4,-10, 8 invoerde dan

- zal de balans op +2 staan (4 -10 +8)
- De som van alle negatieve invoeren. Als de gebruiker dus 4,-10,8,-6 invoerde dan zal dit getal op -16 staan (= -10 -6) .
 - De som van alle positieve invoeren. Als de gebruiker dus 4,-10,8,-6 invoerde dan zal dit getal op +12 staan (= 4+8) .
 - Het gemiddelde van alle ingevoerde getallen

Deze 4 getallen worden steeds geüpdate en getoond wanneer de gebruiker een nieuw getal invoert en op enter duwt.

Raad het getal

Deel 1: Teken de flowchart van het volgende programma.

```
int getal;
int poging = 0;
string pogingString;
bool gevonden= false;
Random rand= new Random();
getal= rand.Next(0,10);

while(!gevonden)
{
    Console.WriteLine("Geef een getal tussen 0 en 10");
    pogingString= Console.ReadLine();
    poging = int.Parse(pogingString);

    if(getal>poging)
    {
        Console.WriteLine("Het gezochte getal is groter, probeer opnieuw.");
    }
    else if(getal<poging)
    {
        Console.WriteLine("Het gezochte getal is kleiner, probeer opnieuw.");
    }
    else
        gevonden=true;
}
Console.WriteLine($"Gevonden! Het te zoeken getal was inderdaad {getal} je had er {poging} pogingen voor nodig.");
```

Deel 2: Pas het programma zo aan dat de tekst die verschijnt bij het inlezen van de volgende poging het resterende interval aangeeft. Gebruik hiervoor twee extra variabelen "ondergrens" en "boven grens" die als beginwaarden respectievelijk 0 en 100 krijgen. Bij het inlezen van de volgende waarde voor poging zal deze toegekend worden aan ondergrens of boven grens naargelang ze groter dan wel kleiner dan het te zoeken getal is.

```
C:\Windows\system32\cmd.exe
Geef een getal tussen 0 en 100
50
Het gezochte getal is kleiner. Probeer opnieuw.
Geef een getal tussen 0 en 50
25
Het gezochte getal is kleiner. Probeer opnieuw.
Geef een getal tussen 0 en 25
12
Het gezochte getal is kleiner. Probeer opnieuw.
Geef een getal tussen 0 en 12
10
Het gezochte getal is kleiner. Probeer opnieuw.
Geef een getal tussen 0 en 10
6
Gevonden! Het te zoeken getal was inderdaad 6
Press any key to continue . . .
```

Zie hieronder een voorbeeldoutput:

Deel 3: Bouw in het programma een controle in die er voor zorgt dat je geen waarde meer kan ingeven die buiten het opgegeven interval ligt (dus ook indien de grenzen aangepast worden moet deze controle blijven werken!) **Deel 4:** Pas het programma aan zodat er een maximum aantal pogingen is toegestaan. Om dit maximum te bepalen moet je uitgaan van de beginwaarden van "ondergrens" en "boven grens". Ga dus na hoeveel pogingen er in het ideale geval (bij het zoeken telkens de middelste waarde als nieuwe poging ingeven) maximaal nodig zijn om een interval te doorzoeken.

Steen schaar papier

Maak een applicatie waarbij de gebruiker steen-schaar-papier met de computer kan spelen. De gebruiker kiest telkens steen, schaar of papier en drukt op enter. Vervolgens kiest de computer willekeurig steen, schaar of papier (gebruik de Random.Next() methode, waarbij je deze tussen 1 en 3 laat varieren). Vervolgens krijgt de winnaar 1 punt:

- Steen wint van schaar, verliest van papier
- Papier wint van steen, verliest van schaar
- Schaar wint van papier, verliest van steen

- Indien beide hetzelfde hebben wint niemand een punt.

Op het scherm wordt telkens getoond wie de huidige ronde heeft gewonnen en hoeveel de tussenscore is. De eerste (pc of gebruiker) die 10 punten haalt wint.

Teken een flowchart van je applicatie.

Tafels van vermenigvuldigen

Gebruik de krach van loops om pijsnel alle tafels van 1 tot en 10 van vermenigvuldigen oph et scherm te tonen (dus van 1x1 tot 10x10 en alles daartussen)

Password generator

Ontwerp een consoletoepassing waarmee je een wachtwoord genereert voor een gebruiker. Het wachtwoord is opgebouwd uit:
de 2 eerste letters van de familienaam: de 1ste letter is een hoofdletter, de 2de letter is een kleine letter. Daarna worden de 2 letters gewisseld; het zonenummer van het telefoonnummer zonder de 0; het eerste cijfer van de postcode in het kwadraat.

Become Neo

Volgende code genereert een beeld zoals dat ook in de cultfilm The Matrix plaatsvindt.

```
Random rangen = new Random();
Console.ForegroundColor = ConsoleColor.Green;
while (true)
{
    //Genereer nieuw random teken:
    char teken = Convert.ToChar(rangen.Next(62, 400));
    //Zet teken op scherm
    Console.Write(teken);

    //Ietwat vertragen
    System.Threading.Thread.Sleep(10); //dit getal is in milliseconden. Speel er gerust mee.

    //Af en toe donker kleurtje
    if(rangen.Next(0,3)==0)
    {
        Console.ForegroundColor = ConsoleColor.DarkGreen;
    }
    else
    {
        Console.ForegroundColor = ConsoleColor.Green;
    }
}
```

Vul de code aan zodat de karakters random kleuren krijgen. Kan je het nog cooler maken?

Methoden

Veel code die we hebben geschreven wordt meerdere keren, al dan niet op verschillende plaatsen, gebruikt. Dit verhoogt natuurlijk de foutgevoeligheid. Door het gebruik van methodes kunnen we de foutgevoeligheid van de code verlagen daar de code maar op één plek, één keer dient geschreven te worden. Echter, ook de leesbaarheid en dus onderhoud van de code wordt verhoogd.

Wat is een methode

Een methode, ook vaak functie genoemd, is in C# een stuk code ('block') bestaande uit een 0, 1 of meerder statements. De methode kan herhaaldelijk opgeroepen worden, al dan niet met extra parameters, en kan ook een resultaat terug geven.

De basis-syntaxis van een methode is de volgende indien je een methode in je hoofdprogramma wenst te schrijven (de werking van het keyword static zien we later):

```
static returntype MethodeNaam(parameters)
{
    //code van methode
}
```

Vervolgens kan je deze methode elders oproepen als volgt, indien de methode geen parameters vereist:

```
MethodeNaam();
```

Indien er wel parameters nodig zijn dan geef je die mee als volgt, het is belangrijk dat de volgorde van de parameters gehanteerd wordt zoals je in de methode zelf hebt beschreven.

```
MethodeNaam(parameter1, parameter2, ...);
```

Returntypes

Het returntype van een methode geeft aan wat het type is van de data die de methode als resultaat teruggeeft bij het beëindigen ervan. Eender welk type dat je kent kan hiervoor gebruikt worden, zoals int, string, char, float, etc. Maar ook klassen (zie later) zoals Student, Canvas, etc.

Het is belangrijk dat in je methode het resultaat ook effectief wordt teruggegeven, dit doe je met het keyword `return` gevuld door de variabele die moet teruggegeven worden. Denk er dus aan dat deze variabele van het type is dat je hebt opgegeven als zijnde het returntype. Van zodra je `return` gebruikt zal je op die plek uit de methode 'vliegen'.

Een voorbeeld eenvoudig voorbeeld. Volgende methode geeft de naam van de auteur van je programma terug

```
static string GetNameAuthor()
{
    string name = "Tim Dams";
    return name;
}
```

Mogelijke manieren om deze methode in je programma te gebruiken zouden kunnen zijn:

```
string myName = GetNameAuthor();
```

Of bijvoorbeeld ook:

```
Console.WriteLine("This program is written by " + GetNameAuthor());
```

Een ander voorbeeld is bijvoorbeeld een methode die de faculteit van 5 berekent. We tonen een voorbeeldprogramma die deze methode gebruikt. De oproep van de methode gebeurt vanuit de Main-methode:

```
partial class Program
{
```

```

static int FaculteitVan5()
{
    int resultaat = 1;
    for (int i = 1; i <= 5; i++)
    {
        resultaat *= i;
    }
    return resultaat;
}

static void Main(string[] args)
{
    Console.WriteLine("Faculteit van 5 is {0}", FaculteitVan5());
}

```

Void returntype

Indien je methode niets teruggeeft wanneer de methode eindigt (bijvoorbeeld indien de methode enkel tekst op het scherm toont) dan dient je dit ook aan te geven. Hiervoor gebruik je het keyword void. Een voorbeeld

```

static void ShowProgramVersion()
{
    Console.Write("The version of this program is: ");
    Console.Write(2.16 + "\n");
}

```

Parameters doorgeven

Parameters kunnen op 2 manieren worden doorgegeven aan een methode:

1. Wanneer een parameter **by value** wordt meegegeven aan een methode, dan wordt een kopie gemaakt van de huidige waarde die wordt meegegeven.
2. Wanneer echter een parameter **by reference** wordt meegegeven dan zal een pointer worden meegegeven aan de methode. Deze pointer bevat het adres van de eigenlijke variabele die we meegeven. Aanpassingen aan de parameters zullen daardoor ook zichtbaar zijn binnen de scope van de originele variabele.

Parameters doorgeven by value

Je methode definitie kan ook 1 of meerdere parameters bevatten. Hierbij gebruik je volgende syntax:

```

static returntype MethodeNaam(type parameter1, type parameter2)
{
    //code van methode
}

```

Deze parameters zijn nu beschikbaar binnen de methode om mee te werken naar believen.

Stel bijvoorbeeld dat we onze FaculteitVan5 willen veralgemenen naar een methode die voor alle getallen werkt, dan zou je volgende methode kunnen schrijven:

```

static int BerekenFaculteit(int grens)
{
    int resultaat = 1;
    for (int i = 1; i <= grens; i++)
    {
        resultaat *= i;
    }
    return resultaat;
}

static void Main(string[] args)
{
    int getal = 5;
    Console.WriteLine("Faculteit van {0} is {1}", getal, BerekenFaculteit(getal));
}

```

Dit geeft als uitvoer: Faculteit van 5 is 120; Je zou nu echter de waarde van getal kunnen aanpassen (door bijvoorbeeld aan de gebruiker te vragen welke faculteit moet berekend worden) en je code zal nog steeds werken.

Stel bijvoorbeeld dat je de faculteiten wenst te kennen van alle getallen tussen 1 en 10, dan zou je schrijven:

```
for (int i = 1; i < 11; i++)
{
    Console.WriteLine("Faculteit van {0} is {1}", i, BerekenFaculteit(i));
}
```

Dit zal als resultaat geven

```
Faculteit van 1 is 1
Faculteit van 2 is 2
Faculteit van 3 is 6
Faculteit van 4 is 24
Faculteit van 5 is 120
Faculteit van 6 is 720
Faculteit van 7 is 5040
Faculteit van 8 is 40320
Faculteit van 9 is 362880
Faculteit van 10 is 3628800
```

Merk dus op dat dankzij je methode, je v  l code maar   n keer moet schrijven wat de kans op fouten gevoelig verlaagt.

Volgorde van parameters

De volgorde waarin je je parameters meegeeft bij de aanroep van een methode is belangrijk. De eerste variabele wordt aan de eerste parameter otekendeend, en zo voort. Volgende voorbeeld toont dit. Stel dat je een methode hebt:

```
static void ToonDeling(double teller, double noemer)
{
    string result=Convert.ToString( teller/noemer);
    Console.WriteLine(teller/noemer);
}
```

Stel dat we nu in onze main volgende aanroep doen:

```
double n= 4.2;
double t= 5.2;
ToonDeling(n,t);
```

Dit zal een ander resultaat geven dan wanneer we volgende code zouden uitvoeren:

```
ToonDeling(t,n);
```

Ook de volgorde is belangrijk zeker wanneer je met verschillende types als parameters werkt:

```
static void ToonInfo(string name, int age)
{
    Console.WriteLine($"{name} is {age} old");
}
```

Deze aanroep is correct:

```
ToonInfo("Tim",37);
```

Deze is **FOUT** en zal niet compileren:

```
ToonInfo(37, "Tim");
```

Commentaar toevoegen

Het is aan te raden om steeds boven een methode een Block-commentaar te zetten dat volgende velden bevat:

- Doel van de methode
- Wat de methode teruggeeft
- Beschrijving van de benodigde parameters

- Zet IN voor de parameter indien deze enkel als IN-parameter dient (dus de parameters die by value worden meegegeven)
- Zet OUT voor de parameter indien deze enkel als OUT-parameter dient (dus de parameters die als out [type] worden gedefinieerd)
- Zet IN-OUT voor de parameter indien deze als IN en OUT-parameter dient (dus de parameters die als ref [type] worden gedefinieerd)

Stel bijvoorbeeld dat je een methode hebt geschreven die de oppervlakte berekent van een rechthoek, dan zou de commentaar en bijhorende methode als volgt zijn:

```
/*
 * OppervlakteDriehoek: Deze methode berekent de oppervlakte van een driehoek
 *
 * Return: De oppervlakte van de driehoek
 *
 * Parameters:
 * IN: double basis: lengte van de basis van de driehoek >0
 * IN: double hoogte: hoogte van de driehoek >0
 *
 */
static double OppervlakteDriehoek(double basis, double hoogte)
{
    double resultaat = 0;
    resultaat = ( basis * hoogte ) / 2;
    return resultaat;
}
```

Voor het vorige voorbeeld is commentaar bij de methode schrijven redelijk nutteloos, daar zowel de methodenaam als de parameters duidelijk weergeven wat de methode zal doen. Merk echter op dat wanneer een methode een complexer probleem oplost dat duidelijke commentaar onontbeerlijk zal zijn. Het is daarom een 'good practice' (Ned. Goede gewoonte) om steeds bij methoden een stukje commentaar te geven.

Nut van methoden

Een eenvoudig voorbeeld (bron: handboek Visual C# 2008, Dirk Louis) waar het gebruik van methoden onmiddellijk duidelijk wordt. Stel, je hebt 15000 euro op een spaarrekening vastgezet waarvoor de bank u een rente geeft van 3,5%. Nu wil je natuurlijk weten hoe je kapitaal hoe je kapitaal van jaar tot jaar groeit. Stel dat je aan de verleiding weerstaat en de jaarlijkse rente niet opneemt, maar op de spaarrekening laat staan. Je berekent dan je kapitaal na n jaren met de volgende formule:

eindkapitaal = startkapitaal $\times (1 + (\text{rentepercentage}/100))^n$ (^ is tot de macht in pseudocode)

Nu kan je berekenen hoeveel geld je de volgende zeven jaren verdient, het bijhorende programma ziet er zo uit:

```
static void Main(string[] args)
{
    double startKapitaal = 15000;
    double rentepercentage = 3.5;
    double eindKapitaal;

    //Berekening eindKapitaal
    eindKapitaal = startKapitaal * Math.Pow((1 + rentepercentage / 100), 1);
    Console.WriteLine("Na 1 jaar:" + (int)eindKapitaal + "euro");

    eindKapitaal = startKapitaal * Math.Pow((1 + rentepercentage / 100), 2);
    Console.WriteLine("Na 2 jaar:" + (int)eindKapitaal + "euro");

    eindKapitaal = startKapitaal * Math.Pow((1 + rentepercentage / 100), 3);
    Console.WriteLine("Na 3 jaar:" + (int)eindKapitaal + "euro");

    eindKapitaal = startKapitaal * Math.Pow((1 + rentepercentage / 100), 4);
    Console.WriteLine("Na 4 jaar:" + (int)eindKapitaal + "euro");

    eindKapitaal = startKapitaal * Math.Pow((1 + rentepercentage / 100), 5);
    Console.WriteLine("Na 5 jaar:" + (int)eindKapitaal + "euro");

    eindKapitaal = startKapitaal * Math.Pow((1 + rentepercentage / 100), 6);
    Console.WriteLine("Na 6 jaar:" + (int)eindKapitaal + "euro");

    eindKapitaal = startKapitaal * Math.Pow((1 + rentepercentage / 100), 7);
    Console.WriteLine("Na 7 jaar:" + (int)eindKapitaal + "euro");
}
```

Dit geeft als uitvoer:

```
Na 1 jaar:15524euro  
Na 2 jaar:16068euro  
Na 3 jaar:16630euro  
Na 4 jaar:17212euro  
Na 5 jaar:17815euro  
Na 6 jaar:18438euro  
Na 7 jaar:19084euro
```

Het programma werkt naar behoren, maar zoals je zelf kan zien wordt er aardig wat code herhaalt, op enkele kleine details na. Bij iedere berekening en het tonen van de interest verandert enkel de macht en het aantal jaar. Als er nu een fout in je interestberekening zou staan dan zal je die op 7 plaatsen telkens moeten veranderen.

Rente berekenen verbeterd, versie 1

We kunnen nu terug naar onze rente-berekenaar en dit programma aanzienlijk vereenvoudigen door gebruik te maken van methoden. Namelijk als volgt:

```
public static void RenteOpRenteBerekenen(double looptijd)  
{  
    double startKapitaal = 15000;  
    double rentepercentage = 3.5;  
    double eindKapitaal;  
  
    //Berekening eindkapitaal  
    eindKapitaal = startKapitaal * Math.Pow((1 + rentepercentage / 100), looptijd);  
    Console.WriteLine("Na " + (int)looptijd+"jaar:" + (int)eindKapitaal + "euro");  
}  
  
static void Main(string[] args)  
{  
    RenteOpRenteBerekenen(1);  
    RenteOpRenteBerekenen(2);  
    RenteOpRenteBerekenen(3);  
    RenteOpRenteBerekenen(4);  
    RenteOpRenteBerekenen(5);  
    RenteOpRenteBerekenen(6);  
    RenteOpRenteBerekenen(7);  
}
```

Dit programma zal dezelfde output geven als het originele programma, maar de code is aanzienlijk verkleint en minder foutgevoelig (je moet maar op één plek je interestberekening aanpassen indien nodig). (Merk op dat we uiteraard de main kunnen verbeteren m.b.v. een for-loop: `for(int i=0;i<8;i++) {RenteOpRenteBerekenen(i);}`)

Rente berekenen verbeterd, versie 2

Je code opdelen in methoden is een zeer goede eerste stap naar modulair programmeren: kleine stukken code die ieder een eigen verantwoordelijkheid hebben. Om perfect modulair te zijn moet een methode zo praktische en algemeen mogelijk blijven, zodat de methode herbruikbaar is in andere projecten.

In het vorige voorbeeld is de methode van de renteberekening niet perfect modulair. Stel dat je later in het programma opnieuw de rente wil berekenen maar niet het resultaat op het scherm wil tonen. Of stel dat je de rente wil berekenen met een andere percentage, dan kunnen we de eerder geschreven methode dus niet gebruiken.

Modulair programmeren: indien je modulair wenst te programmeren moet je je aan volgende zaken houden:

- Beperk de methode strikt tot het uitvoeren van de opgedragen taak. Dus in het voorbeeld: alleen de renteberekening en geen verdere verwerking van de resultaten.
- Als de methode een waarde teruggeeft, declareer hiervoor dan een passende returnwaarde. Geef alle grootheden waarmee je de werkwijze van de methode wilt aanpassen mee aan de methode als parameter. In dit voorbeeld zijn dat dus de variabelen startkapitaal, rentepercentage en looptijd.

De nieuwe algemene, verbeterde methode wordt dan:

```
public static double RenteOpRenteBerekenen(double startKapitaal, double rentepercentage, double looptijd)  
{  
    double eindKapitaal;
```

```
//berekenig eindkapitaal  
eindkapitaal = startkapitaal * Math.Pow((1 + rentepercentage / 100), looptijd);  
  
return eindkapitaal;  
}
```

De aanroep van deze methode in de main wordt dan de volgende:

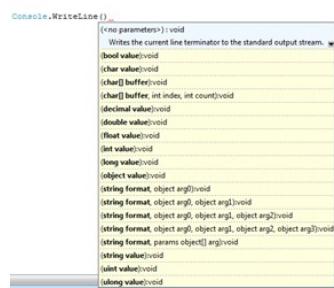
```
double eindkapitaal;  
eindkapitaal = RenteOpRenteBerekenen(15000, 3.5, 7);  
Console.WriteLine("Het eindbedrag na 7 jaar:" +(int)eindkapitaal);
```

Bestaande methoden en bibliotheken

Je herkent een methode aan de ronde haakjes na de methodenaam. Je hebt dus reeds een aantal methoden gebruikt zonder dat je het wist, denk maar aan `WriteLine()`, `ReadLine()` en `Parse()`

Dit zijn dus alle 3 methoden: stukken code die een specifieke taak uitvoeren.

Sommige methoden, zoals `WriteLine()`, vereisen dat je een aantal parameters meegeeft. De parameters dien je tussen de ronde haakjes te zetten. Hierbij is het uiterst belangrijk dat je de volgorde respecteert die de ontwikkelaar van de methode heeft gebruikt. Indien je niet weet wat deze volgorde is kan je altijd Intellisense gebruiken. Typ gewoon de methode in je code en stop met typen na het eerste ronde haakje, vervolgens verschijnen alle mogelijke manieren waarop je deze methoden kan oproepen..



Tussen de haakjes zien we welke parameters en hun type je mag meegeven aan de methode, gevolgd door het return-type van de methode en een eventuele beschrijving (merk dus op dat je de `WriteLine`-methode ook mag aanroepen zonder parameters, dit zal resulteren in een lege lijn in de console).

Met behulp van de F1-toets kunnen meer info over de methode in kwestie tonen. Hiervoor dien je je cursor op de Methode in je code te plaatsen, en vervolgens op F1 te drukken.

Voor `WriteLine` geeft dit:

A screenshot of Intellisense showing the detailed description for the `TextTransformation.WriteLine` method. The description states: "Writes text and the default line terminator to the [GenerationEnvironment](#). When the text template transformation process is completed, [GenerationEnvironment](#) becomes the final generated text output." Below this, the "Overload List" is expanded, showing two overloads: `WriteLine (String)` and `WriteLine (String, Object[])`. Both descriptions mention that they append a copy of the specified string and the default line terminator to the generated text output.

In de overload list zien we de verschillende manieren waarop je de methode in kwestie kan aanroepen. JE kan op iedere methode klikken voor meer informatie en een codevoorbeeld.

Math-bibliotheek

De Math namespace bevat aardig wat handige methoden. Deze bibliotheek die deel uitmaakt van het C# bevat methoden voor een groot aantal typische wiskundige methoden zowel Sinus, Cosinus, Vierkantswortel, Macht, Afronden, etc.

Type als code `Math` gevolgd door een `.` en je krijgt alle methoden van de Math-library te zien te zien, aangeduid met een paars blokje:



PI

Ook het getal `PI`, 3.14159265358979323846 , is beschikbaar in de Math-library. Het witte icoontje voor `PI` bij Intellisense toont aan dat het hier om een 'field' gaat; een eenvoudige variabele met een specifieke waarde. In dit geval gaat het zelfs om een const field, met de waarde van `PI` van het type `double`.

```
public const double PI;
```

Je kan deze gebruiken in berekeningen als volgt:

```
double straal= 5.5;
double omtrek= Math.PI * 2 * straal;
```

Intellisense

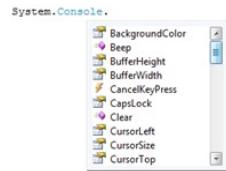
"Hoe kan je deze methode nu gebruiken?" is een veelgestelde vraag. Zeker wanneer je de basis van C# onder knie hebt en je stilletjes aan met bestaande .NET bibliotheken wil gaan werken. Wat volgt is een essentieel onderdeel van VS dat veel gevloek en tandengeknars zal voorkomen.

De help-files van VS zijn zeer uitgebreid en dankzij IntelliSense krijg je ook aardig wat informatie tijdens het typen van de code zelf.

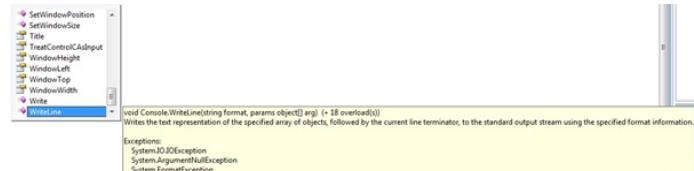
Type daarom onder vorige de WriteLine-zin het volgende:

```
System.Console.
```

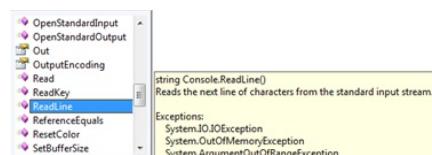
Wacht nu even en er zal na het . een lijst komen van methoden en fields die beschikbaar zijn. Je kan hier met de muis doorheen scrollen en zo zien welke methoden allemaal bij de Console klasse horen



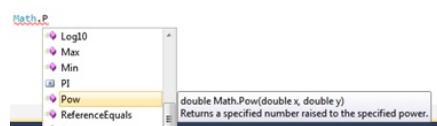
Scroll maar eens naar de WriteLine-methode en klik er op. Je krijgt dan extra informatie over die methode te zien:



Doe hetzelfde voor de ReadLine methode:



Je ziet bovenaan string `Console.ReadLine()` staan. Bij de WriteLine stond er void `Console.WriteLine()`. Die void wil zeggen dat de methode WriteLine niets terugstuurt. In tegenstelling tot ReadLine dat een string teruggeeft. Indien de methode één of meerdere parameters vereist dan zullen deze hier ook getoond worden:



De Math.Pow methode vereist dus bijvoorbeeld 2 parameters van het type double. Wanneer je nu begint te typen dan zal intellisense tonen waarvoor iedere parameter staat wanneer je aan die parameter gaat beginnen typen:

```
Math.Pow(
    double Math.Pow(double x, double y)
    Returns a specified number raised to the specified power.
    x: A double-precision floating-point number to be raised to a power.
```


Parameters by reference doorgeven

Je kan parameters op 2 manieren by reference doorgeven:

- Indien de parameters reeds een waarde heeft dan kan je het ref keyword gebruiken. Dit gebruik je dus voor in/out-parameters.
- Indien de parameter pas in de methode een waarde krijgt toegekend dan wordt het out keyword gebruikt. Dit gebruik je dus voor out-parameter.

Je geeft parameters by reference door door het keyword ref voor de parameter in kwestie te zetten, zowel in de methode-declaratie als in de aanroep van de methode zelf.

Opgelet: Het dient opgemerkt te worden dat parameters by reference vaak tot problemen kan leiden indien je niet goed oplet, daar je rechtstreeks werkt met geheugenlocaties. Als je bijvoorbeeld verkeerdelijk een referentie optelt bij een value dan krijg je een nieuwe referentie die echter naar, mogelijk, een onbestaand stuk geheugen wijst. De ontwikkelaars van Visual Studio raden het gebruik van ref en out dan ook af, zeker indien je een beginnende programmeur bent. .

Out en ref

Via de keywords out en ref kunnen we parameters by reference doorgeven (ipv by value), het verschil daarbij is:

- `ref` : de parameter bevat reeds een waarde wanneer deze naar de methode wordt gestuurd
- `out` : de parameter bevat nog geen parameter en er wordt verwacht dat deze een waarde krijgt in de methode

Het verschil tussen het gebruik van `out` of `ref` keyword tonen we aan in het volgende voorbeeld.

Laten we dit aantonen met een voorbeeld. Stel dat we het vorige voorbeeld herschreven maar 'vergeten' om de parameter tweede een begin-waarde te geven:

```
static void Main(string[] args)
{
    int eerste = 5;
    int tweede;
    RefValueVerschil(eerste, ref tweede);

    Console.WriteLine("Eerste bedraagt na method:{0}", eerste);
    Console.WriteLine("Tweede bedraagt na method:{0}", tweede);
}
```

Dan krijgen we volgende, terechte, foutmelding:

1 Use of unassigned local variable 'tweede'

Door nu het out keyword te gebruiken geven we expliciet aan dat we beseffen dat de parameter in kwestie pas binnen de methode een waarde zal toegekend krijgen.

We zouden dus ons programma kunnen herschrijven met deze parameter. Hierbij moeten we ons ervan vergewissen dat we zeker de parameter getal2 een waarde toekennen in de methode!

```
static void RefValueVerschil(int getal1, out int getal2)
{
    getal2 = 10;
    getal1 = getal1 + 1;
    getal2 = getal2 + 2;
    Console.WriteLine("Getal1 bedraagt in method:{0}", getal1);
    Console.WriteLine("Getal2 bedraagt in method:{0}", getal2);
}

static void Main(string[] args)
{
    int eerste = 5;
    RefValueVerschil(eerste, out int tweede);

    Console.WriteLine("Eerste bedraagt na method:{0}", eerste);
    Console.WriteLine("Tweede bedraagt na method:{0}", tweede);
}
```

Dit geeft terug als output:

```
Getal1 bedraagt in method:6
Getal2 bedraagt in method:12
Eerste bedraagt na method:5
Tweede bedraagt na method:12
```

Volgende methode zal 2 parameters meekrijgen. De eerste wordt bij value gebruikt, de tweede by reference:

```
static void RefValueVerschil(int getal1, ref int getal2)
{
    getal1 = getal1 + 1;
    getal2 = getal2 + 2;
    Console.WriteLine("Getal1 bedraagt in methode:{0}", getal1);
    Console.WriteLine("Getal2 bedraagt in methode:{0}", getal2);
}

static void Main(string[] args)
{
    int eerste = 5;
    int tweede = 10;
    RefValueVerschil(eerste, ref tweede);
    Console.WriteLine("Eerste bedraagt na method:{0}", eerste);
    Console.WriteLine("Tweede bedraagt na method:{0}", tweede);
}
```

De uitvoer is de volgende, zoals verwacht:

```
Getal1 bedraagt in method:6
Getal2 bedraagt in method:12
Eerste bedraagt na method:5
Tweede bedraagt na method:12
```

Merk dus op dat enkel de variabele tweede aangepast wordt buiten de methode doordat we deze by reference doorgeven.

TODO

Named params

Default/Optional params

Method overloading

Geavanceerde methode concepten

Film Default

Maak een methode FilmRuntime() die 3 parameters accepteert:

1. Een string die de naam van de film bevat
2. Een integer die duur in minuten van de film bevat
3. Een enum-type die het genre van de film bevat (Drama, Actie, etc.) Indien de duur van de film niet wordt meegeven wordt een standaard lengte van 90 minuten ingesteld. Indien het genre niet wordt meegeven dan wordt deze default op Onbekend ingesteld.

De methode geeft niets terug maar toont eenvoudigweg de film op het scherm, gevolgd door z'n duur. Toon aan in je main dat de methode werkt met zowel 1, 2 als 3 parameters. Toon ook aan dat je met 'named arguments' de methode kan aanroepen.

Grote Som

Maak een methode Som() die eender welke hoeveelheid parameters van het type int aanvaardt en vervolgens de som van al deze parameters teruggeeft (als int).

Toon in je main aan dat de methode werkt door onder andere 1, 3, 5 en 10 gehele getallen mee te geven.

Toon ook aan dat je een array van 50 ints als parameter kan meegeven aan de methode. (hint:je moet het `params` keyword gebruiken).

Arrays

Arrays zijn een veelgebruikt principe in vele programmeertalen. Het grote voordeel van arrays is dat je een enkele variabele kunt hebben die een grote groep waarden voorstelt van eenzelfde type. Hierdoor wordt je code leesbaarder en eenvoudiger in onderhoud. Arrays zijn een zeer krachtig hulpmiddel, maar er zitten wel enkele venijnige addertjes onder het gras.

Een array is niet meer dan een verzameling waarden van hetzelfde type (bijvoorbeeld een verzameling ints, doubles of chars). Deze waarden kunnen benaderd worden via 1 enkele variabele, de array zelf. Door middel van een *index* kan ieder afzonderlijk element uit de array aangepast of uitgelezen worden.

Een nadeel van arrays is dat, eens we de lengte van een array hebben ingesteld, deze lengte niet meer kan veranderen. Later zullen we leren werken met lists en andere collections die dit nadeel niet meer hebben (zie [hier](#)).

Nut van arrays

Stel dat je de dagelijkse neerslag wenst te bewaren. Dit kan je zonder arrays eenvoudig:

```
int dag1=34;  
int dag2=45;  
int dag3=0;  
int dag4=34;  
int dag5=12;  
int dag6=0;  
int dag7=23;
```

Maar wat als je plots de neerslag van een heel jaar, 365 dagen, wenst te bewaren. Of een hele eeuw? Van zodra je een bepaalde soort data hebt die je veelvuldig wenst te bewaren dan zijn arrays de oplossing.

Bron

Grote delen van dit hoofdstuk zijn vertaald uit het handboek C# 4.0 Essentials.

Array Basics

Arrays declareren

Een array creëren (declareren) kan op verschillende manieren. Hoewel manier 1 de meest gebruikelijke is, zal deze voor de beginnende programmeur nog wat abstract lijken vanwege het gebruik van het new keyword. Manier 2 is de eenvoudigste en snelste manier, maar deze is wel minder flexibel.

Manier 1

De eenvoudigste variant is deze waarbij je een array variabele aanmaakt, maar deze nog niet initialiseert (i.e. je maakt enkel een identifier in aan). De syntax is als volgt:

```
type[] arraynaam;
```

Type kan dus eender welk type zijn dat je reeds kent. De [] (*square brackets*) duiden aan dat het om een array gaat.

Voorbeelden van array declaraties kunnen dus bijvoorbeeld zijn:

```
int[] verkoopCijfers;  
double[] gewichtHuisdieren;  
bool[] examenAntwoorden;
```

Stel dat je dus een array van strings wenst waarin je verschillende kleuren zal plaatsen dan schrijf je:

```
string[] myColors;
```

Vervolgens kunnen we later waarden toekennen aan de array, hiervoor gebruiken we het new statement.

```
string[] myColors;  
myColors = new string[] { "red", "green", "yellow", "orange", "blue" };
```

Je array zal vanaf dit punt een lengte 5 hebben en kan niet meer groeien.

Manier 2

Indien je direct waarden wilt toekennen (initialiseren) tijdens het aanmaken van de array zelf dan mag dit ook als volgt:

```
string[] myColors = { "red", "green", "yellow", "orange", "blue" };
```

Ook hier zal dus vanaf dit punt je array een vaste lengte van 5 elementen hebben. Merk op dat deze manier dus enkel werkt indien je reeds weet welke waarden in de array moeten. In manier 1 kunnen we perfect een array aanmaken en pas veel later in programma ook effectief waarden toekennen (bijvoorbeeld door ze stuk per stuk door een gebruiker te laten invoeren).

Manier 3

Nog een andere manier om arrays aan te maken is de volgende, waarbij je aangeeft hoe groot de array moet zijn, zonder reeds effectief waarden toe te kennen

```
string[] myColors;  
myColors = new string[5];
```

Samengevat

De 3 manieren om arrays te declareren zijn dus:

```
//Manier 1  
string[] myColors;  
myColors = new string[] { "red", "green", "yellow", "orange", "blue" };  
//Manier 2
```

```

string[] myColors = {"red", "green", "yellow", "orange", "blue" };
//Manier 3
string[] myColors;
myColors = new string[5];

```

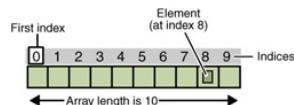
Elementen van een array aanpassen en uitlezen

Van zodra er waarden in een array staan of moeten bijgeplaatst worden dan kan je deze benaderen met de zogenaamde *array accessor* notatie. Deze notatie is heel eenvoudigweg de volgende:

```
myColors[i];
```

We plaatsen de naam van de array, gevolgd door brackets waarbinnen een getal i aangeeft het hoeveelste element we wensen te benaderen (lezen en/of schrijven).

De index van een C#-array start steeds bij 0. Indien je dus een array aanmaakt met lengte 10 dan heb je de indices 0 tot en met 9.



Veelgemaakte foute: Lengte en indexering van een arrays

Het gebeurt vaak dat beginnende programmeurs verward geraken omtrent het aanmaken van een array aan de hand van de lengte en het indexeren.

De regels zijn duidelijk:

- Bij het maken van een array is de lengte van een array gelijk aan het aantal elementen dat er in aanwezig is. Dus een array met 5 elementen heeft als lengte 5.
- Bij het schrijven en lezen van individuele elementen uit de array (zie hierna) gebruiken we een indexering die start bij 0. Bijgevolg is de index **4** van het laatste element in een array met **lengte 5**.

Schrijven

Ook schrijven van waarden naar de array gebruikt dezelfde notatie. Enkel moet je dus deze keer de array accessor-notatie links van de toekenningsoperator plaatsen. Stel dat we bijvoorbeeld de waarde van het eerste element uit de myColors array willen veranderen van red naar indigo, dan gebruiken we volgende notatie:

```
myColors[0] = "indigo";
```

Als we dus bij aanvang nog niet weten welke waarden de individuele elementen moeten hebben in een array, dan kunnen we deze eerst definiëren, en vervolgens individueel toekennen:

```

string[] myColors;
myColors = new string[5];
// ...
myColors[0]= "red";
myColors[1]="green";
myColors[2]= "yellow";
myColors[3]= "orange";
myColors[4]= "blue";

```

Uitlezen

Stel dat we een array aanmaken (eerste lijn) dan kunnen we dus bijvoorbeeld het getal 90 op het scherm tonen als volgt:

```

int[] scores = { 100, 90, 55, 0, 34 };
int kopie= scores[1];
Console.WriteLine(kopie);

```

of nog:

```
int[] scores = { 100, 90, 55, 0, 34 };
Console.WriteLine(scores[1]);
```

Stel dat we een array van getallen hebben, dan kunnen we dus bijvoorbeeld 2 waarden uit die array optellen en opslaan in een andere variabele als volgt:

```
int[] numbers = {5, 10, 30, 45};
int som = numbers[0] + numbers[1];
```

De variabele som zal dan vervolgens de waarde 15 bevatten (5+10).

Stel dat we alle elementen uit de array numbers met 5 willen verhogen, we kunnen dan schrijven:

```
int[] numbers = {5, 10, 30, 45};
numbers[0] += 5;
numbers[1] += 5;
numbers[2] += 5;
numbers[3] += 5;
```

Nog beter is het natuurlijk deze code (die quasi 4keer dezelfde statement bevat) te vereenvoudigen tot:

```
int[] numbers = {5, 10, 30, 45};
int teller = 0;
while (teller < 4)
{
    numbers[teller] += 5;
    teller++;
}
```

Of het equivalent met een for-loop

```
int[] numbers = {5, 10, 30, 45};
for(int teller=0; teller < 4; teller++)
{
    numbers[teller] += 5;
}
```

De lengte van de array te weten komen

Soms kan het nodig zijn dat je in een later stadium van je programma de lengte van je array nodig hebt. De `Length` eigenschap van iedere array geeft dit weer. Volgende voorbeeld toont dit:

```
string[] myColors = { "red", "green", "yellow", "orange", "blue" };
System.Console.WriteLine("Length of array = ", myColors.Length);
```

De variabele `myColors.Length` is een special element, van het type `int`, die iedere array met zich meedraagt (zie volgende semester). Je kan dus deze lengte ook toekennen aan een variabele:

```
int arrayLength = myColors.Length;
```

De `Length`-property wordt vaak gebruikt in for/while loops waarmee je de hele array wenst te doorlopen. Door de `Length`-property te gebruiken als grenscontrole verzekeren we er ons van dat we nooit buiten de grenzen van de array zullen lezen of schrijven:

```
//Alle elementen van een array tonen
for (int i = 0; i < getallen.Length; i++)
{
    Console.WriteLine(getallen[i]);
}
```

Volledig voorbeeldprogramma met arrays

Met al de voorgaande informatie is het nu mogelijk om heel eenvoudig complexere programma's te schrijven die veel data moeten kunnen verwerken. Meestal gebruikt men een for-element om een bepaalde operatie over de hele array toe te passen.

Het volgende programma zal een array van integers aanmaken die alle gehele getallen van 0 tot 99 bevat. Vervolgens zal ieder getal met 3 vermenigvuldigd worden. Finaal tonen we enkel die getallen die een veelvoud van 4 zijn na de bewerking.

```
//Array aanmaken
int[] getallen= new int[100];

//Array vullen
for (int i = 0; i < getallen.Length; i++)
{
    getallen[i] = i;
}

//Alle elementen met 3 vermenigvuldigen
for (int i = 0; i < getallen.Length; i++)
{
    getallen[i] = getallen[i]*3;
}

//Enkel veelvouden van 4 op het scherm tonen
for (int i = 0; i < getallen.Length; i++)
{
    if(getallen[i]%4==0)
        Console.WriteLine(getallen[i]);
}
```

Geheugengebruik bij arrays

Zie volgende filmpje op 31minuten.

Arrays kopieren

Arrays worden 'by reference' gebruikt in C#. Het gevolg hiervan is dat volgende code niet zal doen wat je wenst (`ploegen` , `nieuwePloegen` zijn twee arrays van een bijvoorbeeld `string[]`)

```
nieuwePloegen= ploegen;
```

Deze code zal perfect werken. Wat er echter is gebeurd is dat we de referentie naar `ploegen` ook in `nieuwePloegen` hebben geplaatst. Bijgevolg verwijzen beide variabelen naar dezelfde array, namelijk die waar `ploegen` al naar verwees. We hebben een soort alias gemaakt en kunnen nu op twee manieren de array benaderen. Als je dus schrijft:

```
nieuwePloegen[4]= "Beerschot";
```

Dan is dat hetzelfde als schrijven:

```
ploegen[4]:= "Beerschot";
```

En waar staan de ploegen in de `nieuwePloegen` array? *Die bestaat niet meer!*

Wil je dus arrays kopiëren dan kan dat niet op deze manier: **je moet manueel ieder element van de ene naar de andere array kopiëren** als volgt:

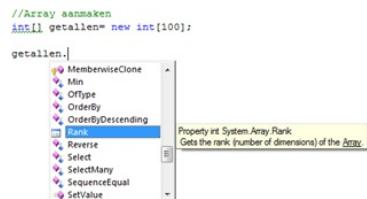
```
for(int i = 0; i<ploegen.Length; i++)
{
    nieuwePloegen[i]=ploegen[i];
}
```

Opgelet: wanneer je met arrays van objecten ([zie later](#)) werkt dan zal bovenstaande mogelijk niet het gewenste resultaten geven daar we nu de individuele referenties van een object kopiëren!

Nuttige array methoden

Net zoals we hebben gezien dat de Math-klasse een hele boel nuttige methoden in zich, zo ook heeft iedere array een aantal methoden waar handig gebruik van kan gemaakt worden.

Wanneer een array hebt gemaakt, dan kan je met de IntelliSense van Visual studio bekijken wat je allemaal kan doen met de array:



Al deze methoden hier beschrijven zal ons te ver nemen. De volgende methoden zijn echter zeer handig om te gebruiken:

Max(), Min(), Sum() en Average()

Volgende code geeft bijvoorbeeld het grootste getal terug uit een array genaamd "leeftijden":

```
int oudsteeftijd=leeftijden.Max();
```

System.Array

Alle C# arrays erven over van de System.Array klasse (klasse en overerving zien we later dit semester), hierdoor kan je zaken zoals Length gebruiken op je array. De System.Array klasse heeft echter ook nog een hoop andere nuttige methoden zoals de BinarySearch(), Sort() en Reverse() methoden. Het gebruik hiervan is steeds dezelfde zoals volgende voorbeelden tonen:

Arrays sorteren

Om arrays te sorteren roep je de Sort() methode op als volgt, als parameter geef je de array mee die gesorteerd moet worden.

Volgende voorbeeld toont hier het gebruik van:

```
string[] myColors = { "red", "green", "yellow", "orange", "blue" };
//Sorteer
System.Array.Sort(myColors);

//Toon resultaat van sorteren
for (int i = 0; i < myColors.Length; i++)
{
    System.Console.WriteLine(myColors[i]);
}
```

Wanneer je de Sort-methode toepast op een array van string dan zullen de arrays alfabetisch gerangschikt worden.

Arrays omkeren

Met de System.Array.Reverse() methode kunnen we dan weer de elementen van de array omkeren (dus het laatste element vooraan zetten en zo verder):

```
System.Array.Reverse(myColors);
```

Arrays leegmaken

Een array volledig leegmaken (alle elementen op 'null' zetten) doe je met de System.Array.Clear methode, als volgt:

```
System.Array.Clear(myColors);
```

Zoeken in arrays

De `BinarySearch`-methode maakt het mogelijk om te zoeken naar de index van een gegeven element in een array. *Deze methode werkt enkel indien de elementen in de array gesorteerd staan!* Je geeft aan de methode 2 parameters mee, enerzijds de array in kwestie en anderzijds het element dat we zoeken. Als resultaat wordt de index van het gevonden element teruggegeven. Indien niets wordt gevonden zal het resultaat -1 zijn.

```
System.Array.BinarySearch(myColors, "red");
```

Manueel zoeken in arrays

Het zoeken in arrays kan met behulp van while of for-loops tamelijk snel. Volgende programmaatje gaat zoeken of het getal 12 aanwezig is in de array. Indien ja dan wordt de index bewaard van de positie waar het getal staat:

```
int teZoekenGetal = 12;

int[] getallen = { 5, 10, 12, 25, 16 };

bool gevonden = false;
int index = -1;

for (int i = 0; i < getallen.Length; i++)
{
    if (getallen[i] == teZoekenGetal)
    {
        gevonden = true;
        index = i;
    }
}
```

Voorgaande stukje code is de meest naïeve oplossing. Bedenk echter wat er gebeurt indien het getal dat we zoeken 2 of meerdere keren in de array staat. Index zal dan de positie bevatten van de laatst gevonden 12 in de array.

Het is zéér belangrijk dat je vlot dit soort algoritmen kan schrijven, zoals:

- Zoeken van elementpositie in array
- Tellen hoe vaak een element in een array voorkomt
- Elementen in een array 1 of meerdere plaatsen omschuiven,

Methoden met arrays als parameter maken

Zoals alle types kan je ook arrays van eender welk type als parameter gebruiken bij het schrijven van een methode.

Opgelet:

Arrays worden altijd 'by reference' doorgegeven aan een methode. Dit heeft twee belangrijke gevolgen:

1. Je hoeft het ref keyword niet mee te geven, dit gebeurt impliciet reeds
2. Je werkt steeds met de eigenlijke array, ook in de methode. Als je dus aanpassingen aan de array aanbrengt in de methode, dan zal dit ook gevolgen hebben op de array van de parent-methode (logisch: het gaat om dezelfde array).

Stel dat je bijvoorbeeld een methode hebt die als parameter 1 array van ints meekrijgt. De methode zou er dan als volgt uitzien.

```
static void EenVoorbeeldMethode(int[] inArray)
{
}
```

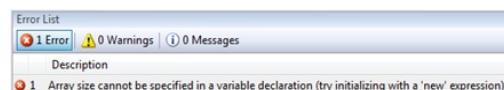
Array grootte in de methode

Een array als parameter meegeven kan dus, maar een ander aspect waar rekening mee gehouden moet worden is dat je niet kan ingeven in de parameterlijst hoe groot de array is! Je zal dus in je methode steeds de grootte van de array moeten uitlezen met de Length-eigenschap.

Volgende methode is dus **FOUT!**

```
static void EenVoorbeeldMethode(ref int[6] inArray)
{
}
```

En zal volgende error genereren:



Arraymethode voorbeeld

Volgende voorbeeld toont een methode die alle getallen van de array op het scherm zal tonen:

```
static void ToonArray(int[] getalArray)
{
    Console.WriteLine("Array output:");
    for (int i = 0; i < getalArray.Length; i++)
    {
        Console.WriteLine(getalArray[i]);
    }
}
```

Stel dat je elders volgende array hebt `int[] leeftijden= {2,5,1,6};`. De `ToonArray` methode aanroepen kan dan als volgt:

```
ToonArray(leeftijden);
```

En de output zal dan zijn:

```
Array output:
2
5
1
6
```

Voorbeeldprogramma met methoden

Volgende programma toont hoe we verschillende onderdelen van de code in methoden hebben geplaatst zodat:

1. de lezer van de code sneller kan zien wat het programma juist doet
2. zodat code herbruikbaar is

Analyseer de code en denk na hoe eenvoudig het is om een ander programma hiervan te maken (bijvoorbeeld vermenigvuldigen met 10 en alle veelvouden van 6 tonen: je hoeft enkel de parameters in de methode-aanroep aan te passen):

```
namespace ArrayMethods
{
    class Program
    {

        static void VulArray(int[] getalArray)
        {
            for (int i = 0; i < getalArray.Length; i++)
            {
                getalArray[i] = i;
            }
        }

        static void VermenigvuldigArray(int[] getalArray, int multiplier)
        {
            for (int i = 0; i < getalArray.Length; i++)
            {
                getalArray[i] = getalArray[i] * multiplier;
            }
        }

        static void ToonVeelvouden(int[] getalArray, int veelvoudenvan)
        {
            for (int i = 0; i < getalArray.Length; i++)
            {
                if (getalArray[i] % veelvoudenvan == 0)
                    Console.WriteLine(getalArray[i]);
            }
        }

        static void Main(string[] args)
        {
            //Array aanmaken
            int[] getallen = new int[100];

            //Array vullen
            VulArray(getallen);

            //Alle elementen met 3 vermenigvuldigen
            VermenigvuldigArray(getallen, 3);

            //Enkel veelvouden van 4 op het scherm tonen
            ToonVeelvouden(getallen, 4);
        }
    }
}
```

Array als return-type bij een methode

Ook methoden kun je natuurlijk een array als returntype laten geven. Hiervoor zet je gewoon het type array als returntype zonder grootte in de methode-signature.

Stel bijvoorbeeld dat je een methode hebt die een int-array maakt van een gegeven grootte waarbij ieder element van de array reeds een beginwaarde heeft die je ook als parameter meegeeft:

```
static int[] MaakArray(int lengte, int beginwaarde)
{
    int[] resultArray = new int[lengte];
    for (int i = 0; i < lengte; i++)
    {
        resultArray[i] = beginwaarde;
    }
    return resultArray;
}
```


Meer-dimensionale Arrays

Voorlopig hebben we enkel met 1-dimensionale array gewerkt. Je kan er echter ook meerdimensionale maken. Denk maar aan een n-bij-m array om een matrix voor te stellen.

Door een komma tussen rechte haakjes te plaatsen tijdens de declaratie kunnen we meer-dimensionale arrays maken.

Bijvoorbeeld 1D:

```
string[,] books;
```

3D:

```
short[,,,] temperatures;
```

(enz.)

Om een array ook onmiddellijk te initialiseren gebruiken we dan volgende uitdrukking:

```
string[,] books = {  
    {"Macbeth", "Shakespeare", "ID12341"},  
    {"Before I Get Old", "Dave Marsh", "ID234234"},  
    {"Security+", "Mike Pastore", "ID3422134"}  
};
```

Merk op dat we dus nu een 3 bij 3 array maken. Iedere rij bestaat uit 3 elementen.

OF bij een 3D:

```
int[,,,] temperatures= {  
    {  
        {3,4}, {5,4}  
    },  
    {  
        {12,34}, {35,24}  
    },  
    {  
        {-12,27}, {3,24}  
    },  
};
```

Stel dat we uit de books-array bijvoorbeeld de auteur van het derde boek wensen te tonen dan kunnen we schrijven:

```
Console.WriteLine(books[2, 1]);
```

Dit zal Mike Pastore op het scherm zetten.

En bij de temperaturen:

```
Console.WriteLine(temperatures[2,0,1]);
```

Zal 27 terug geven: we vragen van de laatste array ([2]), daarbinnenin de eerste array ([0]) en daarvan het tweede ([1])element.

Lengte van iedere dimensie in een n-dimensionale matrix

Indien je de lengte oproeft van een meer-dimensionale array dan krijg je de som van iedere lengte van iedere dimensie. Onze books array zal bijvoorbeeld dus lengte 9 hebben. Je kan echter de lengte van iedere aparte dimensie te weten komen met de GetLength() methode die iedere array heeft. Als parameter geef je de dimensie mee van de welke je de lengte wenst.

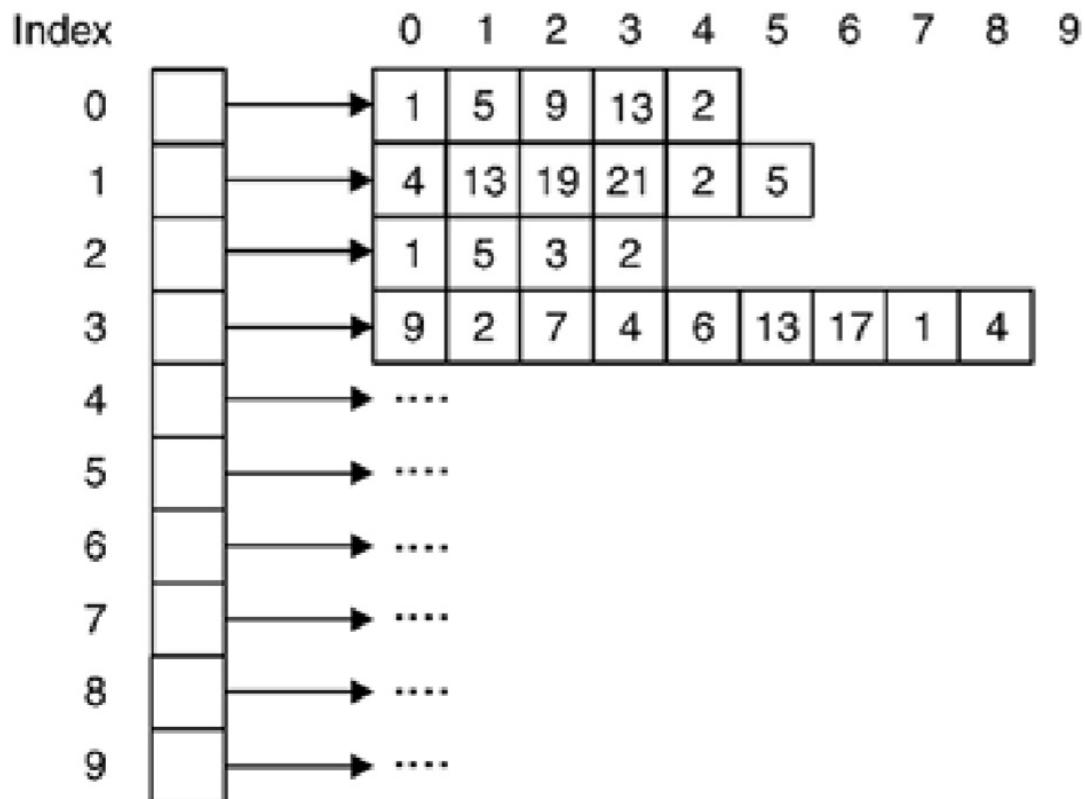
```
int arrayRijen = books.GetLength(0);  
int arrayKolommen = books.GetLength(1);
```

Het aantal dimensies van een array wordt trouwens weergegeven door de rank eigenschap die ook iedere array heeft. Bijvoorbeeld:

```
int arrayDimensions = myColors.Rank;
```

Jagged Arrays

Jagged arrays (letterlijk *gekartelde arrays*) zijn arrays van arrays maar van verschillende lengte. In tegenstelling tot de eerdere meer-dimensionale arrays moeten de interne arrays steeds dezelfde lengte hebben, bijvoorbeeld 3 bij 2 bij 4. Bij jagged arrays hoeft dat dus niet:



Jagged arrays aanmaken

Het grote verschil bij het aanmaken van bijvoorbeeld een 2D jagged arrays is het gebruik van de vierkante haken:

```
double[][] tickets;
```

(en dus niet ``tickets[.])

Vanaf nu kan je dan individuele arrays toewijzen aan ieder element van `tickets :

```
tickets={  
    new double[] {3.0, 40, 24},  
    new double[] {123, 31.3},  
    new double[] {2.1}  
};
```

Zoals je kan zien moeten de interne arrays dus niet dezelfde grootte hebben.

Indexering

De indexering blijft dezelfde, uiteraard moet je er wel rekening mee houden dat niet eender welke index binnen een bepaalde sub-array zal

A two-dimensional (jagged) Array

| | | | | | | | | | |
|----|------|----|----|-----|-----|-----|----|----|-----|
| 6 | 34 | 23 | -8 | 123 | 87 | 19 | 12 | | |
| 2 | 43 | 3 | 22 | -73 | 78 | 17 | 22 | 9 | 255 |
| 16 | 4 | 17 | 63 | 7 | -31 | -18 | 32 | 18 | 127 |
| 0 | -125 | 33 | 99 | 981 | 31 | 16 | | | |

Index [0] [5]

Index [1] [9]

Index [2] [7]

Index [3] [2]

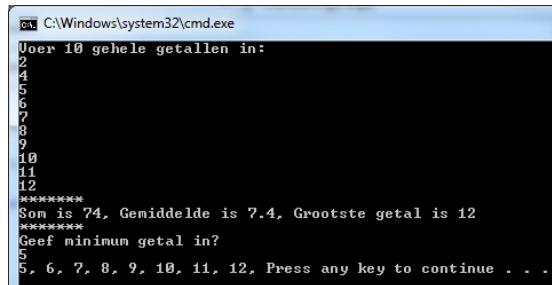
werken.

Basis van Arrays

ArrayOefener 1

Maak een programma dat aan de gebruiker vraagt om 10 waarden (int) in te voeren in een array. Vervolgens toont het programma De som, het gemiddelde en het grootste getal van deze 10.

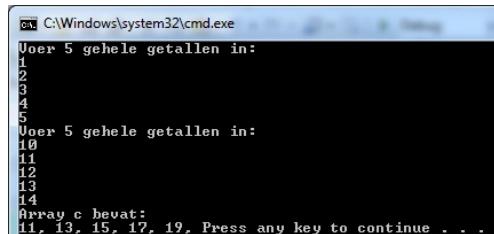
Vervolgens vraagt het programma de gebruiker om een getal in te voeren. Het programma toont dan alle getallen die groter of gelijk zijn aan dit ingevoerde getal zijn die in de array aanwezig zijn. Indien geen getallen groter zijn dan verschijnt een bericht "Niets is groter" op het scherm.



```
C:\Windows\system32\cmd.exe
Uoer 10 gehele getallen in:
2
4
5
6
7
8
9
10
11
12
*****
Som is 74, Gemiddelde is 7.4, Grootste getal is 12
*****
Geef minimum getal in?
5
5, 6, 7, 8, 9, 10, 11, 12, Press any key to continue . . .
```

ArrayOefener 2

Maak een programma dat aan de gebruiker vraagt om 2 keer 5 getallen in te voeren. Bewaar de eerste reeks waarden in een array A, de tweede reeks waarden in array B. Maak een nieuwe array C aan die steeds de som van het respectievelijke element uit arrays A en B. Toon het resultaat.



```
C:\Windows\system32\cmd.exe
Uoer 5 gehele getallen in:
1
2
3
4
5
Uoer 5 gehele getallen in:
10
11
12
13
14
Array c bevat:
11, 13, 15, 17, 19, Press any key to continue . . .
```

Vraag Array

Maak een array die 10 string kan bevatten. Ieder element van de array bevat een vraag (naar keuze te verzinnen) als string waar de gebruiker met een getal op moet antwoorden. Maak een array aan die tot 10 ints kan bevatten. Lees 1 voor 1 de vraag uit de array-string uit en toon deze op het scherm. Lees vervolgens het antwoord uit dat de gebruiker intypt en bewaar dit als int in de 2e array.

Na de 10 vragen toon je vervolgens de 10 vragen met achter iedere vraag het antwoord van de gebruiker.

Array Zoeker

Maak een programma dat eerst weer aan de gebruiker om 10 waarden vraagt die in een array worden gezet.

Vervolgens vraagt het programma welke waarde moet verwijderd worden. Wanneer de gebruiker hierop antwoord met een nieuwe waarde dan zal deze nieuw ingevoerde waarde in de array gezocht worden. Indien deze gevonden wordt dan wordt deze waarde uit de array verwijderd en worden alle waarden die erachter komen met een plaatsje naar links opgeschoven, zodat achteraan de array terug een lege plek komt.

Deze laatste plek krijgt de waarde -1.

Toon vervolgens alle waarden van de array.

Indien de te zoeken waarde meer dan 1 keer voorkomt, wordt enkel de eerst gevonden waarde verwijderd.

```

C:\Windows\system32\cmd.exe
Voor 10 gehele getallen in:
2
3
4
5
8
9
10
12
15
16
*****
welke getal moet verwijderd worden
5
Resultaat is:2, 3, 4, 8, 9, 10, 12, 15, 16, -1,
Press any key to continue . . .

```

LeveringsBedrijf

Maak een programma voor een koerierbedrijf. Maak een array die 10 postcodes bevat (zelf te kiezen) van gemeenten waar het bedrijf naar levert. Maakt een tweede array die de prijs bevat per kg van iedere respectievelijke gemeente. Het eerste element van deze array bevat dus de prijs/kg om naar de gemeente die als eerste in de array met postcodes staat.

Vraag aan de gebruiker een postcode en het gewicht van het pakket. Vervolgens wordt de prijs opgezocht voor die gemeente en wordt deze berekend gegeven het ingegeven gewicht.

Indien het bedrijf niet levert aan de ingetypte postcode dan wordt een foutmelding weergegeven.

Methoden met arrays als parameter

Parkeergarage

Een parkinggarage vraagt sowieso €2.00 om tot maximum 3uur te parkeren. Per extra uur NA die 3uur wordt telkens €0.5 aangerekend (dus 4uur parkeren kost €2.5. Er wordt maximum €10 aangerekend per dag. En veronderstel dat er nooit langer dan 1 dag (24u) kan geparkeerd worden.

Schrijf een programma dat het verschuldigde bedrag toont gegeven de duur van een auto. Bij het opstarten van het programma wordt eerst gevraagd hoeveel auto's ingevoerd zullen worden, dan wordt per auto de duur van het parkeren gevraagd. Finaal wordt, netjes getabuleerd, alle informatie getoond, inclusief het totaal bedrag. Gebruik minstens 1 methode 'berekenKosten' die de kost voor 1 gebruiker telkens teruggeeft, gegeven de duur als parameter. Gebruik ook een methode die een array als parameter aanvaardt (bijvoorbeeld de array met daarin de respectievelijke uren per auto).

Voorbeeldoutput: Opstart:

```

C:\Windows\system32\cmd.exe
Geef aantal auto's in
3
Geef parkeertijd auto 1 in <uren>:
1.5
Geef parkeertijd auto 2 in <uren>:
4
Geef parkeertijd auto 3 in <uren>:
24

```

Resultaat:

```

C:\Windows\system32\cmd.exe
Auto    Duur      Kost
1       1.5       2.00
2       4.0        2.50
3      24.0       10.00
Totaal  29.5      14.50
Druk op een toets om door te gaan. . .

```

Voetbalcoach

Maak een console-applicatie voor een assistent voetbaltrainer (of een sport naar keuze)

De voetbalcoach wil na de match iedere knappe en domme actie van een speler weten. Op die manier weet hij aan het einde van de match wie er de meeste goede en slechte acties doet. De spelers hebben rugnummers 1 tot en met 12.

Wanneer de coach een rugnummer in typt kan hij vervolgens ingeven of hij (a) een knappe actie of (b) een domme actie wil ingeven. Vervolgens geeft hij een getal in. Gebruik een 2dimensionale array die per speler het aantal domme en goede acties bijhoudt (de array is dus 12 bij 2 groot: 1 lijn per speler, bestaande uit 2 kolommen voor goede en domme actie. De index van de lijn is de rugnummer van speler -1).

Een typische invoer kan dus zijn:

```
2  
a  
6
```

De coach kiest dus speler met rugnummer 2, hij kiest voor een knappe actie, en voer 6 in als aantal goede acties.

In de array op index 1 (rugnummer -1) zal in de de 0'de kolom(0=goede, 1=slechte) het getal 6 geplaatst worden.

Vervolgens kan de coach een ander rugnummer (of hetzelfde) invoeren en zo verder.

Wanneer de coach 99 invoert stopt het programma en worden de finale statistieken getoond: per speler/rugnummer wordt het aantal goede en domme acties getoond, met daarnaast het verschil tussen beide:

(gebruik \t om goede tabs te zetten tussen de data)

| Rugnummer | Goede | Domme | Verschil |
|-----------|-------|-------|----------|
| 1 | 5 | 2 | 3 |
| 2 | 6 | 7 | -1 |

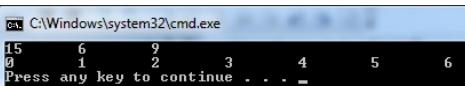
Het programma toont ook welke speler(s) het meest performant waren, namelijk zij met het grootste positieve verschil tussen goede en domme acties, alsook de minst performante en de meeste "gemiddelde" speler (i.e. verschil == 0)

Array Viewer

Maak een programma dat een methode VisualiseerArray implementeert. De methode zal een array (type int) als parameter hebben en niets teruggeven (void). Echter, de methode zal met behulp van Write() de array, van elkeer welke grootte, op het scherm tonen. Tussen ieder element van dezelfde rij dient een tab ('\t') gezet te worden. Je dient in de methode gebruik te maken van een for-loop. Voorbeeld van main:

```
int[] array={15,6,9};  
int[] array2={0,1,2,3,4,5,6};  
VisualiseerArray(array);  
VisualiseerArray(array2);
```

Geeft volgende output:



```
15   6   9  
0   1   2   3   4   5   6  
Press any key to continue . . .
```

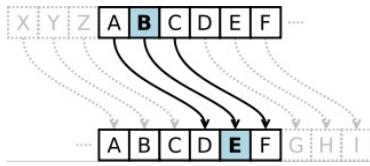
Caesar-encryptie

Maak 2 methoden `Encrypt` en `Decrypt` die als parameters telkens een array krijgen en een integer. Bedoeling is dat de Encrypt-methode de array van het type string versleuteld gegeven de sleutel x volgens het Caesar cipher (zie hieronder). Als resultaat komt er uit de methode de geëncrypteerde array. De decrypt-methode doet hetzelfde maar omgekeerd: je kan er een versleutelde tekst insteken en de sleutel en de ontcijferde tekst komt terug (merk op dat je decrypt-methode gebruik kan maken van de encrypt-methode!).

Toon in je main aan dat je methoden werken (door bijvoorbeeld aan de gebruiker een stuk tekst te vragen en een sleutel en deze dan te encrypteren/de-crypteren).

Encryptie is de kunst van het versleutelen van data. Hierbij gaat men een gewone tekst zodanig omvormen (*versleutelen*) zodat deze onleesbaar is en enkel kan ontcijferd worden door de ontvanger die weet hoe de tekst terug kan verkregen worden en enkel indien deze ook de 'private' sleutel heeft.

Een klassiek encryptie-algoritme uit de oudheid is de Caesar substitutie. Hierbij gaan we het alfabet met x plaatsen opschuiven en vervolgens de te versleutelen tekst letter per letter te vervangen met z'n respectievelijke opgeschoven versie. Hierbij is x dus de geheime sleutel die zender en ontvanger moeten afspreken.



Stel bijvoorbeeld dat $x=3$ dan krijgen we volgende nieuwe alfabet: DEFGHIJKLMNOPQRSTUVWXYZABC Waarbij dus de A zal vervangen worden door een D, de Z door een C, etc.

Willen we deze tekst dus 'encrypteren': the quick brown fox jumps over the lazy dog

dan krijgen we: WKH TXLFN EURZQ IRA MXPSV RYHU WKH ODCB GRJ

Meer-dimensionale arrays

Determinant

Schrijf een programma dat een methode BerekenDeterminant heeft. Deze methode heeft één parameter als input: een 2 bij 2 array van integers. Als resultaat geeft de methode de determinant als integer terug. Zoek zelf op hoe je de determinant van een matrix kunt berekenen.

Volgende voorbeeld-main dient te werken,

```
int[,] aMatrix={  
    {2,4},  
    {3,5}  
};  
Console.WriteLine($"Determinant van matrix is {BerekenDeterminant(aMatrix)}");
```

geeft als output:

```
Determinant van matrix is -2
```

Extra: Breidt uit zodat de BerekenDeterminant-methode ook werkt voor 3-bij-3 matrices. De methodeaanroep blijft dezelfde, enkel de interne code van de methode zal nu rekening moeten houden met de grootte van de matrix .

2D Array Viewer

Breidt het ArrayViewer programma uit zodat ook 2-dimensionale arrays gevisualiseerd kunnen worden. (Hint: gebruik de GetLength() methode van een array).

Voorbeeld van main:

```
int [,] array= { {15,6,9},{1,2,3},{6,9,12}};  
VisualiseerArray(array);
```

Output:

```
15 6 9  
1 2 3  
6 9 12
```

MatrixMultiplier

Schrijf een methode VermenigvuldigMatrix die 2 matrices als invoer verwacht en als resultaat een nieuwe matrix teruggeeft die het product van beide matrices bevat.

Pro

Galgje

Maak een spel , vergelijkbaar als galgje, waarin de speler een woord moet raden. Zie [Wiki](#) voor de spelregels indien je deze niet kent.

Voorbeeld output:



```
*****  
Geef letter in, of typ het volledige woord indien je het denkt te weten:  
a*****  
Geef letter in, of typ het volledige woord indien je het denkt te weten:  
s*****  
Geef letter in, of typ het volledige woord indien je het denkt te weten:  
i*****  
ar****is  
Geef letter in, of typ het volledige woord indien je het denkt te weten:  
artesis  
Correct! U hebt het juiste woord gevonden.  
Benodigde pogingen:4  
Druk op een toets om door te gaan. . . -
```

All In One

Volgende hoofdstukken bevatten grotere projecten waarin wordt getracht meerdere technieken uit de vorige hoofdstukken te combineren.
Het doel van dit hoofdstuk is tweeledig:

1. Op een andere manier tonen hoe specifieke C# elementen in een 'realistische' programma's kunnen gebruikt worden
2. Aantonen dat, indien je tot hier bent geraakt, je al een aardig hoop skills hebt om grote, complexe applicaties te maken die verder gaan dan de standaard oefeningen die je na ieder hoofdstuk hebt gemaakt.

Volgt nu een beschrijving van de belangrijkste technieken die je in de projecten hierna zal tegenkomen:

- [Console Matrix](#): Console, ConsoleColor, Loops, Random
- [Ascii filmpjes maken met loops](#): Loops, Beslissingen, Chars, Casting
- [Ascii filmpjes maken met methoden](#): Vorige project + Methoden, Chars, Casting
- [Tekst-gebaseerd Maze game](#): Loops, Methoden, Arrays, `out`

Matrix Console

In de vorige eeuw was The Matrix een uiterst memorabele film. In volgende code tonen we hoe je het "bekende" computer-beeld kunnen nadoen waarin groene, random letters op het scherm verschijnen.

```
Random rangen = new Random();
Console.ForegroundColor = ConsoleColor.Green;
while (true)
{
    //Genereer nieuw random teken:
    char teken = Convert.ToChar(rangen.Next(62, 400));
    //Zet teken op scherm
    Console.Write(teken);

    //Ietwat vertragen
    System.Threading.Thread.Sleep(1);

    //Af en toe donker kleurtje
    if(rangen.Next(0,3)==0)
    {
        Console.ForegroundColor = ConsoleColor.DarkGreen;
    }
    else
    {
        Console.ForegroundColor = ConsoleColor.Green;
    }
}
```

Enkele opmerkingen:

- `System.Threading.Sleep()` is een ingebouwde C# methode die aan de computer verteld dat je applicatie(thread) gedurende x milliseconden mag gepauzeerd (Sleep) worden. Het argument geeft weer hoeveel milliseconden dit moet zijn. Wil je dus 1 seconden pauzeren dan geef je 1000 mee. **Opgelat**: Sleep zal je programma volledig "blokkeren", het zal met andere woorden ook geen andere zaken doen zoals input van de gebruiker detecteren.

Console-filmpjes maken

Volgende tutorial toont hoe je een eenvoudig filmpje kan maken dat je, mits wat fantasie, vlot kan uitbreiden over enkele weken met interactieve aspecten.

Basisloop voor het filmpje

Volgende voorbeeld toont wat je bijvoorbeeld kan doen. Kopieer dit alles naar een eigen project tussen de accolades van de main:

```
int framenummer = 0;
int sleeptime = 1000; // in milliseconden
while (true)
{
    framenummer = framenummer + 1;
    if (framenummer == 1)
    {
        Console.WriteLine("Het begin");
    }
    else if (framenummer == 2)
    {
        Console.WriteLine("Tweede frame");
    }
    else if (framenummer == 3)
    {
        Console.ForegroundColor = ConsoleColor.Green;
        Console.WriteLine("Derde frame Andere kleur");
    }
    else if (framenummer == 4)
    {
        sleeptime = 4000; //vergeet niet terug te zetten in latere frames indien je dit maar eenmalig wil
        Console.WriteLine("Frame dat langer blijft staan");
    }
    else if (framenummer == 5)
    {
        sleeptime = 500;
        Console.WriteLine("Moving ball: *");
    }
    else if (framenummer == 6)
    {
        Console.WriteLine("Moving ball: *");
    }
    else if (framenummer == 7)
    {
        Console.WriteLine("Moving ball: *");
    }
    else if (framenummer == 8)
    {
        Console.WriteLine("Moving ball: *");
    }

    //Voeg hier frames tussen

    else
    {
        Console.ForegroundColor = ConsoleColor.Yellow;
        Console.WriteLine("\n\n\nThe End");
    }
    System.Threading.Thread.Sleep(sleeptime); //Pauzeer programma
    Console.Clear();
}
```

Deze code zal een teller (framenummer) per seconde (instelbaar via sleeptime) met 1 verhogen. Vervolgens zal de code uitgevoerd worden binnen de else-if clause die overeenkomt met de huidige framenummer. De eerste keer staat de teller op 1 en wordt dus de code tussen volgende if uitgevoerd.

```
if (framenummer == 1)
{
    Console.WriteLine("Het begin");
}
```

Vervolgens wordt deze met 1 verhoogd en wordt deze code uitgevoerd:

```

else if (framenummer == 2)
{
    Console.WriteLine("Tweede frame");
}

```

En zo voort. Je kan dus zelf frames toevoegen door steeds een constructie als de volgende toevoegen waar de commentaar `//Code om uit te voeren in dit frame staat:`

```

else if (framenummer == X)
{
    //Code om uit te voeren in dit frame
}

```

(vervang X door het framenummer dat dit moet zijn)

Inspiratie nodig

Mogelijke uitbreidingen kunnen zijn:

- Kleuren aanpassen om zo mooiere zaken te tonen (zie voorbeeldframe 3)
- Timing aanpassen wanneer volgende frame moet getoond worden (zie voorbeeldframe 4)
 - Bijgevolg kan je ook animaties doen, verlaag gewoon de tijd tussen frame en zet zaken op andere plekken (zie voorbeeldframes 5 t.e.m. 8)
- Ipv tekst zou je heeldere ASCII-art afbeeldingen kunnen tonen (én animeren)

Geavanceerde uitbreidingen

Niet-sequentiële flow

Je kan ook bepalen wat het volgende frame moet zijn door de variabele framenummer aan te passen naar het framenummer dat je nodig hebt -1 . Stel dat je bijvoorbeeld in frame 8 wenst dat na dit frame frame 3 wordt uitgevoerd, dan schrijf je:

```

else if (framenummer == 8)
{
    framenummer = 2;
    Console.WriteLine("Moving ball:      *");
}

```

Cursor verzetten

Via de methode `ConsoleCursorPosition` kan je instellen waar de cursor moet gezet worden. Je geeft tussen de haakjes van deze methode de x,y coördinaten (integers) mee waar de cursor moet gezet worden. Als je vervolgens tekst schrijft dan wordt die weggeschreven vanaf dat punt. De coördinaten zijn x,y coördinaten, waarbij het punt (1,1) het eerste karakter linksboven in de console is. Volgende frame zet bijvoorbeeld een "X" 10 letters naar rechts, 20 lijnen naar beneden

```

else if (framenummer == 9)
{
    Console.SetCursorPosition(10, 20);
    Console.WriteLine("X");
}

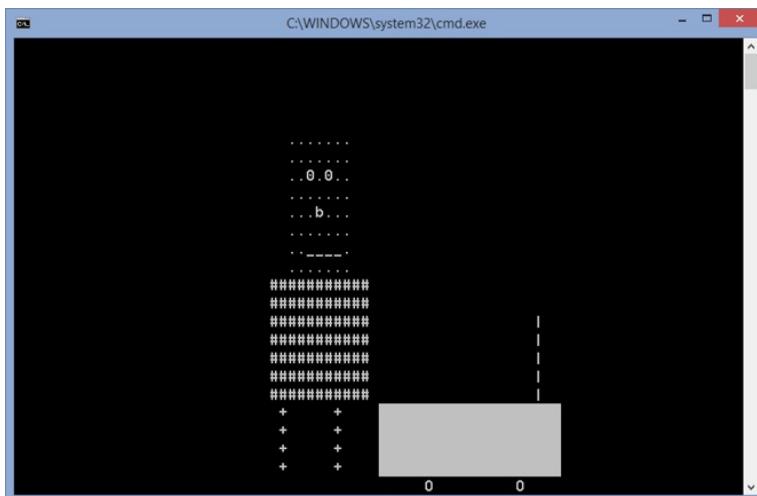
```

Methoden gebruiken om een ascii-filmpje te maken

Volgende demonstratie toont de kracht van methoden. We zullen een steeds complexer geheel maken, dat dankzij methoden, nog steeds onderhoudbaar én leesbaar zal blijven. We zullen een bottom-up approach hanteren waarbij we eerst beginnen met de meest basis-functionaliteit die we nodig hebben en zo steeds een schil, in de vorm van een methode, er omheen coderen.

Ons doel is een method `SpeelFilm` te maken die een filmpje, bestaande uit opeen volgende frames en scenes, zal afspelen in de Console.

We zullen een uiterst boeiend filmpje maken waarin een mannetje naar z'n auto wandelt en er vervolgens in wegrijdt.



Karakter op scherm tekenen

We maken een methode die 1 karakter op het scherm kan plaatsen op een positie naar keuze. Omdat we willen voorkomen dat dit mislukt indien de coördinaten buiten het scherm vallen, zullen we in deze methode eerst controleren of de coördinaten geldig zijn.

```
static void DrawChar(char drawchar, int posX, int posY)
{
    if (posX >= 0 && posX < Console.WindowWidth)
    {
        if (posY >= 0 && posY < Console.WindowHeight)
        {
            Console.SetCursorPosition(posX, posY);
            Console.Write(drawchar);
        }
    }
}
```

Willen we deze methode gebruiken dan kunnen we bijvoorbeeld in de `Main` schrijven: `DrawChar('@',5,10);` Dit zal resulteren in het "@"-teken op het scherm op de positie 5, 10 (plaatsen naar rechts, 10 lijnen naar beneden).

Rechthoek tekenen

Een rechthoek tekenen kan nu heel eenvoudig met behulp van 2 for-lussen. Eén lus om van links naar rechts te gaan. Eén lus om van boven naar onder te gaan.

De methode aanvaardt naast het karakter dat we willen tekenen ook nog :

1. De positie van de linkerbovenhoek van rechthoek op het scherm
2. De lengte en de breedte van de rechthoek

```
static void DrawRectangle(char drawchar, int posX, int PosY, int width, int height)
{
    for (int i = posX; i < posX + width; i++)
    {
        for (int j = PosY; j < PosY + height; j++)
        {
```

```
        DrawChar(drawchar, i, j);  
    }  
}  
}
```

We kunnen dus deze methode aanroepen als volgt: `DrawRectangle('*', 4, 6, 3, 6);`

Gezicht tekenen

Het hek is van de dam. We kunnen nu allerlei complexere zaken tekenen bestaande uit combinaties van rechthoeken en karakters. Een gezicht kan bijvoorbeeld bestaan uit 1 grote rechthoek voor het gezicht. Met daarin 2 aparte ogen, voorgesteld door het karakter '0' (dus mbv DrawChar) en een neus 'b'. Alsook een mond die bestaat uit een korte rechthoek bestaande uit underscores:

```
static void DrawFace(int posX, int posY)
{
    DrawRectangle('. ', posX, posY, 7, 8);
    DrawChar('0', posX + 2, posY + 2);
    DrawChar('0', posX + 4, posY + 2);
    DrawChar('b', posX + 3, posY + 4);
    DrawRectangle('_', posX + 2, posY + 6, 4, 1);
}
```

De enige extra complexiteit is dat we steeds rekening moeten houden met de locatie van het gezicht (`posx` , `posy`) en de onderlinge posities van de gezichts-onderdelen. We plaatsen dus bijvoorbeeld de mons op 6e lijn ten opzichte van de bovenkant van het gezicht. Denk er ook aan dat we de volgorde van methode-aanroepen hier relevant is. Indien we de eerste `DrawRectangle` aanroep helemaal onderaan zouden plaatsen, dan zou deze al de andere gezichts-onderdelen overtekenen.

Auto tekenen

```
static void DrawCar(int posX, int posY)
{
    DrawRectangle('■', posX, posY + 5, 20, 4);
    DrawRectangle('|', posX + 17, posY, 1, 5);
    DrawChar('O', posX + 5, posY + 9);
    DrawChar('O', posX + 15, posY + 9);
}
```

Mannetje tekenen

Een mannetje tekenen bestaat dan uit een gezicht een liif (rechthoek) en 2 benen (rechthoek). Z'n armen is hij kwitaarakt:

```
static void DrawMan(int posX, int posY)
{
    DrawFace(posX + 4, posY);
    DrawRectangle('+', posX + 3, posY + 14, 1, 5);
    DrawRectangle('+', posX + 9, posY + 14, 1, 5);
    DrawRectangle('#', posX + 2, posY + 8, 11, 7);
}
```

Of een mannetje dat spring?

```
static void DrawJumpingMan(int posX, int posY)
{
    DrawFace(posX + 4, posY);
    DrawRectangle('+', posX + 3, posY + 14, 1, 5);
    DrawRectangle('+', posX + 9, posY + 15, 5, 1);
    DrawRectangle('#', posX + 2, posY + 8, 11, 7);
}
```

Volgende code zal een animatie afspeelt van een mannetje dat springt en staat afwisselend:

```
int teller = 0;  
while (true)  
{
```

```

    if(teller%2==0)
        DrawMan(4,4);
    else
    {
        DrawJumpingMan(4,4);
    }
    teller++;
}

```

Man in auto

We kunnen weer een niveau hoger gaan: we combineren onze `DrawMan` en `DrawCar` methode om zo een mannetje in een auto te krijgen:

```

static void DrawManInCar(int posX, int posY)
{
    DrawMan(posX + 2, posY);
    DrawCar(posX, posY + 10);
}

```

Film maken

We kunnen nu een über-boeiend filmpje maken waarin het mannetje naar de auto loopt en dan er in wegrijdt. We zullen dit stukje top-down benaderen. Eerst maken we een methode `SpeelFilm()` die frame per frame het filmpje zal afspelen. Afhankelijk van het framenummer zal een andere scene getoond worden (lijnen 6,8):

```

private static void SpeelFilm()
{
    for (int i = 0; i < 60; i++)
    {
        if (i < 35)
            ManNaarAutoScene(i);
        else if (i >= 35)
            ManRijdWegScene(i - 35);

        System.Threading.Thread.Sleep(100);
        Console.Clear();
    }
}

```

Merk op dat we `i` gebruiken als framenummer en zo weten wanneer welke scene moet afgespeeld worden. Voorts geven we het framenummer door naar de scene-methoden voor het geval ze deze nodig hebben om bijvoorbeeld de correcte positie te bepalen:

```

private static void ManRijdWegScene(int framenummer)
{
    DrawManInCar(40+framenummer*2,5);
}

private static void ManNaarAutoScene(int framenummer)
{
    if (framenummer % 2 == 0)
    {
        DrawJumpingMan(1 + framenummer, 5);
    }
    else
    {
        DrawMan(1 + framenummer, 5);
    }

    DrawCar(40, 15);
}

```

Vreemde tekens in console tonen

Niets is zo leuk als de vreemdste tekens op het scherm tonen. In oude console-games werden deze tekens vaak gebruikt om *complex* tekeningen op het scherm te tonen: Om je filmpjes nog cooler te maken leggen we daarom uit hoe je dit kan tewerkstelligen, gebruikmakende van je kennis over converteren.



Unicode karakters tonen

Zonder een uitleg te geven over het verschil tussen ASCII en Unicode is het vooral belangrijk te weten dat je best met Unicode werkt.

1. Zoek het teken(s) dat je nodig hebt in een Unicode-tabel ([Deze is handig](#))
2. Plaats bovenaan je Main: `Console.OutputEncoding = System.Text.Encoding.UTF8;`
3. Je kan nu op 2 manieren dit teken in console plaatsen

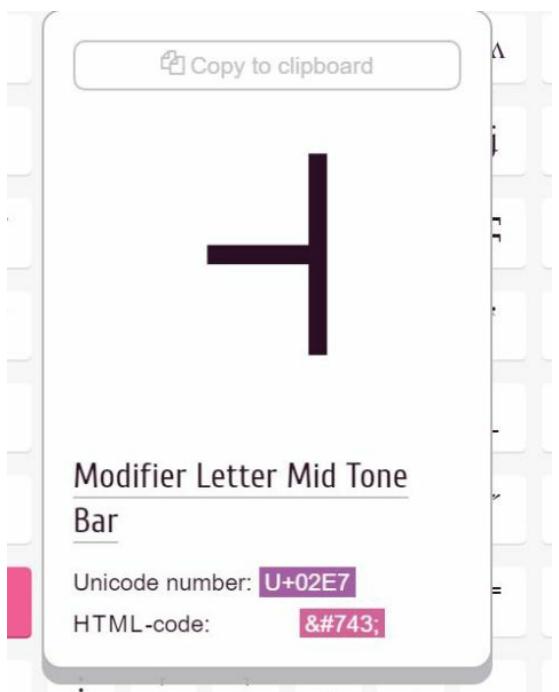
Manier 1: copy/paste

Kopieer het karakter zelf en plaats het in je code waar je het nodig hebt, bijvoorbeeld:

```
Console.WriteLine("ሴ");
```

Manier 2: hexadecimale code casten naar char

Noteer de hexadecimale code van het karakter dat in de tabel staat.



In dit geval is de code 0x02e7.

Om dit teken te tonen schrijf je dan:

```
char blokje = (char)0x02e7;  
Console.WriteLine(blokje);
```

In C# schrijf je hexadecimale getallen als volgt als je ze rechstreeks in een string wilt plaatsen: \u02e7

Wil je dus bovenstaande teken schrijven dan kan dan ook als volgt:

```
Console.WriteLine("\u02e7");
```

The Array MazeGame



Inleiding

In dit all-in-one tonen we hoe je, stap voor stap, kan komen tot een speelbaar, eenvoudige tekst-gebaseerd spel. We hanteren hierbij de principes van "refactoring": we gaan onze code steeds verbeteren op gebied van leesbaarheid en onderhoudbaarheid. Bij iedere stap zullen we dan ook extra functionaliteit toevoegen.

Het doel is te komen tot een spel waarbij de gebruiker kan wandelen door een kaart. De kaart zelf is dynamisch, bepaalde ruimtes zijn pas toegankelijk wanneer aan bepaalde voorwaarden is voldaan.

A screenshot of a Windows command prompt window titled 'C:\Windows\system32\cmd.exe'. Inside the window, there is a 2D array representation of a map labeled 'MAP' at the top left. The map consists of a grid of zeros ('0') and a single red 'X' mark. To the right of the map, the text 'In de securityroom' is displayed. Below the map, there is a multi-line message:

De veiligheidsagent houdt je nauwlettend in het oog. Typ "G" om een geheime ruimte te ontdekken
NOZW? Naar waar wil je?
-

Vereiste kennis

Deze tutorial gaat er van uit dat je volgende zaken beheerst:

- Basisprincipes van Arrays, zowel 1D als 2D arrays: aanmaken, waarden toevoegen/uitlezen
- Werken met de Console-bibliotheek: in het bijzonder Clear(), SetCursorPosition(), ForegroundColor en BackGroundColor, Write() vs WriteLine()
- Je kan werken met while en for-loops
- Je begrijpt de werking van het `out` keyword

Fase 1: Een saai spel

We gebruiken een array van strings om de opeenvolgende kamers te beschrijven. Door middel van een for-loop doorlopen we de array en tonen we iedere beschrijving van de kamer op het scherm.

Telkens de gebruiker op enter drukt verschijnt de volgende kamer.

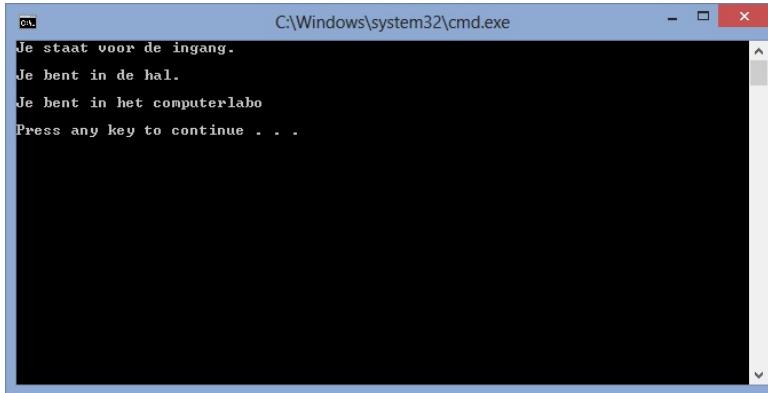
Merk op dat de array-lengte geen invloed heeft op de forloop. We kunnen dus eenvoudig kamers toevoegen zonder dat dit invloed heeft op de werking van het programma. We zullen blijven behouden doorheen het hele programma (de speciale kaart uitgezonderd in fase 8).

```

string[] Kamers = new string[]
{
    "Je staat voor de ingang.",
    "Je bent in de hal.",
    "Je bent in het computerlabo",
};

for (int i = 0; i < Kamers.Length; i++)
{
    Console.WriteLine(Kamers[i]);
    Console.ReadLine();
}

```



Fase 2: Een interactief saai spel

We bieden de mogelijkheid aan aan de gebruiker om zelf te kiezen naar welke kamer er wordt gegaan. De gebruiker kan dus "vooruit" of "achteruit" gaan in de array. We houden hiervoor een variabele (`huidigekamer`) bij die bijhoudt waar de gebruiker zich momenteel bevindt.

Telkens de gebruiker zich wil verplaatsen controleren we of deze verplaatsen toegestaan is. De `Huidigekamer` variabele is dus automatisch ook de index van de te tonen kamer in de string-array.

```

string[] Kamers = new string[]
{
    "Je staat voor de ingang.",
    "Je bent in de hal.",
    "Je bent in het computerlabo",
};

int huidigekamer = 0;
string keuze = "";
while (keuze != "q")
{
    Console.WriteLine(Kamers[huidigekamer]);

    Console.WriteLine("Vooruit= V, ACHTERUIT = A, q= quit");
    keuze = Console.ReadLine();
    if (keuze == "V" && huidigekamer != Kamers.Length - 1)
        huidigekamer++;
    else if (keuze == "A" && huidigekamer != 0)
        huidigekamer--;
    else if (keuze == "q")
        Console.WriteLine("Byebye");
    else
    {
        Console.WriteLine("Foute invoer");
    }
}

```

```

C:\Windows\system32\cmd.exe
Je staat voor de ingang.
Vooruit= V, ACHTERUIT = A, q= quit
V
Je bent in de hal.
Vooruit= V, ACHTERUIT = A, q= quit
V
Je bent in het computerlabo
Vooruit= V, ACHTERUIT = A, q= quit
A
Je bent in de hal.
Vooruit= V, ACHTERUIT = A, q= quit
A
Je staat voor de ingang.
Vooruit= V, ACHTERUIT = A, q= quit
V
Je bent in de hal.
Vooruit= V, ACHTERUIT = A, q= quit
-
```

Fase 3: Een 2D-wereld met lookup-table

Stap 1: Kaart maken

Vervolgens willen we de mogelijkheid om een 2D wereld aan te bieden. Hierbij gebruiken we een zogenaamde **lookup-table** zodat we onze wereld array eenvoudig kunnen houden én kamers kunnen herbruiken.

Eerste definiëren we de verschillende kamers die er bestaan:

```

string[] Kamers =
{
    "Onbekend terrein", //0
    "In een gang", //1
    "In de lobby", //2
    "In de bar", //3
    "In de keuken", //4
    "Achtertuin"//5
};
```

Vervolgens maken we 2D-array die onze kaart voorstelt. De array is van het type `int`. Iedere cijfer in de array zal de index bevatten van de kamer die op die plek moet komen. Dit is dus een zogenaamde look-up-table of Lut (meer info: [wiki](#)):

```

int[,] Kaart =
{
    {1, 2, 1, 3},
    {0, 1, 0, 1},
    {0, 4, 0, 5}
};
```

Linksboven beginnen we dus met een Gang, met rechts ervan een lobby, etc.

Plaatsen die we met een 0 (onbekend terrein) definiëren gaan we beschouwen als plaatsen waar de gebruiker niet mag komen.

Merk op dat we dus onze *wereld* zo groot of zo klein kunnen maken als we zelf wensen.

Stap 2: Wandelen op de kaart

Daar we ons nu op een 2D-kaart bevinden hebben we 2 variabelen nodig om onze huidige positie te onthouden:

```

int posX = 0;
int posY = 0;
```

We spreken af dat de locatie (0,0) zicht linksboven in de array bevindt.

We maken een oneindige loop die steeds de volgende stappen zal doen:

1. Huidige kamertekst op het scherm tonen
2. Aan de gebruiker vragen naar waar hij wil wandelen
3. Positie van gebruiker veranderen
4. Terug naar 1.

Eerst gebruiken we dus de `lut` om de huidige kamer beschrijving te tonen. We gebruiken de huidige spelerlocatie als index's in de `Kaart`-array en vragen zo de kamerindex op. Die kamerindex gebruiken we om de tekst uit de `Kamers`-array te tonen.

```
while (true)
{
    int kamerindex = Kaart[posX, posY];
    Console.WriteLine(Kamers[kamerindex]);
```

De gebruiker kan zich naar het noorden, oosten, zuiden of westen begeven (respectievelijk naar boven, links, onder, rechts op de kaart). We vragen dus telkens de gebruiker naar waar hij:

```
Console.WriteLine("NOZW? Naar waar wil je?");
char inp = Convert.ToChar(Console.ReadLine());
```

Stap 3: Positie aanpassen

We verwerken de richting in een switch:

```
switch (inp)
{
```

Naargelang de richting die de gebruiker ingeeft moeten we dus telkens 2 zaken controleren:

1. Bevindt de gebruiker zich momenteel (VOOR we z'n locatie aanpassen) aan de rand van de array (0 of Length-1)
2. Probeert de gebruiker zich naar verboden vakje te begeven (een **onbekend terrein** vakje)

Indien aan deze 2 voorwaarden niet is voldaan dan mogen we de huidige locatie van de gebruiker zonder problemen veranderen. Dit behelst dus dat we , naargelang de richting, de `posX` en `posY` waarden veranderen, namelijk:

- Noorden: `posX` met 1 verlagen
- Zuiden: `posX` met 1 verhogen
- Oosten: `posY` met 1 verhogen
- Westen: `posY` met 1 verlagen

We krijgen in de switch dus:

```
case 'N':
    if (posX != 0 && Kaart[posX - 1, posY] != 0)
        posX--;
    else Console.WriteLine("Kan niet");
    break;
case 'O':
    if (posY != Kaart.GetUpperBound(1) && Kaart[posX, posY + 1] != 0)
        posY++;
    else Console.WriteLine("Kan niet");
    break;
case 'Z':
    if (posX != Kaart.GetUpperBound(0) && Kaart[posX + 1, posY] != 0)
        posX++;
    else Console.WriteLine("Kan niet");
    break;
case 'W':
    if (posY != 0 && Kaart[posX, posY - 1] != 0)
        posY--;
    else Console.WriteLine("Kan niet");
    break;
}
```

Stap 4: Volledige code

De volledige code van deze fase is dus geworden:

```
int[,] Kaart =
{
    {1, 2, 1, 3},
    {0, 1, 0, 1},
    {0, 4, 0, 5}
};

string[] Kamers =
```

```

    {
        "Onbekend terrein", //0
        "In een gang", //1
        "In de lobby", //2
        "In de bar", //3
        "In de keuken", //4
        "Achtertuin"//5
    };

    int posX = 0;
    int posY = 0;

    while (true)
    {
        int kamerindex = Kaart[posX, posY];
        Console.WriteLine(Kamers[kamerindex]);

        Console.WriteLine("NOZW? Naar waar wil je?");

        char inp = Convert.ToChar(Console.ReadLine());
        switch (inp)
        {
            case 'N':
                if (posX != 0 && Kaart[posX - 1, posY] != 0)
                    posX--;
                else Console.WriteLine("Kan niet");
                break;
            case 'O':
                if (posY != Kaart.GetUpperBound(1) && Kaart[posX, posY + 1] != 0)
                    posY++;
                else Console.WriteLine("Kan niet");
                break;
            case 'Z':
                if (posX != Kaart.GetUpperBound(0) && Kaart[posX + 1, posY] != 0)
                    posX++;
                else Console.WriteLine("Kan niet");
                break;
            case 'W':
                if (posY != 0 && Kaart[posX, posY - 1] != 0)
                    posY--;
                else Console.WriteLine("Kan niet");
                break;
        }
    }
}

```

Fase 4: Tekenen van de kaart op het scherm

We wensen een visuele indicatie van de kaart te tonen aan de gebruiker (zonder dat hij ziet wat voor kamer het is). We voegen daarom een methode `DrawMap()` toe die de kaart iedere keer opnieuw zal tekenen. Deze methode gaat ook de positie van de gebruiker duidelijk maken a.d.h.v. een "X" op de kaart. Onze *game-loop* veranderen we dus naar:

```

while (true)
{
    Console.Clear();
    DrawMap(Kaart, posX, posY);

    int kamerindex = Kaart[posX, posY];
    Console.WriteLine(Kamers[kamerindex]);

    Console.WriteLine("NOZW? Naar waar wil je?")
}

```

De `DrawMap()` methode toont dus de huidige locatie als een "X". Voorts willen we dat enkel bereikbare kamers getoond worden (we gebruiken een "O" hiervoor). Elementen op de kaart die wijzen naar index 0 ("Onbekend terrein") worden niet getoond.

We doorlopen in de `DrawMap()` methode de volledige kaart. Lijn per lijn. Hiervoor gebruiken we 2 geneste for-loops. De outer-loop (index `i`) zal de X-coördinaat aflopen, oftewel lijn per lijn. De inner loop (index `j`) zal de Y-coördinaat aflopen, oftewel kolom per kolom:

```

private static void DrawMap(int[,] Kaart, int posX, int posY)
{
    for (int i = 0; i <= Kaart.GetUpperBound(0); i++)
    {
        for (int j = 0; j <= Kaart.GetUpperBound(1); j++)
        {

```

Merk op dat ook deze methode geen *hardcoded* array-grenzen bevat. We kunnen dus eender welke kaart aan deze methode aanbieden.

Binnen de inner-for gaan we nu element per element van 1 rij op het scherm tonen. Eerst controleren we of de speler zich bevindt in het element dat we op het punt staan te tekenen. Als dat zo is dan plaatsen we een "X":

```
if (posX == i & posY == j)
    Console.WriteLine("X");
```

Anders plaatsen we een "o" indien het gaat om gebied waar de speler toegelaten is:

```
else if (Kaart[i, j] != 0)
    Console.WriteLine("o");
```

Niet toegelaten gebied tonen we niet, we zetten dus een spatie in de plaats:

```
else
    Console.WriteLine(" ");
```

Na iedere inner-loop moeten we vervolgens een newline toevoegen, anders worden alle rijen van de kaart naast mekaar gezet. Finaal krijgen we dus:

```
}
Console.WriteLine("\n");
}
```

Dit resulteert in volgende finale code voor deze fase:

```
static void Main()
{
    int[,] Kaart = { {1, 2, 1, 3}, {0, 1, 0, 1}, {0, 4, 0, 5} };

    string[] Kamers = {"Onbekend terrein", "In een gang", "In de lobby", "In de bar", "In de keuken", "Achtertuin"};

    int posX = 0;
    int posY = 0;

    while (true)
    {
        Console.Clear();
        DrawMap(Kaart, posX, posY);

        int kamerindex = Kaart[posX, posY];
        Console.WriteLine(Kamers[kamerindex]);
        Console.WriteLine("NOZW? Naar waar wil je?");

        char inp = Convert.ToChar(Console.ReadLine());
        switch (inp)
        {
            case 'N':
                if (posX != 0 && Kaart[posX - 1, posY] != 0)
                    posX--;
                else Console.WriteLine("Kan niet");
                break;
            case 'O':
                if (posY != Kaart.GetUpperBound(1) && Kaart[posX, posY + 1] != 0)
                    posY++;
                else Console.WriteLine("Kan niet");
                break;
            case 'Z':
                if (posX != Kaart.GetUpperBound(0) && Kaart[posX + 1, posY] != 0)
                    posX++;
                else Console.WriteLine("Kan niet");
                break;
            case 'W':
                if (posY != 0 && Kaart[posX, posY - 1] != 0)
                    posY--;
                else Console.WriteLine("Kan niet");
                break;
        }
    }

    private static void DrawMap(int[,] Kaart, int posX, int posY)
    {
        for (int i = 0; i <= Kaart.GetUpperBound(0); i++)
        {
            for (int j = 0; j <= Kaart.GetUpperBound(1); j++)
            {
                if (posX == i & posY == j)
                    Console.WriteLine("X");
                else if (Kaart[i, j] != 0)
                    Console.WriteLine("o");
                else
                    Console.WriteLine(" ");
            }
        }
    }
}
```

```

    {
        for (int j = 0; j <= Kaart.GetUpperBound(1); j++)
        {
            if (posX == i & posY == j)
                Console.Write("X");
            else if (Kaart[i, j] != 0)
                Console.Write("o");
            else
                Console.Write(" ");
        }
        Console.WriteLine("\n");
    }
}

```

Fase 5: Kaart vergroten

Zoals reeds aangehaald staat niets je in de weg om je spel-wereld groter te maken. Hiervoor hoeft je enkel (momenteel) de Kamers en Kaart arrays aan te passen. Alle code zal blijven werken.

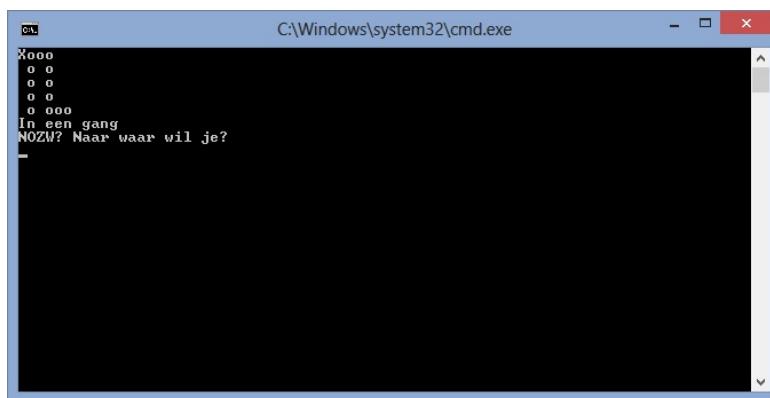
Bijvoorbeeld:

```

int[,] Kaart =
{
    {1, 2, 1, 3, 0, 0},
    {0, 1, 0, 1, 0, 0},
    {0, 4, 0, 1, 0, 0},
    {0, 1, 0, 1, 0, 0},
    {0, 7, 0, 6, 1, 8},
};

string[] Kamers =
{
    "Onbekend terrein", //0
    "In een gang", //1
    "In de lobby", //2
    "In de bar", //3
    "In de keuken", //4
    "Achtertuin", //5
    "In de securityroom", //6
    "In de personeelsruimte", //7
    "In de folterkamer" //8
};

```



Fase 6: Een extra look-up-table voor meer wereld-details

Het principe van een lut verschilt eigenlijk weinig van een eenvoudige database. We zouden dus meerdere look-up-tables (tabellen) kunnen definiëren en deze gebruiken om meer *informatie* in onze spelwereld te plaatsen.

We kunnen bijvoorbeeld per kamer ook een beschrijving tonen van die kamer. Daar we nog niets kennen van `struct` en `class` (zogenaamde datastructuren) moeten we ons dus behelpen als volgt: we definiëren een nieuwe array `Beschrijving` waarbij ieder element de index heeft van de respectievelijke kamer:

```

int[,] Kaart =

```

```

    {
        {1, 2, 1, 3, 0, 0},
        {0, 1, 0, 1, 0, 0},
        {0, 4, 0, 1, 0, 0},
        {0, 1, 0, 1, 0, 0},
        {0, 7, 0, 6, 1, 8},
    };

    string[] Kamers =
    {
        "Onbekend terrein", //0
        "In een gang", //1
        "In de lobby", //2
        "In de bar", //3
        "In de keuken", //4
        "Achtertuin", //5
        "In de securityroom", //6
        "In de personeelsruimte", //7
        "In de folterkamer" //8
    };

    string[] Beschrijving =
    {
        "", //0
        "Een ordinaire saaie gang met een mooie vloer", //1
        "De receptioniste kijkt je verbaast aan. Een plant in de hoek is het enige groen in de purperen ruimte.", //2
        "2 gasten zitten half beschonken aan de toog. Een verliefd koppel is zachtjes aan het praten", //3
        "Overal liggen etensresten, maar verder is hier niets of niemand interessant.", //4
        "Mooie plantjes, enkele bomen en een gezellig terras.", //5
        "De veiligheidsagent houdt je nauwlettend in het oog", //6
        "Overal staan kastjes. Hier en daar is er een personeelslid zich aan het omkleden", //7
        "Wat doet deze vreemde plek in het hotel." //8
    };
}

```

Door 1 extra lijntje (+ een lijntje voor een visuele scheiding tussen beschrijving en kamertitel) plaatsen we nu steeds de kamerbeschrijving onder de kamertype:

```

Console.WriteLine(Kamers[kamerindex]);
Console.WriteLine("*****");
Console.WriteLine(Beschrijving[kamerindex]);

```



Fase 7: Dynamische kaart

Na iedere actie van de speler verwerken we steeds weer de kaart in zowel de `DrawMap()`-methode als tijdens het verwerken van de spelerinput. We kunnen dus eenvoudig een *dynamische* kaart maken die zich aanpast naargelang bepaalde acties.

Je moet de kennis die we zo meteen tonen bijvoorbeeld aan de start van het programma met een lege kaart: naargelang de speler zich verplaatst in de wereld zal de kaart aangevuld worden. (tip: gebruik hiervoor een array `VolledigeKaart` en een array `ReedsOntdekteKaart` of iets dergelijks. De speler krijgt steeds de `ReedsOntdekteKaart` te zien in `DrawMap()`. Naargelang acties van de speler kopieer je dan bepaalde elementen uit `VolledigeKaart` naar `ReedsOntdekteKaart`).

We definiëren onze kaart (merk op dat we de folterkamer en geheime gang verwijderen rechts onderraan):

```

int[,] Kaart =

```

```

    {
        {1, 2, 1, 3, 0, 0},
        {0, 1, 0, 1, 0, 0},
        {0, 4, 0, 1, 0, 0},
        {0, 1, 0, 1, 0, 0},
        {0, 7, 0, 6, 0, 0},
    };
}

```

We willen volgende functionaliteit inbouwen:

indien de gebruiker in de kamer met index 6 ("SecurityRoom") een bepaalde actie onderneemt dan zal een *geheime gang en kamer* (folterkamer) op de kaart bij verschijnen, rechts van de securityroom.

De actie gaan we nu even eenvoudig beschouwen als volgt: de gebruiker kan in alle kamers "G" als opdracht doorgeven. Echter, enkel wanneer de gebruiker zich in de kamer met index 6 bevind dan zal de geheime kamer zichtbaar worden. We voegen daarom een extra case toe aan onze switch:

```

case 'G':
    if (kamerindex != 6)
    {
        Console.WriteLine("Dat zal hier niet werken");
    }
    else
    {

```

Als de speler wél in de securityroom is dan gaan we de kaart-array aanpassen. We voegen rechtsonder in de array de 2 nieuwe kamers toe:

```

Console.WriteLine("Je ontdekt een geheime ruimte");
Kaart[4, 4] = 1;
Kaart[4, 5] = 8;

```

Wanneer we nu de kaart hertekenen dan deze nieuwe ruimte verschijnen en weet de gebruiker dat hij zich daar kan begeven.

De volledige code wordt dan (we laten de DrawMap()-methode even achterwege):

```

static void Main()
{
    int[,] Kaart =
    {
        {1, 2, 1, 3, 0, 0}, {0, 1, 0, 1, 0, 0}, {0, 4, 0, 1, 0, 0}, {0, 1, 0, 1, 0, 0}, {0, 7, 0, 6, 0, 0},
    };

    string[] Kamers =
    {
        "Onbekend terrein", //0
        "In een gang", //1
        "In de lobby", //2
        "In de bar", //3
        "In de keuken", //4
        "Achtertuin",//5
        "In de securityroom", //6
        "In de personeelsruimte", //7
        "In de folterkamer" //8
    };

    string[] Beschrijving =
    {
        "", //0
        "Een ordinaire saaie gang met een mooie vloer", //1
        "De receptioniste kijkt je verbaast aan. Een plant in de hoek is het enige groen in de purperen ruimte.", //2
        "2 gasten zitten half beschonken aan de toog. Een verliefd koppel is zachtjes aan het praten", //3
        "Overal liggen etensresten, maar verder is hier niets of niemand interessant.", //4
        "Mooie plantjes, enkele bomen en een gezellig terras.",//5
        "De veiligheidsagent houdt je nauwlettend in het oog. Typ \"G\" om een geheime ruimte te ontdekken", //6
        "Overal staan kastjes. Hier en daar is er een personeelslid zich aan het omkleden.", //7
        "Wat doet deze vreemde plek in het hotel." //8
    };

    int posX = 0;
    int posY = 0;

    while (true)
    {

```

```

Console.Clear();
DrawMap(Kaart, posX, posY);

int kamerindex = Kaart[posX, posY];
Console.WriteLine(Kamers[kamerindex]);
Console.WriteLine("*****");
Console.WriteLine(Beschrijving[kamerindex]);
Console.WriteLine();
Console.WriteLine("NOZW? Naar waar wil je?");

char inp = Convert.ToChar(Console.ReadLine());
switch (inp)
{
    case 'N':
        if (posX != 0 && Kaart[posX - 1, posY] != 0)
            posX--;
        else Console.WriteLine("Kan niet");
        break;
    case 'O':
        if (posY != Kaart.GetUpperBound(1) && Kaart[posX, posY + 1] != 0)
            posY++;
        else Console.WriteLine("Kan niet");
        break;
    case 'Z':
        if (posX != Kaart.GetUpperBound(0) && Kaart[posX + 1, posY] != 0)
            posX++;
        else Console.WriteLine("Kan niet");
        break;
    case 'W':
        if (posY != 0 && Kaart[posX, posY - 1] != 0)
            posY--;
        else Console.WriteLine("Kan niet");
        break;
    case 'G':
        if (kamerindex != 6)
        {
            Console.WriteLine("Dat zal hier niet werken");
        }
        else
        {
            Console.WriteLine("Je ontdekt een geheime ruimte");
            Kaart[4, 4] = 1;
            Kaart[4, 5] = 8;
            Console.ReadLine();
        }
        break;
}
}
}

```

The image contains two separate windows of a command-line interface (CMD) running on Windows. Both windows show identical text logs from a game:

```

C:\Windows\system32\cmd.exe
0000
0 0
0 0
0 0
0 X
In de securityroom
*****
De veiligheidsagent houdt je nauwlettend in het oog. Typ "G" om een geheime ruimte te ontdekken
NOZW? Naar waar wil je?
G
Je ontdekt een geheime ruimte
-

```

The second window shows the same log, but with a different room name:

```

C:\Windows\system32\cmd.exe
0000
0 0
0 0
0 0
0 ox
In de folterkamer
*****
Wat doet deze vreemde plek in het hotel.
NOZW? Naar waar wil je?
-
```

Fase 8: Go nuts

Vanaf dit punt kun je nu al een relatief eenvoudig, toch leuk spel maken, op voorwaarde dat je *verhaal* goed zit. Echter, voor we je hierop loslaten gaan we nog enkele zaken *refactoren* zodat de code wat leesbaarder blijft. In hoofzaak willen we bepaalde stukken code uit de main-body halen en naar aparte methodes extraheren.

Stap 1: Kaart initialiseren in aparte methode

Beeld je in dat je de kaart(en) voor je spel uit een bestand laadt. Op zich is dat niet zo moeilijk, maar het vereist natuurlijk extra lijnen code in je, reeds overbevolkte, Main-methode. We verhuizen daarom de code waarin we onze kaarten initialiseren naar een aparte methode. In onze Main schrijven we dan (merk het gebruik van het `out` keyword op!):

```

int[,] Kaart;
string[] Kamers;
string[] Beschrijving;

InitialiseerSpel(out Kaart, out Kamers, out Beschrijving);

```

Deze methode bevat dan gewoon de code van daarnet, mooi verpakt en afgeschermd:

```

private static void InitialiseerSpel(
    out int[,] Kaart,
    out string[] Kamers,
    out string[] Beschrijving)
{
    Kaart = new int[,]
    {
        {1, 2, 1, 3, 0, 0},
        {0, 1, 0, 1, 0, 0},
        {0, 4, 0, 1, 0, 0},
        {0, 1, 0, 1, 0, 0},
        {0, 7, 0, 6, 0, 0}
    };
    //Idem voor Kamers en beschrijving
    //..
}

```

Stap 2: Input verwerken in aparte methode

Het verwerken van de userinput kunnen we ook makkelijk extraheren naar aparte methode zodat onze while-loop overzichtelijker wordt :

```
while (true)
{
    Console.Clear();
    DrawMap(Kaart, posX, posY);

    int kamerindex = Kaart[posX, posY];
    Console.WriteLine(Kamers[kamerindex]);

    Console.WriteLine("*****");
    Console.WriteLine(Beschrijving[kamerindex]);
    Console.WriteLine();
    Console.WriteLine("NOZW? Naar waar wil je?");

    VerwerkInput(Kaart, kamerindex, ref posX, ref posY); //NEW
}
```

Merk op dat we zelfs de volledige loop naar een aparte methode op zijn beurt kunnen extraheren. Maar dat laten we aan de lezer over. We dienen de posities van de speler by reference mee te geven, daar we de posities onmiddellijk willen updaten in de `VerwerkInput()`-methode.

```
private static void VerwerkInput(
    int[,] Kaart,
    int kamerindex,
    ref int posX,
    ref int posY)
{
    char inp = Convert.ToChar(Console.ReadLine());
    switch (inp)
    {
        //etc
```

Stap 3: Een mooier kaartje tekenen

Als kers op de taart tonen we snel hoe je het kaartje *sexier* kan tonen op het scherm. Hier zijn echter een paar belangrijke opmerkingen aan de orde:

- De code bevat enkele *hardcoded* waarden zoals het plaatsen van de cursor m.b.v. `Console.SetCursorPosition`. Beter zou zijn als deze waarden als Magic numbers worden behandeld of on-the-fly worden berekend.
- De kaart bevat ascii-art met vaste grootte. Dit zal bugs geven indien onze kaart-array groter is dan de dimensies van de ascii-art: de art zal over de randen van de ascii-art getekend worden. We kunnen dit oplossen door delen van de ascii-art te berekenen (bv het aantal *lege lijnen* en de breedte van een pagina).

We definiëren de nieuwe Methode en voegen als eerste actie ascii-art toe van een kaart:

```
private static void DrawMapCool(int[,] Kaart, int posX, int posY)
{
    string background =
        @".-/|~~~~~MAP~~~~~\|/~*~~~|\ \-." + "\n" +
        @"||||| : ||||" + "\n" +
        @"|||/=====:\|/=====||" + "\n" +
        @"`-----";
```

Daar we gaan spelen met de kleuren is het aan te raden om steeds volgende acties te ondernemen indien we de kleur van een bepaald karakter of zinnen willen veranderen:

1. De huidige kleur van de console bewaren (fore en/of background) in een tijdelijke variabele
2. Kleur veranderen
3. Karakter of zin op scherm plaatsen
4. Kleur terug naar de huidige kleur aanpassen. We tonen dit in de volgende code waarin we de background array (die de ascii-art bevat) op het scherm willen tekenen. Daarbij willen we dat de karakters donker-cyaan zijn en dat enkel karakters die geen spatie of liggenstreepje zijn een donkergele achtergrond hebben. De commentaar toont de zonet beschreven stappen:

```

ConsoleColor oll = Console.ForegroundColor; //1
Console.ForegroundColor = ConsoleColor.DarkCyan; //2
for (int i = 0; i < background.Length; i++)
{
    if (background[i] != ' ' && background[i] != '_')
        Console.Write(background[i]); //3
    else
    {
        ConsoleColor bll = Console.BackgroundColor; //1
        Console.BackgroundColor = ConsoleColor.DarkYellow; //2
        Console.WriteLine(background[i]); //3
        Console.BackgroundColor = bll; //4
    }
}
Console.ForegroundColor = oll; //4

```

Vervolgens gebruiken we SetCursorPosition om onze spelerskaart 'over' de Ascii-art te tekenen. Hierbij voegen we nog wat extra kleurtje toe, de speler-X wordt rood gekleurd:

```

 ConsoleColor bll2 = Console.BackgroundColor;
Console.BackgroundColor = ConsoleColor.DarkYellow;

for (int i = 0; i <= Kaart.GetUpperBound(0); i++)
{
    Console.SetCursorPosition(7, 3 + i);
    for (int j = 0; j <= Kaart.GetUpperBound(1); j++)
    {
        if (posX == i & posY == j)
        {
            ConsoleColor l = Console.ForegroundColor;
            Console.ForegroundColor = ConsoleColor.Red;
            Console.Write("X");
            Console.ForegroundColor = l;
        }
        else
            if (Kaart[i, j] != 0)
                Console.Write("o");
            else
                Console.Write(" ");
    }
    Console.WriteLine('\n');
}
Console.SetCursorPosition(1, 15);
Console.BackgroundColor = bll2;
}

```

De while-loop in de Main()-methode passen we nu nog aan zodat:

1. We de nieuwe DrawCoolMap methode gebruiken
2. De titel van de kamer steeds op de rechterpagina van de kaart ascii-art wordt getoond
3. De beschrijving en andere tekst steeds onder map komt en niet erover

```

while (true)
{
    Console.Clear();
    DrawMapCool(Kaart, posX, posY); //a

    int kamerindex = Kaart[posX, posY];
    ConsoleCursorPosition(26, 6); //b
    Console.WriteLine(Kamers[kamerindex]);

    ConsoleCursorPosition(1, 16); //c
    Console.WriteLine("*****");
    Console.WriteLine(Beschrijving[kamerindex]);
    Console.WriteLine();
    Console.WriteLine("NOZW? Naar waar wil je?");

    VerwerkInput(Kaart, kamerindex, ref posX, ref posY);
}

```

Alle code samen



De volledige code voor dit extra-

ordinaire spel wordt dan:

Main-methode

```
static void Main()
{
    int[,] Kaart;
    string[] Kamers;
    string[] Beschrijving;

    InitialiseerSpel(out Kaart, out Kamers, out Beschrijving);

    int posX = 0;
    int posY = 0;

    while (true)
    {
        Console.Clear();
        DrawMapCool(Kaart, posX, posY); //a

        int kamerindex = Kaart[posX, posY];
        Console.SetCursorPosition(26, 6); //b
        Console.WriteLine(Kamers[kamerindex]);

        Console.SetCursorPosition(1, 16); //c
        Console.WriteLine("*****");
        Console.WriteLine(Beschrijving[kamerindex]);
        Console.WriteLine();
        Console.WriteLine("NOZW? Naar waar wil je?");

        VerwerkInput(Kaart, kamerindex, ref posX, ref posY);
    }
}
```

InitialiseerSpel-methode

```
private static void InitialiseerSpel(
    out int[,] Kaart,
    out string[] Kamers,
    out string[] Beschrijving)
{
    Kaart = new int[,]
    {
        {1, 2, 1, 3, 0, 0},
        {0, 1, 0, 1, 0, 0},
        {0, 4, 0, 1, 0, 0},
        {0, 1, 0, 1, 0, 0},
        {0, 7, 0, 6, 0, 0}
    };

    Kamers = new string[]
    {
        "Onbekend terrein", "In een gang", "In de lobby", "In de bar", "In de keuken", "Achtertuin",
        "In de securityroom", "In de personeelsruimte", "In de folterkamer"
    };

    Beschrijving = new string[]
    {
        "", "Een ordinaire saaie gang met een mooie vloer",
        "De veiligheidsagent houdt je nauwlettend in het oog. Typ \"G\" om een geheime ruimte te ontdekken"
    };
}
```

```

        "De receptioniste kijkt je verbaast aan. Een plant in de hoek is het enige groen in de purperen ruimte.",
        "2 gasten zitten half beschonken aan de toog. Een verliefd koppel is zachtjes aan het praten",
        "Overal liggen etensresten, maar verder is hier niets of niemand interessant.",
        "Mooie plantjes, enkele bomen en een gezellig terras.",
        "De veiligheidsagent houdt je nauwlettend in het oog. Typ \"G\" om een geheime ruimte te ontdekken",
        "Overal staan kastjes. Hier en daar is er een personeelslid zich aan het omkleden.",
        "Wat doet deze vreemde plek in het hotel."
    );
}
}

```

VerwerkInput-methode

```

private static void VerwerkInput(
    int[,] Kaart,
    int kamerindex,
    ref int posX,
    ref int posY)
{
    char inp = Convert.ToChar(Console.ReadLine());
    switch (inp)
    {
        //etc
        case 'N':
            if (posX != 0 && Kaart[posX - 1, posY] != 0)
                posX--;
            else Console.WriteLine("Kan niet");
            break;
        case 'O':
            if (posY != Kaart.GetUpperBound(1) && Kaart[posX, posY + 1] != 0)
                posY++;
            else Console.WriteLine("Kan niet");
            break;
        case 'Z':
            if (posX != Kaart.GetUpperBound(0) && Kaart[posX + 1, posY] != 0)
                posX++;
            else Console.WriteLine("Kan niet");
            break;
        case 'W':
            if (posY != 0 && Kaart[posX, posY - 1] != 0)
                posY--;
            else Console.WriteLine("Kan niet");
            break;
        case 'G':
            if (kamerindex == 6)
            {
                Console.WriteLine("Je ontdekt een geheime ruimte");
                Kaart[4, 4] = 1;
                Kaart[4, 5] = 8;
                Console.ReadLine();
            }
            else
            {
                Console.WriteLine("Dat zal hier niet werken");
            }
            break;
    }
}

```

DrawMapCool-methode

```

private static void DrawMapCool(int[,] Kaart, int posX, int posY)
{
    string background =
        @".-~/|~~~~~MAP~~~~~\~/~~~~~*~~~~~|\~.-" + "\n" +
        @"|||| : ||||" + "\n" +
        @"|||| : ||||" + "\n" +
        @"| ||| : _____| ||||" + "\n" +
        @"| | /=====:\| /=====\\| " + "\n" +
        @"`-----' ";
}

```

```

ConsoleColor oll = Console.ForegroundColor; //1
Console.ForegroundColor = ConsoleColor.DarkCyan; //2
for (int i = 0; i < background.Length; i++)
{
    if (background[i] != ' ' && background[i] != '_')
        Console.Write(background[i]); //3
    else
    {
        ConsoleColor bll = Console.BackgroundColor;//1
        Console.BackgroundColor = ConsoleColor.DarkYellow;//2
        Console.Write(background[i]);//3
        Console.BackgroundColor = bll;//4
    }
}

Console.ForegroundColor = oll;//4

ConsoleColor bll2 = Console.BackgroundColor;
Console.BackgroundColor = ConsoleColor.DarkYellow;

for (int i = 0; i <= Kaart.GetUpperBound(0); i++)
{
    Console.SetCursorPosition(7, 3 + i);
    for (int j = 0; j <= Kaart.GetUpperBound(1); j++)
    {
        if (posX == i & posY == j)
        {
            ConsoleColor l = Console.ForegroundColor;
            Console.ForegroundColor = ConsoleColor.Red;
            Console.Write("X");
            Console.ForegroundColor = l;
        }
        else
        if (Kaart[i, j] != 0)
            Console.Write("o");
        else
            Console.Write(" ");
    }
    Console.WriteLine();
}

}
Console.SetCursorPosition(1, 15);
Console.BackgroundColor = bll2;
}
}

```

Dit hoofdstuk is verre van compleet (itt andere hoofdstukken van dit semester in deze cursus)

TODO:

Klassen en objecten aanmaken

Public en private

Object methoden

Object constructors en constructor overloading

Object initializer syntax

Objecten als parameters en returntypes

Object references en null

Object arrays

Static

Operator overloading

Constructors

Properties

In een wereld zonder properties

Properties (*eigenschappen*) zijn de C# manier om objecten hun interne staat in en uit te lezen. Ze zorgen voor een gecontroleerde toegang tot de interne structuur van je objecten.

Stel dat we volgende klasse hebben:

```
class SithLord
{
    private int energy;
    private string sithName;
}
```

Een `SithLord` heeft steeds een verborgen Sith Name en ook een hoeveelheid energie die hij nodig heeft om te strijden. **Het is uit den boze dat we eenvoudige data fields (`energy` en `name`) public maken.** Zouden we dat wel doen dan kunnen externe objecten deze geheime informatie uitlezen!

```
SithLord Palpatine= new SithLord();
Console.WriteLine(Palpatine.sithName);
```

We willen echter wel van buitenuit het energy-level van een `sithLord` kunnen instellen. Maar ook hier hetzelfde probleem: wat als we de `energy`-level op -1000 instellen? Terwijl `energy` nooit onder 0 mag gaan.

Properties lossen dit probleem op

Full properties

Een **full property** ziet er als volgt uit:

```
class SithLord
{
    private int energy;
    private string sithName;

    public int Energy
    {
        get
        {
            return energy;
        }
        set
        {
            energy=value;
        }
    }
}
```

We zullen de property stuk per stuk analyseren:

- `public int Energy`: een property is altijd `public`. Vervolgens zeggen we wat voor type de property moet zijn en geven we het een naam. Indien je de property gaat gebruiken om een intern dataveld naar buiten beschikbaar te stellen, dan is het een goede gewoonte om dezelfde naam als dat veld te nemen maar nu met een hoofdletter. (dus `Energy` i.p.v. `energy`).
- `{ }:` Vervolgens volgen 2 accolades waarbinnen we de werking van de property beschrijven.
- `get {}`: indien je wenst dat de property data **naar buiten** moet sturen, dan schrijven we de get-code. Binnen de accolades van de get schrijven we wat er naar buiten moet gestuurd worden. In dit geval `return energy` maar dit mag even goed bijvoorbeeld `return 4` of een hele reeks berekeningen zijn. Het element dat je returnt in de get code moet uiteraard van hetzelfde type zijn als waarmee je de property hebt gedefinieerd (`int` in dit geval).
 - We kunnen nu van buitenaf toch de waarde van `energy` uitlezen via de property en het get-gedeelte:
`Console.WriteLine(Palpatine.Energy);`
- `set{ }:` in het set-gedeelte schrijven we de code die we moeten hanteren indien men van buitenuit een waarde aan de property wenst te geven om zo een interne variabele aan te passen. De waarde die we van buitenuit krijgen (als een parameter zeg maar) zal **altijd** in

een lokale variabele `value` worden bewaard. Deze zal van het type van de property zijn. Vervolgens kunnen we `value` toewijzen aan de interne variabele indien gewenst: `energy=value`

- We kunnen vanaf nu van buitenaf waarden toewijzen aan de property en zo `energy` vtoch bereiken: `Palpatine.Energy=50` .

Snel property schrijven

Visual Studio heeft een ingebouwde shortcut om snel een full property, inclusief een bijhorende private dataveld, te schrijven. **Typ `propfull` gevolgd door twee tabs!**

Full property met toegangscontrole

De full property `Energy` heeft nog steeds het probleem dat we negatieve waarden kunnen toewijzen (via de `set`) die dan vervolgens zal toegewezen worden aan `energy`.

Properties hebben echter de mogelijkheid om op te treden als wachters van en naar de interne staat van objecten.

We kunnen in de `set` code extra controles inbouwen. Als volgt:

```
public int Energy
{
    get
    {
        return energy;
    }
    set
    {
        if(value>=0)
            energy=value;
    }
}
```

Enkel indien de toegewezen waarde groter of gelijk is aan 0 zal deze ook effectief aan `energy` toegewezen worden. Volgende lijn zal dus geen effect hebben: `Palpatine.Energy=-1;`

We kunnen de code binnen `set` (en `get`) zo complex als we willen maken.

Property variaties

We zijn niet verplicht om zowel de `get` en de `set` code van een property te schrijven.

Read-only property

```
public int Energy
{
    set
    {
        if(value>=0)
            energy=value;
    }
}
```

We kunnen dus enkel `energy` uitlezen, maar niet van buitenaf aanpassen.

Write-only property

```
public int Energy
{
    get
    {
        return energy;
    }
}
```

We kunnen dus enkel `energy` een waarde geven, maar niet van buitenaf uitlezen.

Read-only property met private set

Soms gebeurt het dat we van buitenuit enkel de gebruiker de property read-only willen maken. We willen echter intern (in de klasse zelf) nog steeds controleren dat er geen illegale waarden aan private datafields worden gegeven. Op dat moment definieren we een read-only property met een private setter:

```
public int Energy
{
    get
    {
        return energy;
    }
    private set
    {
        if(value>=0)
            energy=value;
    }
}
```

Van buitenuit zal enkel code werken die de `get`-`van deze property aanroeft: `Console.WriteLine(Palpatine.Energy);`. Code die de `set` van buitenuit nodig heeft zal een fout geven zoals: `Palpatine.Energy=65`; ongeacht of deze geldig is of niet.

Nu goed opletten: indien we in het object de property willen gebruiken dan moeten we deze dus ook effectief aanroepen, anders omzeilen we hem als we rechtstreeks `energy` instellen.

Kijk zelf naar volgende **slechte** code:

```
class SithLord
{
    private int energy;
    private string sithName;

    public void ResetLord()
    {
        energy=-1;
    }

    public int Energy
    {
        get
        {
            return energy;
        }
        private set
        {
            if(value>=0)
                energy=value;
        }
    }
}
```

De nieuw toegevoegde methode `ResetLord` willen we gebruiken om de lord z'n energy terug te verlagen. Door echter `energy=-1` te schrijven geven we dus -1 rechtstreeks aan `energy`. Nochtans is dit een illegale waarden. We moeten dus in de methode ook expliciet via de property gaan en dus schrijven:

```
public void ResetLord()
{
    Energy=-1; // Energy i.p.v. energy
}
```

Het is een goede gewoonte om zo vaak mogelijk via de properties je interne variabele aan te passen en niet rechtstreeks het datafield zelf.

Read-only Get-omvormers

Je bent uiteraard niet verplicht om voor iedere interne variabele een bijhorende property te schrijven. Omgekeerd ook: mogelijk wil je extra properties hebben voor data die je 'on-the-fly' kan genereren.

Stel dat we volgende klasse hebben

```
class Persoon
```

```
{
    private string voornaam;
    private string achternaam;
}
```

We willen echter ook soms de volledige naam op het scherm tonen ("Voornaam + Achternaam"). Via een read-only property kan dit supereenvoudig:

```
class Persoon
{
    private string voornaam;
    private string achternaam;
    public string FullName
    {
        get{ return $"{voornaam} {achternaam}"; }
    }
}
```

Of nog eentje:

```
class Aarde
{
    public double ZwaarteKracht
    {
        get
        {
            return 9.81;
        }
    }
}
```

Nog een voorbeeldje:

```
class Persoon
{
    private int age;

    public bool IsWaarschijnlijkNogLevend
    {
        get
        {
            if(age>120) return false;
            return true;
        }
    }
}
```

Vaak gebruiken we dit soort read-only properties om data te transformeren. Stel bijvoorbeeld dat we volgende klasse hebben:

```
class Persoon
{
    private int age; //in jaren

    public int LeeftijdInMaanden
    {
        get
        {
            return age*12;
        }
    }
}
```

Auto properties

Automatische eigenschappen (autoproperties) in C# staan toe om eigenschappen (properties) die enkel een waarde uit een private variabele lezen en schrijven verkort voor te stellen.

Zo kan je eenvoudig de klasse Person herschrijven met behulp van autoproperties. De originele klasse:

```
public class Person
{
    private string _firstName;
    private string _lastName;
    private int _age;

    public string FirstName
    {
        get
        {
            return _firstName;
        }
        set
        {
            _firstName = value;
        }
    }

    public string LastName
    {
        get
        {
            return _lastName;
        }
        set
        {
            _lastName = value;
        }
    }

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

De herschreven klasse met autoproperties (autoprops):

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

Beide klassen hebben exact dezelfde functionaliteit, echter de klasse aan de rechterzijde is aanzienlijk eenvoudiger om te lezen en te typen.

Beginwaarde van autoprops

Je mag autoproperties beginwaarden geven door de waarde achter de property te geven, als volgt:

```
public int Age {get;set;} = 45;
```

Altijd auto-properties?

Merk op dat je dit enkel kan doen indien er geen extra logica in de property aanwezig is. Stel dat je bij de setter van age wil controleren op een negatieve waarde, dan zal je dit zoals voorheen moeten schrijven en kan dit niet met een automatic property:

```
set
{
    if( value > 0)
        _age = value;
}
```

Voorgaande property kan dus NIET herschreven worden met een automatic property.

Alleen-lezen eigenschap

Je kan automatic properties ook gebruiken om bijvoorbeeld een read-only property te definiëren . Als volgt:

Originele klasse:

```
public string FirstName
{
    get
    {
        return _firstName;
    }
}
```

Met autoprops:

```
public string FirstName { get; private set; }
```

En andere manier die ook kan is als volgt:

```
public string FirstName { get; }
```

De enige manier om FirstName een waarde te geven is via de constructor van de klasse. Alle andere manieren zal een error genereren.[Meer info](#).

Snel autoproperties typen in Visual Studio:

Als je in Visual Studio in je code `prop` typt en vervolgens twee keer de tabtoets indrukt dan verschijnt al de nodige code voor een automatic property. Je hoeft dan enkel nog volgende zaken in orde te brengen:

- Het type van de property
- De naam van de property (identifier)
- De toegankelijkheid van get/set (public, private, protected)

Static

TODO

Expression bodied members

Wanneer je methoden, constructors of properties schrijft waar **exact 1 expressie** (*1 lijn code die een resultaat teruggeeft*) nodig is dan kan je gebruik maken van de **expression bodied member syntax** (EBM). Deze is van de vorm:

```
member => expression
```

Dankzij EBM kan je veel kortere code schrijven.

We tonen telkens een voorbeeld hoe deze origineel is en hoe deze naar EBM syntax kan omgezet worden.

Methoden en EBM

Origineel:

```
public void ToonLeeftijd(int age)
{
    Console.WriteLine(age);
}
```

Met EBM:

```
public void ToonLeeftijd(int age) => Console.WriteLine(age);
```

Nog een voorbeeld, nu met een return. Merk op dat we return niet moeten schrijven:

```
public int GeefGewicht()
{
    return 4 * 34;
}
```

Met EBM:

```
public int GeefGewicht() => 4 * 34;
```

Constructors en EBM

Ook constructors die maar 1 expressie bevatten kunnen koper nu. Origineel:

```
class Student
{
    private int age;
    public Student(int inage)
    {
        age = inage;
    }
}
```

Met EBM:

```
class Student
{
    private int age;
    public Student(int inage) => age = inage;
}
```

Full Properties met EBM

Properties worden een soort blend tussen full en autoproperties. Originele full property:

```
private int name;
public int Name
{
    get
    {
        return name;
    }
    set
    {
        name=value;
    }
}
```

Met EBM:

```
private int name;
public int Name
{
    get => name;
    set => name=value;
}
```

Read-only properties met EBM

Bij read-only properties hoeft het `get` keyword zelfs niet meer getypt te worden bij EBM.

Origineel:

```
private int name;
public int Name
{
    get
    {
        return name;
    }
}
```

Met EBM:

```
private int name;
public int Name => name;
```

Practica klassen

Klassen objecten

Sports

Kies je favoriete sport of game (voor zij die enkel pc/console-sporten :p). Maak een klasse aan die een speler uit deze sport kan voorstellen. Verzin een 4-tal datavelden die deze spelers hebben, alsook 2 methoden die de speler moet kunnen uitvoeren. Je mag deze datavelden en methoden voorlopig allemaal 'public' zetten (snap je properties al? Cool, ga dan ineens voor properties!).

Voorzie ook minstens 1 "Naam" (string) dataveld.

Bijvoorbeeld:

klasse Waterpolospeler,

datavelden: SpelerNaam(string), Mutsnummer (int), IsDoelman (bool), IsReserve(bool), Reeks (string, bv "Cadet").

Methoden: GooiBal, Watertrappen

Wanneer de methoden worden aangeroepen zal er een tekst (mbv console.WriteLine in de methode) op het scherm verschijnen die bv zegt
Ik (Jos) gooい de bal . Waarbij de naam van de speler in kwestie uit het Naam dataveld wordt gebruikt om mee getoond te worden.

Maak vervolgens een console-applicatie aan waarin je de werking van de klasse aantoon. Maar in de applicatie een aantal speler-objecten aan, vervolgens stel je (via code) hun datavelden in. Vervolgens roep je enkele methoden van de spelers aan en toon je via (Console.WriteLine) ook de datavelden van de individuele spelerobjecten.

Toon maw aan dat je:

- Een klasse kunt maken (in een aparte file!)
- Instanties (objecten) van deze klasse kunt maken
- Kunt werken met deze instanties (datavelden instellen én uitlezen, aanroepen van methoden)

Vervolgens: Schrijf een methode genaamd: static void SimuleerSpeler(Waterpolospeler testspeler)

(vervang Waterpolospeler door de klasse die je zelf hebt gemaakt)

De SimuleerSpeler-methode zal beide methoden van je klasse telkens 3x aanroepen m.b.v. een for-loop in de methode (dus in mijn geval 3x GooiBal en 3xWatertrappen)

Test je methode door 2 objecten aan te maken en telkens mee te geven als parameter.

Alledaags

Zoek een foto naar keuze (nieuws, privé, etc) waarop meer dan één element opstaat (dus geen pasfoto of foto van blauwe lucht zonder wolken). Tracht de nodige klassen te verzinnen (met enkele datavelden en methoden) en maak in een console-applicatie vervolgens objecten van deze klassen aan. Voeg de foto aan je solution-folder toe.

Bijvoorbeeld: een foto van een betoging. Je zou minstens 3 klassen kunnen verzinnen (gebouw, politie, betoger). Van ieder van deze klassen maak je dan objecten aan zoals je ze op de foto ziet (uiteraard gaan we geen 30 betoger-instanties maken, enkele zijn genoeg, als voorbeeld).

RapportModule

Ontwerp een klasse Resultaat die je zal tonen wat je graad is gegeven een bepaalde behaalde percentage. Het enige dat je aan een Resultaat-object moet kunnen geven is het behaalde percentage. Enkel het totaal behaalde % wordt bijgehouden via een auto-property. Via een methode PrintGraad kan de behaalde graad worden weergegeven. Dit zijn de mogelijkheden:

< 50: niet geslaagd; tussen 50 en 68: voldoende; tussen 68 en 75: onderscheiding; tussen 75 en 85: grote onderscheiding;

85: grootste onderscheiding.

Schrijf een overloaded constructor die ervoor zorgt dat het behaalde percentage zo kan ingesteld worden (Resultaat r= new Resultaat (74)); Test je klasse door enkele objecten in je main aan te maken en de verschillende properties waarden te geven en methoden aan te roepen.

Methoden in klassen

Nummers

Maak een eenvoudige klasse Nummers. Deze klasse bevat 2 getallen (type int). Er zijn 4 methoden:

- Som : geeft som van beide getallen weer
- Verschil : geeft verschil van beide getallen weer
- Product : geeft product van beide getallen weer
- Quotient : geeft deling van beide getallen door. Toon "Error" indien je zou moeten delen door 0.

Toon in je main aan dat je code werkt.

Volgende code zou namelijk onderstaande output moeten geven:

```
Nummers paar1 = new Nummers();
paar1.getal1 = 12;
paar1.getal2 = 34;

Console.WriteLine("Paar: " + paar1.getal1 + ", " + paar1.getal2);
Console.WriteLine("Som = " + paar1.Som());
Console.WriteLine("Verschil = " + paar1.Verschil());
Console.WriteLine("Product = " + paar1.Product());
Console.WriteLine("Quotient = " + paar1.Quotient());
```

Output:

```
Paar: 12, 34
Som = 46
Verschil = -22
Product = 408
Quotient = 0,352941176470588
```

Figuren

Maak een eenvoudige klasse Rechthoek aan die een lengte en breedte als public fields heeft. Maak ook een klasse Driehoek die een basis en hoogte als fields heeft.

Beide klassen hebben een methode `ToonOppervlakte` die de oppervlakte van de figuur in kwestie op het scherm toont.

Toon de werking van het project aan door een aantal instanties van Driehoek en Rechthoek te maken, met verschillende groottes. Roep van iedere figuur de `ToonOppervlakte`-methode aan.

Properties in klassen

PizzaTime

Maak een klasse Pizza. Deze klasse heeft een aantal private fields:

- toppings (string): bevat beschrijving van wat er op ligt, bv. ananas, pepperoni, etc.
- diameter (integer): doorsnede van de pizza in cm
- price (double): prijs van de pizza in euro. Zorg ervoor dat je met behulp van properties deze 3 velden kan uitlezen en aanpassen. Bouw controle in zodat de fields geen foute waarden kunnen gegeven worden (denk maar aan negatieve prijs en diameter, pizza zonder topping, etc.). Maak in je main een aantal pizza-instances aan en toon de werking van de properties aan.

Student Organizer Deluxe

Herschrijf de `Student-klasse` en zorg ervoor dat iedere public field private is. Vervolgens maak je bijhorende public properties aan die toegang tot deze private fields verzorgen. Controleer in de properties op illegale input zodat bijvoorbeeld geen de punten enkel getallen tussen 0 en 20 mogen zijn.

Herschrijf de studentmanager zodat deze werkt met deze nieuwe klasse .

BankManager

Ontwerp een klasse Account die minstens een Naamveld, bedrag en rekeningnummer bevat. Voorzie 3 methoden:

1. WithdrawFunds: bepaald bedrag wordt van rekening verwijderd
2. PayInFunds: bepaald bedrag (als parameter) wordt op de rekening gezet
3. GetBalance: het totale bedrag op de rekening wordt teruggegeven

Pas de WithdrawFunds methode aan zodat als returntype het bedrag (int) wordt teruggegeven. Indien het gevraagde bedrag meer dan de balance is dan geef je al het geld terug dat nog op de rekening staat en toon je in de console dat niet al het geld kan worden gegeven.

Maak 2 instanties van het type Account aan en toon aan dat je geld van de ene account aan de andere kunt geven, als volgt:

```
BankAccount rekening1=new BankAccount();
BankAccount rekening2=new BankAccount();
```

Voeg aan de Account-klasse een private field toe zijnde van het type accountState dat een enumeratie bevat. De account kan in volgende states zijn "Geldig", "Geblokkeerd"). Maak een bijhorende publieke Methode waarmee je de account van state kunt veranderen. Deze methode (noem ze ChangeState) vereist één parameter van het type accountState natuurlijk.

Indien een persoon geld van of naar een Geblokkeerde rekening wil sturen dan zal er een error op het scherm verschijnen. Maak een array aan van 10 klanten. Wanneer je met klassen werkt moet je bij de initialisatie van de array ook ieder element afzonderlijk initialiseren, als volgt:

```
BankAccount[] lijst = new BankAccount[10];
//Init
for(int i=0; i<lijst.Length;i++)
{
    lijst[i]= new BankAccount();
}
```

Schrijf nu een BankManager systeem. Voorzie een console- menu waarbij de gebruiker volgende zaken kan doen:

1. Nieuwe klant aanmaken (max 10)
2. Status van bestaande klant tonen
3. Geld op bepaalde account zetten
4. Geld van bepaalde account afhalen
5. Geld tussen 2 accounts overschrijven.
6. Een totaaloverzicht van alle accounts tonen (Allerlei statistieken zoals de totale som op alle rekeningen samen, rijkste account, etc worden in een tabel getoond)

Voorzie extra functionaliteit naar keuze.

Operator overloading

Breuk

Maak een klasse 'Breuk' dat dus een breuk zal voorstellen met een noemer en teller.

Voorzie properties voor noemer en teller, waarbij de noemer niet 0 mag zijn (zet deze op 1 indien de gebruiker dit toch probeert).

Voeg 2 constructors toe:

- Default constructor: de breuk wordt ingesteld op 0/1
- Overloaded constructor: zowel de noemer als teller worden als parameter meegegeven

Voorts:

1. Voeg een + operator toe die het mogelijk maakt om 2 breuken bij elkaar op te tellen. Wanneer de +operatie is toegepast wordt ook automatisch de Vereenvoudig-methode aangeroepen (zie verder) voor het resultaat wordt teruggegeven. Belangrijk: je dient aan operator overloading te doen. We willen dus dat je bijvoorbeeld kan schrijven Breuk breuksom= breuk1 + breuk2;
2. Voeg voorts een * operator toe die breuken vermenigvuldigen mogelijk maakt (ook hier wordt het resultaat vereenvoudigd teruggegeven).
3. Voeg een methode 'AlsString' toe die de breuk als string teruggeeft, waarbij de breuklijn als slash wordt voorgesteld.

Maak een array van 4 breuken in je main en laat de gebruiker deze alle 4 invullen. Toon vervolgens de som en vermenigvuldiging van deze 4 breuken als strings op het scherm.

Pro: Voeg een methode 'Vereenvoudig' toe. Deze zal de breuk vereenvoudigen indien mogelijk. Als dus de breuk op 2/4 staat dan wordt deze na het uitvoeren van deze methode 1/2.

Pokémon

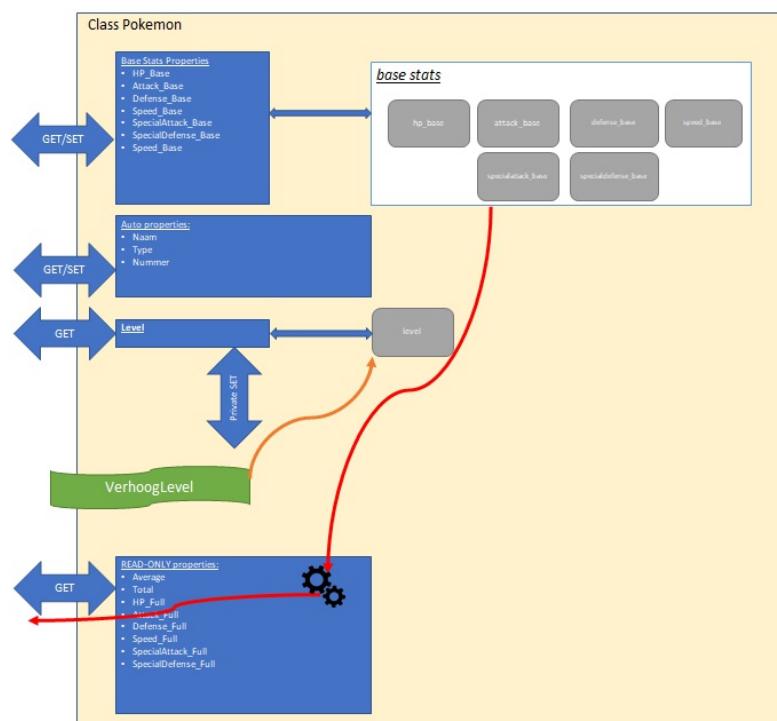
We gaan een programma schrijven dat ons toelaat enkele basis-eigenschappen van specifieke pokémon te berekenen terwijl ze levelen. Nadruk van deze oefening is het juist gebruiken van properties. Bekijk de cheat sheet bij twijfel.

Disclaimer: de informatie in deze tekst is een vereenvoudigde versie van de echte Pokémon-stats in de mate dat ik het allemaal een beetje kan begrijpen en juist interpreteren.

Hoe Pokémons werken

Korte uitleg over Pokémons en hun interne werking: Iedere Pokémon wordt uniek gemaakt door z'n base-stats, deze zijn voor iedere Pokémon anders. Deze base-stats (punt 3) zijn onveranderlijk en blijven dus doorheen het hele leven van een Pokémon dezelfde. Je kan de base-stats als het dna van een Pokémon beschouwen.

De full-stats (punt 9) zijn echter de stats die de effectieve 'krachten' van een Pokémon bepalen in een gevecht. Deze stats worden berekend gebaseerd op de vaste base-stats en het huidige level van de Pokémon. Hoe hoger het level van de Pokémon, hoe hoger dus zijn full-stats.



De Pokémonopdracht

1. Maak een consoleapplicatie.
2. Maak een klasse Pokémon.
3. Maak een klasse Pokemon die je toelaat om de basiseigenschappen (base stats) van een Pokémon te bewaren. Maak voor al deze basis-eigenschappen full properties van het type int: a. HP_Base b. Attack_Base c. Defense_Base d. SpecialAttack_Base e. SpecialDefense_Base f. Speed_Base
4. Voorts wordt een pokémon ook gedefinieerd door z'n naam (string), type (string, bv "grass & poison") en nummer (int), maak hiervoor auto properties aan.
5. Voeg een fullproperty Level(type int). Deze heeft een public get, maar een private setter.
6. Voeg een publieke methode "VerhoogLevel" toe. Deze methode zal, via de private setter van Level (zie vorig punt), de level van de Pokémon met 1 verhogen.
7. Voeg 2 read-only properties toe (enkel get, géén set) genaamd "Average" en "Total". De Average-property geeft het gemiddelde van de 6 basestats terug, dus $(HP_Base+Attack_Base+Defense_Base +SpAttack_Base +SpDefense_Base +Speed_Base)/6$. De Total-property geeft de som terug van de 6 basestats.
8. Voeg een read-only HP_Full property (int) toe om de maximum health voor te stellen. Deze wordt berekend als volgt: $((HP_Base + 50$

-) * Level) / 50) + 10
 Zie hier voor de exacte berekening voor de Pokémon-fans onder ons.
9. Voeg voor iedere basetats een stat_full toe (int). Dus Defense_Full, Speed_Full, etc. Ook deze properties zijn readonly. Deze stats worden berekend als volgt: $(\text{stat_BaseLevel}) / 50 + 5$. Bv: Attack_Full wordt berekend als: $((\text{Attack_BaseLevel}) / 50) + 5$
 10. Kies enkele Pokémon uit [deze lijst](#) en maak in je Main enkele Pokémon-objecten aan met de juiste eigenschappen. Toon aan dat de Average, Total , HP en andere stats correct berekend worden (controleer in de tabel op de voorgaande url).
 11. Maak een kleine loop die je toelaat om per loop een bepaalde pokémon z'n level met 1 te verhogen en vervolgens toon je dan z'n nieuwe stats.
 12. Test eens hoe de stats na bv 100 levels evolueren. Je zal zien dat bepaalde stats pas na een paar keer levelen ook effectief beginnen stijgen.
 13. Voeg extra functionaliteit naar keuze toe

Pokémons en constructors

Pas de kennis van constructors toe op je Pokémon-project. Zorg ervoor dat je Pokémons op 3 kunt aanmaken als volgt:

- Via een default constructor: alle base stats worden daarbij op 10 standaard ingesteld via de constructor
- Via een overloaded constructor die de gebruiker toelaat om de 6 base stats als parameters mee te geven
- Via object initializer syntax waarbij je eerder welke stat kunt instellen.

Deel 2: De Pokémontester

[vergelijk je oplossing uit het vorige deel [met volgende oplossing](#)] Maak een nieuwe console-applicatie genaamd "Pokémon Tester":

- voeg de Pokémon-klasse-bestand toe aan dit project. Verander de "namespace" van dit bestand naar de namespace van je nieuwe console-applicatie (zie "Aanpassen van klasse" in [volgende uitleg](#))
- Voeg een overloaded constructor aan de Pokémon-klasse toe die toelaat dat je pokémons kunt aanmaken door de zes base-stats als parameters mee te geven (bv new Pokémon(45,42,50,65,34,67))
- Schrijf een applicatie die aan de gebruiker eerst de 6 base-stats vraagt. Vervolgens wordt de pokémon aangemaakt met die stats en worden de full-stats aan de gebruiker getoond
- Vraag nu aan de gebruiker tot welke level de pokémon moet gelevelled worden. Roep zoveel keer de LevelUp-methode aan van de Pokémon.
- Toon terug de full-stats van de nu ge-levelde Pokémon

Digitale kluis

Maak een klasse DigitaleKluis die we gaan gebruiken om een kluis voor te stellen.

De klasse heeft volgende elementen:

- Een private variabele die de toegangscode van de kluis bewaard als geheel getal (naam: code)
- Een overloaded constructor die als parameter een geheel getal toelaat. Dit getal zal worden toegewezen aan de private variabele code.
- Een full property "CanShowCode" die kan ingesteld worden op true or false, om aan te geven of de code van buitenuit kan gezien worden.
- Een read-only property "CodeLevel" van type int. Deze property zal de "level" van de code teruggeven. Het level is eenvoudigweg de code gedeeld door 1000 als geheel getal (dus indien de code 500 is zal 0 worden teruggegeven, indien de code 2000 is wordt 2 teruggegeven, etc.)
- Een fullproperty Code met private set. De get van deze property zal -666 teruggeven, tenzij CanShowcode op true staat, in dit geval zal de effectieve code worden terug gegeven.
- Een methode "TryCode" die een geheel getal als parameter aanvaardt. De methode geeft een true terug indien de code correct was, anders false. Deze methode kan gebruikt worden om extern een code te testen , indien deze overeenkomt met de bewaarde code dan zal gemeld worden dat de code geldig is en wordt ook getoond hoeveel keer de gebruiker geprobeerd heeft. Indien de gebruiker -666 meegaf dan meldt de methode dat de gebruiker een cheater is . Indien de gebruiker een foute code meegaf dan meldt de methode dat dit een foute code was en wordt het aantal pogingen met 1 verhoogd.
- Een private variabele "aantalpogingen" om bij te houden hoe vaak de gebruiker geprobeerd heeft de code te vinden. Maak enkele Digitale Kluis objecten aan in je main en test of je bovenstaande klasse correct is geïmplementeerd.

Grotere oefeningen

Studentklasse

Maak ee nieuwe klasse `Student` toe. Deze klasse heeft 6 public fields:

- Naam (string)
- Leeftijd (int)
- Klas (maak dit van een `enum`)
- PuntenCommunicatie (int)
- PuntenProgrammingPrinciples (int)
- PuntenWebTech (int)

Daar deze fields allemaal public zijn kunnen we deze dus rechtstreeks veranderen.

Voeg aan de klasse een methode `BerekenTotaalCijfer` toe. Wanneer deze methode wordt aangeroepen dan wordt het gemiddelde van de 3 punten teruggegeven als double zodat dit op het scherm kan getoond worden.

Voeg aan de klasse ook de methode `GeefOverzicht` toe. Deze methode zal een volledig "Rapport" van de student tonen (inclusief het gemiddelde m.b.v. de `BerekenTotaalCijfer`-methode).

Test je programma door enkele studenten aan te maken en in te stellen. Volgende main zou dan de bijhorende output moeten krijgen:

```
Student student1;
student1.Klas = Klassen.EA2;
student1.Leeftijd = 21;
student1.Naam = "Joske Vermeulen";
student1.PuntenCommunicatie = 12;
student1.PuntenProgrammingPrinciples = 15;
student1.PuntenWebTech = 13;

student1.GeefOverzicht();
```

Output:

```
Joske Vermeulen, 21 jaar
Klas: EA2

Cijferrapport:
*****
Communicatie:      12
Programming Principles: 15
Web Technology:    13
Gemiddelde:        13.3
```

Student Organizer

We gaan nu de Student-klasse gebruiken om een array van studenten te vullen.

Maak daarom een studenten-array aan die 5 studenten bevat :

```
Student[] alleStudenten= new Student[5];
for(int i =0 ; i<5;++)
    alleStudenten[i]= new Student();
```

Initialiseer alle fields van iedere student op een standaard-waarde (mbv een for-loop), bv:

Het programma start op en geeft de gebruiker een menu waaruit kan gekozen worden:

1. Student gegevens invoeren (eerstvolgende student wordt ingevuld) Vervolgens moet de gebruiker kiezen welke student (nummer) moet ingevuld worden, van 1 tot 5. Vervolgens kan de gebruiker de gegevens 1 voor 1 invullen (oude gegevens worden zonder pardon overschreven).
1. Student gegevens tonen (alle studenten) Wanneer de gebruiker voor 2 kiest dan wordt de `GeefOverzicht`-methode aangeroepen van iedere student zodat de 5 'rapportjes' onder elkaar op het scherm

Extra's: Bouw extra functionaliteit naar keuze bij de StudentOrganizer, zoals:

- Vragen aan de gebruiker of de oude gegevens overschreven mogen worden, indien deze reeds ingevuld zijn.
- Inbouwen van een eenvoudige zoekfunctie. Je kan bijvoorbeeld zoeken op naam (exacte invoer) of alle studenten tonen die in een bepaalde klas zitten of wiens punten onder/boven een bepaalde waarde zitten. Je kan dit als extra menuitem inbouwen, waarbij een nieuw menu verschijnt dat de gebruiker de verschillende zoekmogelijkheden voorstelt.
- Verwijderen van een student (waarbij alle gegevens worden gewist)
- Controle in alle methode inbouwen zodat 'lege studenten' worden genegeerd. Wanneer bijvoorbeeld menu item 2 wordt uitgevoerd (alle studenten tonen) dan worden enkel de ingevulde studenten getoond.

APCenture-Job Agency

Deel 1: constructors

Maak een klasse `Job`. Deze klasse heeft vier data fields:

- Description (string) bijvoorbeeld "ruitzen wassen"
- Duration (int), stelt tijd voor die nodig is om job uit te voeren
- RatePerHour (int), stelt kostprijs per uur voor van deze job
- TotalFee (int), stelt totale prijs voor zijnde duration x rateperhour Voorzie properties voor deze 4 velden, echter de TotalFee heeft geen 'set' daar deze een berekening van andere properties is. Telkens de Duration of RatePerHour wordt aangepast (set) wordt de TotalFee herberekend (je zal dus een private totalFee nodig hebben waar de public property TotalFee z'n waarde van krijgt).

Voorzie 2 constructors:

- Default constructor: stelt de description in op "onbekend" en zet duration en rateperhour in op 0.
- Overloaded constructor: waarbij je de 3 velden (behalve TotalFee) kan aanpassen tijdens de constructie van een Job-object

Toon de werking van je klasse aan door enkele objecten aan te maken met zowel de default als de overloaded constructor. Toon vervolgens dat TotalFee correct werkt.

Deel 2: Operator Overloading

Pas de klasse 'Job' aan zodat de + operator kan gebruikt worden om 2 job-objecten bij elkaar op te tellen. Bv:

```
Job epicDuoJob= jobOne+jobTwo; //jobOne en jobTwo zijn ook van het type Job
```

De som van 2 job-objecten gaat als volgt te werk:

- Description: beide description worden na elkaar geplakt, waarbij het voegwoord 'en' tussen beide wordt gezet.
- Duration: som van beide durations
- RatePerHour: gemiddelde de `rateperhour` van beide objecten Toon in je main aan dat je nieuwe klasse werkt en dat je 2 jobs kan samenvoegen. Toon ook aan dat je vervolgens deze nieuwe samenvoeging op zijn beurt kan samenvoegen met een andere job (of zelfs met een andere samengevoegde job!).

Geheugenmanagement in C

Tot nog toe lagen we er niet van wakker wat er achter de schermen van een C# programma gebeurde. Nu dat we arrays hebben geïntroduceerd wordt het tijd om dit wel te doen. Het is, zeker naar het volgende semester, essentieel dat je deze materie ten gronde beheerst.

Twee soorten geheugen

Wanneer een C# applicatie wordt uitgevoerd krijgt het twee soorten geheugen toegewezen dat het 'naar hartelust' kan gebruiken:

1. Het kleine, maar snelle **stack** geheugen
2. Het grote, maar tragere **heap** geheugen

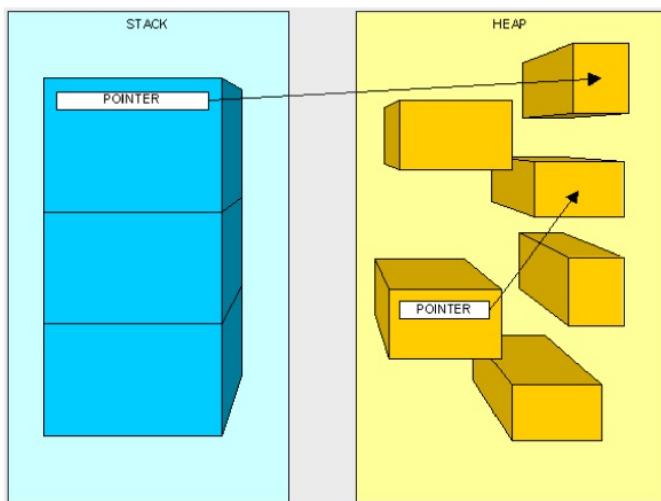
Afhankelijk van het soort variabele wordt ofwel de stack, ofwel de heap gebruikt. **Het is uitermate belangrijk dat je weet in welk geheugen de variabele zal bewaard worden!**

Er zijn namelijk twee soorten variabelen:

1. Value types
2. Reference types

Als je volgende tabel begrijpt dan beheers je geheugenmanagement in C#:

| | Value types | Reference types |
|-------------------------|----------------------------|---|
| Inhoud van de variabele | De eigenlijke data | Een referentie naar de eigenlijke data |
| Locatie | (Data) Stack | Heap (globaal)geheugen |
| Beginwaarde | 0, 0.0, "", false, etc. | null |
| Effect van = operator | Kopieert de actuele waarde | Kopieert het adres naar de actuele waarde |



Waarom twee geheugens?

Waarom plaatsen we niet alles in de stack? De reden hiervoor is dat bij het compileren van je applicatie er reeds zal berekend worden hoeveel geheugen de stack zal nodig hebben. Wanneer je programma dus later wordt uitgevoerd weet het OS perfect hoeveel geheugen het minstens moet reserveren. Er is echter een probleem: we kunnen niet alles perfect berekenen/voorspellen. Een variabele van het type `int` is perfect geweten hoe groot die zal zijn (32 bit). Maar wat met een string? Of met een array waarvan we pas tijdens de uitvoer de lengte aan de gebruiker misschien vragen? Het zou nutteloos (en zonde) zijn om reeds bij aanvang een bepaalde hoeveelheid voor een array te reserveren als we niet ween hoe groot die zal worden. Beeld je maar eens in dat we 2k byte reserveren om dan te ontdekken dat we maar 5byte ervan nodig hebben. RAM is goedkoop, maar toch... De heap laat ons dus toe om geheugen op een wat minder gestructureerde manier in te palmen. Tijdens de uitvoer van het programma zal de heap als het ware dienst doen als een grote zandbak waar eender welke plek kan ingepalmd worden om zaken te bewaren. De stack daarentegen is het kleine bankje naast de zandbak: handig, snel, en perfect geweten hoe groot.

Value types

Value types worden in de stack bewaard. De effectieve waarde van de variabele wordt in de stack bewaard. Dit zijn alle gekende, 'eenvoudige' datatypes die we tot nog toe gezien hebben, inclusief enums en structs (zie later):

- `sbyte` , `byte`
- `short` , `ushort`
- `int` , `uint`
- `long` , `ulong`
- `char`
- `float` , `double` , `decimal`
- `bool`
- `structs`
- `enums`

= operator bij value types

Wanneer we een value-type willen kopiëren dan kopiëren de echte waarde:

```
int getal=3;
int anderGetal= getal;
```

Vanaf nu zal `anderGetal` de waarde `3` hebben. Als we nu een van beide variabelen aanpassen dan zal dit **geen** effect hebben op de andere variabelen.

We zien hetzelfde effect wanneer we een methode maken die een parameter van het value type aanvaardt - we geven een kopie van de variabele mee:

```
void DoeIets(int a)
{
    a++;
    Console.WriteLine($"In methode {a}");
}

// Elders:
int getal= 5;
DoeIets(getal);
Console.WriteLine($"Na methode {getal}");
```

De parameter `a` zal de waarde 5 gekopieerd krijgen. Maar wanneer we nu zaken aanpassen in `a` zal dit geen effect hebben op de waarde van `getal`. De output van bovenstaand programma zal zijn:

```
In methode 6
Na methode 5
```

Reference types

Reference types worden in de heap bewaard. De effectieve waarde wordt in de heap bewaard, en in de stack zal enkel een **referentie** of **pointer** naar de data in de heap bewaard worden. Een referentie (of pointer) is niet meer dan het geheugenadres naar waar verwezen wordt (bv `0xA3B3163`). Concreet zijn dit alle zaken die vaak redelijk groot zullen zijn:

- objecten, interfaces en delegates
- arrays

= operator bij reference types

Wanneer we de = operator gebruiken bij een reference type dan kopiëren we de referentie naar de waarde, niet de waarde zelf.

Bij objecten We zien dit gedrag bij alle reference types, zoals objecten:

```
Student stud= new Student();
```

Wat gebeurt er hier?

1. `new Student()` : `new` roept de constructor van `Student` aan. Deze zal een constructor in de **heap** aanmaken en vervolgens de

geheugenlocatie teruggeven.

2. Een variabele `stud` wordt in de **stack** aangemaakt.
3. De geheugenlocatie uit de eerste stap wordt vervolgens in `stud` opgeslagen in de stack.

Bij arrays Maar ook bij arrays:

```
int[] nummers= {4,5,10};  
int[] andereNummers= nummers;
```

In dit voorbeeld zal `andereNummers` dus nu ook verwijzen naar de array in de heap waar de actuele waarden staan.

Als we dus volgende code uitvoeren dan ontdekken we dat beide variabele naar dezelfde array verwijzen:

```
andereNummers[0]=999;  
Console.WriteLine(andereNummers[0]);  
Console.WriteLine(numbers[0]);
```

We zullen dus als output krijgen:

```
999  
999
```

Hetzelfde gedrag zien we bij objecten:

```
Student a= new Student("Abba");  
Student b= new Student("Queen");  
a=b;  
Console.WriteLine(a.Naam);
```

We zullen in dit geval dus `Queen` op het scherm zien omdat zowel `b` als `a` naar hetzelfde object in de heap verwijzen. Het originele "abba"-object zijn we kwijt en zal verdwijnen (zie Garbage collector verderop).

Methoden en reference parameters

Ook bij methoden geven we de dus de referentie naar de waarde mee. In de methode kunnen we dus zaken aanpassen van de parameter en dan passen we eigenlijk de originele variabele aan:

```
void DoeIets(int[] a)  
{  
    a[0]++;  
    Console.WriteLine($"In methode {a[0]}");  
}  
  
//Elders:  
int[] getallen= {5,3,2};  
DoeIets(getallen);  
Console.WriteLine($"Na methode {getallen[0]}");
```

We krijgen als uitvoer:

```
In methode 6  
Na methode 6
```

Opgelet: Wanneer we een methode hebben die een value type aanvaardt en we geven één element van de array met dan geven dus een kopie van de actuele waarde mee!

```
void DoeIets(int a)  
{  
    a++;  
    Console.WriteLine($"In methode {a}");  
}  
  
//Elders:  
int[] getallen= {5,3,2};  
DoeIets(getallen[0]); //<= VALUE TYPE!  
Console.WriteLine($"Na methode {getallen[0]}");
```

De output bewijst dit:

```
In methode 6  
Na methode 5
```

De Garbage Collector (GC)

Een essentieel onderdeel van .NET is de zogenaamde GC, de Garbage Collector. Dit is een geautomatiseerd onderdeel van ieder C# programma dat ervoor zorgt dat we geen geheugen voor niets gereserveerd houden. De GC zal geregeld het geheugen doorlopen en kijken of er in de heap data staat waar geen references naar verwijzen. Indien er geen references naar wijzen zal dit stuk data verwijderd worden.

In dit voorbeeld zien we dit in actie:

```
int[] array1= {1,2,3};  
int[] array2= {3,4,5};  
array2=array;
```

Vanaf de laatste lijn zal er geen referentie meer naar `{3,4,5}` in de heap zijn, daar we deze hebben overschreven met een referentie naar `{1,2,3}`. De GC zal dus deze data verwijderen.

Wil je dat niet dan zal je dus minstens 1 variabele moeten hebben dat naar de data verwijst. Volgende voorbeeld toont dit:

```
int[] array1= {1,2,3};  
int[] array2= {3,4,5};  
int[] bewaarArray= array2;  
array2=array;
```

De variabele `bewaarArray` houdt dus een referentie naar `{3,4,5}` bij en we kunnen dus later via deze variabele alsnog aan de originele data.

Meer weten?

Meer info, lees zeker volgende artikels:

- [Reference en value types](#)
- [Stack vs heap](#)

TODO

Bookmark Manager

Maak een "bookmark manager". Deze tool zal in de console aan de gebruiker 5 favoriete sites vragen: naam en url. Vervolgens zal de tool alle sites in een lijst tonen met een nummer voor. De gebruiker kan dan de nummer intypen en de tool zal automatisch de site in de browser openen.

Je opdracht:

1. Maak een array waarin je 5 bookmark objecten kan plaatsen.
2. Vul de applicatie aan zodat de gebruiker: een bestaand bookmark kan verwijderen en een bestaand bookmark kan aanpassen

Enkele zaken die je nodig hebt:

BookMark klasse:

```
class BookMark
{
    public string Naam { get; set; }
    public string URL { get; set; }
    public void OpenSite()
    {
        Process.Start("Iexplore.exe", URL);
    }
}
```

Voorbeeld van hoe de bookmark klasse zal werken:

```
BookMark u = new BookMark();
u.Naam = "Windows";
u.URL = "www.google.be";
u.OpenSite();
```

Overerving

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

De child-klasse kan alles wat de parent-klasse kan, maar de nieuwe klasse kan nu ook extra specialisatie code krijgen.

Is-een relatie

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

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

(Opgelet: wanneer we "x heeft een y" zeggen gaat het **niet** over overerving, maar over compositie)

Inheritance in C

Overerving duid je aan met behulp van het dubbele punt(:) bij de klassedefinitie:

Een voorbeeld:

```
class Paard: Dier
{
    public bool KanHinniken{get;set;}
}

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

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

```
Dier aDier= new Dier();
Paard bPaard= new Paard();
aDier.Eet();
bPaard.Eet();
bPaard.KanHinniken=false;
aDier.KanHinniken=false; //!!! zal niet werken!
```

Multiple inheritance

In C# is het niet mogelijk om een klasse van meer dan een parent-klasse te laten overerven (zogenaamde multiple inheritance), wat wel mogelijk is in sommige andere object georiënteerde talen.

Transitive

Overerving in C# is transitief, dit wil zeggen dat de child-klasse ALLES overerft van de parent-klasse: methoden, properties, etc.

Protected

Ook al is overerving transitief, hou er rekening mee dat private variabelen en methoden van de parent-klasse NIET rechtsreeks aanroepbaar zijn in de child-klasse. Private geeft aan dat het element enkel in de klasse zichtbaar is:

```
class Paard: Dier
{
    public void MaakOuder()
    {
        leeftijd++; // !!! dit zal error geven!
    }
}

class Dier
{
    private int leeftijd;
}
```

Je kan dit oplossen door de **protected** access modifier ipv private te gebruiken. Met protected geef je aan dat het element enkel zichtbaar is binnen de klasse **en** binnen child-klassen:

```
class Paard: Dier
{
    public void MaakOuder()
    {
        leeftijd++; // werkt nu wel
    }
}

class Dier
{
    protected int leeftijd;
}
```

Virtual en Override

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

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

class Raket: Vliegtuig
{}
```

Stel dat we 2 objecten aanmaken en laten vliegen:

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

De uitvoer zal dan zijn:

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

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

```
class Raket:Vliegtuig
{
    public override void Vlieg()
    {
        Console.WriteLine("De raket verdwijnt in de ruimte.");
    }
}
```

De uitvoer van volgende code zal nu anders zijn:

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

Uitvoer:

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

Base keyword

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

Stel dat we volgende 2 klassen hebben:

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

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

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

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

Constructors bij overerving

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

- Eerst wordt de constructor aangeroepen van de basis-klasse: dus steeds eerst die van `System.Object`
- Gevolgd door de constructors van alle parent-klassen
- Finaal de constructor van de klasse zelf.

Volgende voorbeeld toont dit in actie:

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

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

Indien je vervolgens een object aanmaakt van het type Medic:

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

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

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

Er wordt dus verondersteld in dit geval dat er een default constructor in de basis-klasse aanwezig is.

Overloaded constructors

Indien je klasje Soldier een overloaded constructor heeft, dan geeft deze niet automatisch een default constructor. Volgende code zou dus een probleem geven indien je een Medic wilt aanmaken via `new Medic()`:

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

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

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

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

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

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

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

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

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

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

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

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

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

System.Object

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

Because every class descends from `Object`, every object "is an" `Object`.

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

Hoe ziet `System.Object` er uit?

Wanneer je een lege klasse maakt dan zal je zien dat instanties van deze klasse reeds 4 methoden ingebouwd hebben, dit zijn uiteraard de methoden die in de `System.Object` klasse staan gedefinieerd:

| Methode | Beschrijving |
|----------------------------|--|
| <code>Equals()</code> | Gebruikt om te ontdekken of twee instanties gelijk zijn. |
| <code>GetHashCode()</code> | Geeft een unieke code (hash) terug van het object; nuttig om o.a. te sorteren. |
| <code>GetType()</code> | Geeft het type (of klasse) van het object terug. |
| <code>ToPrint()</code> | Geeft een string terug die het object voorstelt. |

GetType()

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

```
Student stud1= new Student();
Console.WriteLine(stud1.GetType());
```

Dit zal als uitvoer de namespace gevuld door het type op het scherm geven. Als je project bijvoorbeeld "StudentManager" heet (en je namespace dus ook) dan zal er op het scherm verschijnen: `StudentManager.Student` .

ToString()

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

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

Wordt je code eigenlijk herschreven naar:

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

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

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

Merk twee zaken op:

1. `GetType` wordt aangeroepen en die output krijg je terug.
2. De methode is `virtual` gedefinieerd. **Alle 4 methoden in `System.Object` zijn `virtual`, en je kan deze dus `override` 'n!**

ToString() overiden

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

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

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

Wanneer je nu `Console.WriteLine(stud1);` (gelet dat hij een Naam en Leeftijd heeft) zou schrijven dan wordt je output: `Student Tim Dams (Leeftijd:35)`.

Equals()

Ook deze methode kan je dus overiden om twee objecten met elkaar te testen. Op het [einde van deze cursus](#) zal dieper in `Equals` ingaan worden om objecten te vergelijken, maar we tonen hier reeds een voorbeeld:

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

De `Equals` methode heeft dus als signatuur: `public virtual bool Equals(Object o)`. Twee objecten zijn gelijk voor .NET als aan volgende afspraken wordt voldaan:

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

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

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

Equals overiden

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

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

De lijn `Student temp = (Student)o;` zal het object `o` casten naar een `Student`. Doe je dit niet dan kan je niet aan de interne `Student`-variabelen van het object `o`.

Dit concept heet [polymorfisme](#) en wordt later uitgelegd.

GetHashCode

Indien je Equals override dan moet je eigenlijk ook GetHashCode overriden, daar er wordt verondersteld dat twee gelijke objecten ook dezelfde unieke hashcode teruggeven. Wil je dit dus implementeren dan zal je dus een (bestaand) algoritme moeten schrijven dat een uniek nummer genereert voor ieder niet-gelijk object.

Bekijk volgende [StackOverflow post](#) indien je dit wenst toe te passen.

Ik ben nog niet helemaal mee?

Niet getreurd, je bent niet de enige. Overerving, System.Object, Equals,... het is allemaal een hoop nieuwe kennis om te verwerken. Aan het [einde van deze cursus](#) gaan we dieper in bovenstaande materie in om een volledige Equals methode op te bouwen en we bij iedere stap uitgebreide uitleg geven.

Abstract

Abstracte klassen

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

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

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

Een voorbeeld:

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

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

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

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

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

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

En als we polymorfisme gebruiken ([zie verder](#)) dan mag dit ook: `Dier paardje= new Paard();`

Abstracte methoden

Het is logisch dat we mogelijk ook bepaalde zaken in de abstracte klasse als abstract kunnen aanduiden. Beeld je in dat je een Methode "MaakGeluid" hebt in je klasse Dier. Wat voor een geluid maakt 'een dier'? We kunnen dus ook geen implementatie (code) geven in de abstracte parent klasse.

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

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

Merk op dat er geen accolades na de signature komen.

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

De Paard-klasse wordt dan:

```
class Paard: Dier
{
    public override string MaakGeluid()
    {
        return "Hinnikhinnik";
    }
}
```

(en idem voor de wolf-klasse uiteraard)

Abstracte methoden enkel in abstracte klassen

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

Book

Deel 1

Maak een klasse `Book` en gebruik auto-properties voor de velden:

- ISBN (int)
- Title (string)
- Author (string) Maak voorts een property voor Price, met bijhorende private price field.
- Price (double) Maak een child-klasse die van Book overerft genaamd 'TextBook'. Een textbook heeft één extra field:
- GradeLevel (int) Maak een child-klasse die van Book overerft genaamd 'CoffeeTableBook'. Deze klasse heeft geen extra velden.

Voorts kunnen boeken "opgeteld" worden om als omnibus uitgebracht te worden. De titel wordt dan "Omnibus van [X]". waarbij X de Authors bevat, gescheiden met een komma. De prijs van een Omnibus is steeds de som van beide boeken gedeeld door 2.

In beide child-klassen, override de Price-setter zodat a) Bij Textbook de prijs enkel tussen 20 en 80 kan liggen b) Bij CoffeeTableBooks de prijs enkel tussen 35 en 100 kan liggen

Deel 2

- Zorg ervoor dat boeken de ToString overiden zodat je boekobjecten eenvoudig via Console.WriteLine(myBoek) hun info op het scherm tonen. Ze tonen deze info als volgt: "Title - Auteur (ISBN) Price" (bv *The Shining - Stephen King (05848152) 50*)
- Zorg ervoor dat de equals methode werkt op alle boeken. Boeken zijn gelijk indien ze hetzelfde ISBN nummer hebben

Toon de werking aan van je 3 klassen: Maak boeken aan van de 3 klassen, toon dat de prijs niet altijd zomaar ingesteld kan worden en toon aan dat je Equals –methode werkt (ook wanneer je bijvoorbeeld een Book en TextBook wil vergelijken).

Money, money, money

Maak enkele klassen die een bank kan gebruiken.

1. Abstracte klasse `Rekening` : deze bevat een methode `voegGeldToe` en `haalGeldAf`. Het saldo van de rekening wordt in een private variabele bijgehouden (en via de voorgaande methoden aangepast) die enkel via een read-only property kan uitgelezen worden. Voorts is er een abstracte methode `BerekenRente` de rente als double teruggeeft.
2. Een klasse `BankRekening` die een Rekening is. De rente van een BankRekening is 5% wanneer het saldo hoger is dan 100 euro, zoniet is deze 0%.
3. Een klasse `SpaarRekening` die een Rekening is. De rente van een SpaarRekening bedraagt steeds 2%.
4. Een klasse `ProRekening` die een SpaarRekening is. De ProRekening hanteert de Rente-berekening van een SpaarRekening (`base.BerekenRente`) maar zal per 1000 euro saldo nog eens 10 euro verhogen.

Schrijf deze klasse en toon de werking ervan in je main.

Geometric figures

Maak een abstracte klasse `GeometricFigure` . Iedere figuur heeft een hoogte, breedte en oppervlakte. Maak properties voor deze 3 fields. De oppervlakte is read-only want deze wordt berekend gebaseerd op de hoogte en breedte .

Voorzie een abstracte methode "BerekenOppervlakte" die een int teruggeeft. Maak 3 klassen:

- Rechthoek: erft over van GeometricFigure. Oppervlakte is gedefinieerd als breedte * hoogte.
- Vierkant: erft over van Rechthoek. Voorzie een constructor die lengte en breedte als parameter aanvaardt: deze moeten gelijk zijn, indien niet zet je deze tijdens de constructie gelijk. Voorzie een 2e constructor die één parameter aanvaardt dat dan geldt als zowel de lengte als breedte. Deze klasse gebruikt de methode BerekenOppervlakte van de Rechthoek-klasse.
- Driehoek: erft over van GeometricFigure. Oppervlakte is gedefinieerd als breedte*hoogte/2.

Maak een applicatie waarin je de werking van deze klassen aantoon

Dierentuin

Maak een console-applicatie waarin je een zelfverzonnen abstract klasse Dier in een List kunt plaatsen. Ieder dier heeft een gewicht en een methode `Zegt` (die abstract is) die het geluid van het dier in kwestie op het scherm zal tonen. Maak enkele childklassen die overerven van Dier en uiteraard de `Zegt` methode overriden.

Vervolgens vraag je aan de gebruiker wat voor dieren er in deze lijst moeten toegevoegd worden. Wanneer de gebruiker 'q' kiest stopt het programma met vragen welke dieren moeten toegevoegd worden en komt er een nieuw keuze menu. Het keuze menu heeft volgende opties: a. Dier verwijderen , gevolgd door de gebruiker die invoert het hoeveelste dier weg moet uit de List. b. Diergewicht gemiddelde: het gemiddelde van alle dieren hun gewicht wordt getoond c. Dier praten: alle dieren hun `Zegt()` methode wordt aangeroepen en via `WriteLine` getoond d. Opnieuw beginnen: de List wordt leeggemaakt en het programma zal terug van voor af aan beginnen.

Probeer zo modulair (methoden en oo!) te werken.

Compositie

We spreken over compositie (**composition**) wanneer we een object in een ander object gebruiken. Denk bijvoorbeeld aan een object van het type motor dat je gebruikt in een object van het type auto.

Heeft een-relatie

Wanneer twee objecten een "heeft een"-relatie hebben dan spreken we over compositie:

- Een auto heeft een motor
- Een computer heeft een harde schijf

Het lijdende voorwerp zal steeds het object zijn dat binnen het onderwerp zal geplaatst worden (motor in auto, schijf in computer).

In de praktijk

We bekijken het voorbeeld van de computer en de harde schijf. We hebben twee klassen

```
class PC
{
}

class Disk
{
}
```

Een PC heeft een Disk, dit wil zeggen dat we in de klasse PC een object van het type Disk zullen definiëren:

```
class PC
{
    private Disk cDisk;
}
```

Uiteraard moeten we deze Disk nog instantiëren, dit kan op verschillende mogelijkheden en is afhankelijk van wat je nodig hebt. We tonen er drie, maar er zijn er nog.

Manier 1

Ogenblikkelijk:

```
class PC
{
    private Disk cDisk=new Disk();
}
```

Manier 2

Ogenblikkelijk, maar in constructor:

```
class PC
{
    public PC()
    {
        cDisk= new Disk();
    }
    private Disk cDisk;
}
```

Manier 3

Door een extern object, bv via een property:

```

class PC
{
    public Disk CDisk
    {
        get{return CDisk;}
        set{CDisk= value;}
    }
    private Disk CDisk;
}

// Elders dan:
myPC.CDisk= new Disk();

```

Kortom

Alle manieren die ja al kende om met bestaande types objecten aan te maken gelden nog steeds. Compositie deed je al de hele tijd wanneer je bijvoorbeeld zij "een student heeft een leeftijd" en dan een variabele `int age` aanmaakte. Het grote verschil is echter dat objecten moeten geïnstantieerd worden , wat niet moest met value-types.

NullReference is een klassieke fout

Een veelvoorkomende fout bij compositie van objecten is dat je een object aanspreekt dat nooit werd geïnstantieerd. Je krijgt dan een `NullReferenceException`.

"Heeft meerdere"- relatie

Wanneer een object meerdere objecten van een specifiek type heeft (denk maar aan "een boek heeft meerdere pagina's" of "een boom heeft bladeren") dan zullen we een array of een list als compositie-object gebruiken.

Voorbeeld:

```

class Page{}

class Boek
{
    private Page[] allPages= new Page[100];
}

```

Indien je nu een pagina wenst toe te voegen dan moet je ook deze individuele array-elementen nog instantiëren. Een voorbeeld waarbij men van buitenuit het object bestaande pagina's kan toevoegen:

```

class Book
{
    public InsertPage(Page toAdd, int Position)
    {
        allPages[Position]= toAdd
    }

    private Page[] allPages= new Page[100];
}

//Elders
Book myBook= new Book();
Page myThirdPage= new Page();
myBook.InsertPage(myThirdPage, 2);

```

Of een voorbeeld met List:

```

class Book
{
    public List<Page> AllPages{get;set;} = new List<Page>();

}

//Elders
myBook.AllPages.Insert(new Page(), 5);

```


Polymorfisme

Polymorfisme oftewel "meerderen vormen" is een programmeertechniek waarbij je code schrijft die door objecten van verschillende klassen kan gebruikt worden. Hierdoor kan je dus compactere code schrijven en hoeft je niet voor iedere klasse apart (deel)code te schrijven. Samen met encapsulatie en overerving is polymorfisme een derde belangrijke eigenschap van object georiënteerd programmeren.

We tonen het nut van polymorfisme aan de hand van drie voorbeelden:

Polymorfisme in de praktijk: Dieren

Een voorbeeld maakt veel duidelijk. Stel dat we een aantal Dier-gerelateerde klassen hebben die allemaal op hun eigen manier een geluid voortbrengen. We hanteren de klasse dier uit een eerder hoofdstuk ([abstracte klassen in overerving](#)):

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

Twee child-klassen:

```
class Paard: Dier
{
    public override string MaakGeluid()
    {
        return "Hinnikhinnik";
    }
}

class Varken: Dier
{
    public override string MaakGeluid()
    {
        return "Oinkoink";
    }
}
```

Dankzij polymorfisme kunnen we nu elders objecten van Paard en Varken in een `Dier` bewaren, maar toch hun eigen geluid laten reproduceren:

```
Dier someAnimal = new Varken();
Dier anotherAnimal = new Paard();
Console.WriteLine(someAnimal.MaakGeluid()); //Hinnikhinnik
Console.WriteLine(anotherAnimal.MaakGeluid()); //Oinkoink
```

Polymorfisme in de praktijk: Presidenten



Beeld je in dat je een klasse President

hebt met een methode "RunTheCountry" (voorbeeld van [StackOverflow](#)). De President heeft toegang tot tal van adviseurs die hem kunnen helpen (inzake militair, binnenlands beleid, economie). Zonder de voordelen van polymorfisme zou de klasse President er zo kunnen uitzien, **slechte manier**:

```
public class President
{
    public void RunTheCountry()
    {
        // people walk into the Presidents office and he tells them what to do
        // depending on who they are.

        // Fallujah Advice - Mr Prez tells his military exactly what to do.
        Petraeus.IncreaseTroopNumbers();
        Petraeus.ImproveSecurity();
        Petraeus.PayContractors();

        // Condi diplomacy advice - Prez tells Condi how to negotiate

        Condi.StallNegotiations();
        Condi.LowBallFigure();
        Condi.Fire DemocraticallyElected IraqiLeader BecauseIDontLikeHim();

        // Health care mr X

        MrX.IncreasePremiums();
        MrX.AddPreexistingConditions();
    }
}
```

Je merkt dat de President (of de programmeur van deze klasse) aardig wat specifieke kennis moet hebben van de vele verschillende departementen van het land. Uiteraard is dat onmogelijk (een fictief voorbeeld: stel je Trump voor...Denk je echt dat die zo veel weet?). Bovenstaande code is dus zeer slecht. Telkens er zaken binnen een specifiek landsonderdeel wijzigen moet dit ook in de klasse President aangepast worden.

Dankzij polymorfisme kunnen we dit alles veel mooier oplossen:

1. We verplichten alle adviseurs dat ze overerven van de abstracte klasse `Advisor` die maar 1 abstracte methode heeft `Advise`:

```
abstract class Advisor
{
    abstract public void Advise();
}

class MilitaryMinistor:Advisor
{
    public override void Advise()
    {
        increaseTroopNumbers();
        improveSecurity();
        payContractors();
    }

    private void increaseTroopNumbers()
```

```

{
//...
}
private void improveSecurity()
{
//...
}
private void improveSecurity()
{
//...
}
}

class ForeignSecretary:Advisor
{
//...
}
class HealthOfficial:Advisor
{
//...
}

```

2° Het leven van de President wordt plots véél makkelijker:

```

public class MisterPresident
{
    public void RunTheCountry()
    {
        Advisor Petraeus = new MilitaryAdvisor();
        Advisor Condi = new ForeignSecretary();
        Advisor mrX= new HealthOfficial();
        Petraeus.Advise(); // # Petraeus says send 100,000 troops to Fallujah
        Condi.Advise(); // # she says negotiate trade deal with Iran
        mrX.Advise(); // # they say we need to spend $50 billion on ObamaCare
    }
}

```

3° En we kunnen hem nog helpen door met een array of `List<Advisor>` te werken zodat hij ook niet steeds de "namen" van z'n adviseurs moet kennen:

```

public class MisterPresident
{
    public void RunTheCountry()
    {

        List<Advisor> allMinisters= new List<Advisor>();
        allMinisters.Add(new MilitaryAdvisor());
        allMinisters.Add(new ForeignSecretary());
        allMinisters.Add(new HealthOfficial());

        //Ask advise from each:
        foreach (Advisor minister in allMinisters)
        {
            minister.Advise();
        }
    }
}

```

En wie zei dat het presidentsschap moeilijk was?!

Nog voorbeelden van polymorfisme nodig?

Volgende tekst heeft een leuke insteek om polymorfisme uit te leggen... aan de hand van...wait for it... Zeemeerminnen! :) [Lezen maar!](#)

Volgende voorbeeld is iets praktischer: [Arena with a mage in C# .NET \(inheritance and polymorphism\)](#)

Why should I care?



Polymorfisme is een heel krachtig concept. Door objecten te bewaren in hun basistype en , wanneer nodig, ze als 'zichzelf' te gebruiken wordt je code een pak eenvoudiger. Vaak weet je niet op voorhand wat voor elementen je in je lijst wilt plaatsen. Via polymorfisme lossen we dit op. Stel bijvoorbeeld dat je een lijst van Personen hebt (`List<Person>`) waar echter elementen van subklassen in terecht kunnen komen (`Bakker` , `Student` , etc) , dan laat polymorfisme dit gewoon toe om ook deze elementen in die lijst te plaatsen.

Interfaces

Delen van deze sectie komen uit het voortreffelijke handboek "Programmeren in C#" (2nd Edition door Douglas Bell en Mike Parr, ISBN:9789043032421)

In C# worden interfaces gebruikt om de uitwendige verschijningsvorm van een klasse te beschrijven. Vaak zijn klassen grote, complexe stukken code bestaande uit tientallen tot honderden methoden en properties. Wanneer we een dergelijke klasse moeten gebruiken boeit meestal enkel de uitwendige, of publieke, vorm: dit zijn de enige zaken waar je als externe gebruiker van de klasse mee kunt praten. Interfaces zijn als het ware stempels die we op een klasse kunnen plakken om zo te zeggen " deze klasse gebruikt interface xyz". Gebruikers van de klasse hoeven dan niet de hele klasse uit te spitten en weten dat alle klassen met interface xyz dezelfde publieke properties en methoden hebben.

Merk op dat dit niets met grafische user interfaces te maken heeft in dit geval.

Een interface is niet meer dan een belofte: het zegt enkel welke publieke methoden en properties de klassen bezit. Het zegt echter niets over de effectieve code/implementatie van deze methoden en properties.

Volgende code toont hoe we een interface definiëren:

```
interface ISuperHeld
{
    void SchietLasers();
    int VerlaagKracht(bool isZwak);
    int Power{get;set;}
}
```

Enkele opmerkingen:

- Het woord `class` wordt niet gebruikt, in de plaats daarvan gebruiken we `interface`.
- Het is een goede gewoonte om interfaces met een I te laten starten in hun naamgeving
- Methoden en properties gaan niet vooraf van `public`: interfaces zijn van nature net publiek, dus alle methoden en properties van de interface zijn dat bijgevolg ook.
- Er wordt geen code/implementatie gegeven: iedere methode eindigt ogenblikkelijk met een komma punt.

Het is in de klassen dat we nu vervolgens verplicht zijn deze methode en properties te implementeren.

Een interface is een beschrijving hoe een component een andere component kan gebruiken, zonder te zeggen hoe dit moet gebeuren. De interface is met andere woorden 100% scheiding tussen de methode/Property-signatuur en de eigenlijke implementatie ervan.



Interface regels

- Je kan geen membervariabelen (fields) declareren in een interface (dat hoort bij de implementatie)
- Je kan geen constructor declareren
- Je kan geen access specificeren (public, protected, etc): alles is public
- Je kan nieuwe types (bv enum, struct) in een interface declareren.
- Een interface kan niet overerven van een klasse, wel van een andere interface.

Interfaces en klassen

We kunnen nu aan klassen de stempel `ISuperHeld` geven zodat programmeurs weten dat die klasse gegarandeert de methoden `SchietLasers`, `VerlaagKracht` en de property `Power` zal hebben.

Volgende code toont dit:

```
class Zorro: ISuperHeld
{
    public void RoepPaard(){...}
    public bool HeeftSnor{get;set;}
    public void SchietLasers() //interface ISuperHeld
    {
        Console.WriteLine("pewpew");
    }
    int VerlaagKracht(bool isZwak)//interface ISuperHeld
    {
        if(isZwak) return 5;
        return 10;
    }
    int Power{get;set;} //interface ISuperHeld
}
```

Zolang de klasse Zorro niet exact de interface inhoud implementeert zal deze klasse niet gecompileerd kunnen worden.

Meerder interfaces

Een nadeel van overerving is dat een klasse maar van 1 klasse kan overerven. Een klasse mag echter wel meerdere interfaces met zich meedragen:

```
interface ISuperHeld{...}
interface ICoureur{...}
class Man {...}

class Zorro:Man, ISuperHeld
{...}

class Batman:Man, ISuperHeld, ICoureur
{...}
```

Ook mogen interfaces van elkaar overerven:

```
interface IGod:ISuperHeld
{...}
```

Voorbeeld: Presidenten en interfaces

In het hoofdstuk Polymorfisme bespraken we een voorbeeld van een klasse `President` die enkele `Advisor`-klassen gebruikt om hem te helpen ([lees hier na](#)).

Een nadeel van die voorgaande aanpak is dat al onze Advisors maar 1 "job" kunnen hebben: ze erven allemaal over van `Advisor` en kunnen nergens anders van overerven (geen multiple inheritance is toegestaan in C#). Via interfaces kunnen we dit oplossen. Een advisor gaan we dan eerder als een "bij-job" beschouwen en niet de hoofdreden van een klasse.

We definiëren daarom eerst een nieuwe interface `IAdvisor`:

```
interface IAdvisor
```

```
{
    void Advise();
}
```

Vanaf nu kan eender *wie* die deze interface implementeert de President advies geven. Hoera! En daarnaast kan die klasse echter ook nog tal van andere zaken doen. Beeld je bijvoorbeeld een CEO van een bedrijf in die ook adviseur van de President wilt zijn. De bestaande klasse is bijvoorbeeld:

```
class MicrosoftCEO: CEO //CEO kan een parentklasse zijn die elders bijvoorbeeld algemene CEO-concepten beschrijft
{
    public void EarnBigBucks()
    {
        Console.WriteLine("I'm getting rich!!!");
    }
    public void FireDepartement()
    {
        Console.WriteLine("You're all fired!");
    }
}
```

Nu we de interface `IAdvisor` hebben kunnen we deze klasse aanvullen met deze interface:

```
class MicrosoftCEO: CEO, IAdvisor
{

    public void Advise()
    {
        Console.WriteLine("I think you should allow our monopoly. *Grin*");
    }

    public void EarnBigBucks()
    {
        Console.WriteLine("I'm getting rich!!!");
    }
    public void FireDepartement()
    {
        Console.WriteLine("You're all fired!");
    }
}
```

De CEO kan dus z'n bestaande job blijven uitoefenen maar ook als adviseur optreden.

Ook de `President` moet aangepast worden om nu met een lijst van `IAdvisor` ipv `Advisor` te werken:

```
public class MisterPresident
{
    public void RunTheCountry()
    {

        List<IAdvisor> allMinisters= new List<IAdvisor>();
        allMinisters.Add(new MicrosoftCEO());
        //Ask advise from each:
        foreach (IAdvisor minister in allMinisters)
        {
            minister.Advise();
        }
    }
}
```

De eerder beschreven `MilitaryAdvisor`, `ScienceAdvisor` en `EconomyAdvisor` dienen ook niet meer van de abstracte klasse `Advisor` (deze zou je kunnen verwijderen) over te erven en kunnen gewoon de interface implementeren:

```
class MilitaryMinistor:IAdvisor
{
    public void Advise()
    {
        increaseTroopNumbers();
        improveSecurity();
        payContractors();
    }

    private void increaseTroopNumbers()
    {
        //...
    }
}
```

```
    }
    private void improveSecurity()
    {
        //...
    }
    private void improveSecurity()
    {
        //...
    }
}
```

Why should I care?



In kleine projecten lijken interfaces wat overkill, en dat zijn ze vaak wel. Van zodra je een iets complexer project krijgt met meerdere klasse die onderling met elkaar allerlei zaken moeten doen, dan zijn interfaces je dikke vrienden! Je hebt misschien al over de [SOLID programmeerprincipes gehoord](#)?

And if not, niet erg. Samengevat zegt SOLID dat we een bepaalde hoeveelheid abstractie inbouwen enerzijds (zodat we niet de gore details van klassen moeten kennen om er mee te programmeren) anderzijds dat er een zogenaamde 'separation of concerns' (SoC) moet zijn (ieder deel/klasse/module van je code heeft een specifieke opdracht).

Met interfaces kunnen we volgens SOLID programmeren: het boeit ons niet meer wat er in de klasse zit, we kunnen gewoon aan de interfaces van een klasse zien wat hij kan doen. Handig toch!

Ok, als de helft van bovenstaande zinnen je wat filosofisch overkwamen, don't worry maar geloof ons: als je tegen interfaces kan programmeren i.p.v. klasse dan zal je code vaak een pak beter worden in de long run.

Toch nog voorbeeld? Beeld je in dat je een complexe klasse `DiskWriter` hebt die je programma gebruikt om data van en naar de harde schijf te schrijven. De klasse implementeert een interface `IData` die twee methoden heeft (`ReadData()` en `WriteData()`). Als je later beslist om je data naar een online server te schrijven en niet naar de harde schijf, dan kan je gewoon die klasse schrijven (bv. `InternetWriter`) en vervolgens ook de `IData` interface laten implementeren. Al je andere code moet dan niet aangepast worden! Ze (je andere klassen) kunnen gewoon blijven zeggen `ReadData` en `WriteData` en weten misschien zelfs niet dat hun data niet meer naar de HD maar naar het internet wordt gestuurd. Mooi toch!

Interfaces in praktijk

De bestaande .NET klassen gebruiken vaak interfaces om bepaalde zaken uit te voeren. Zo heeft .NET tal van interfaces gedefinieerd waar je zelfgemaakte klassen mogelijk aan moeten voldoen indien ze bepaalde bestaande methoden wensen te gebruiken. Een typisch voorbeeld is het gebruik van de `Array.Sort` methode. We tonen dit in een voorbeeld zo meteen.

Enkele veelgebruikte interfaces binnen .NET (louter ter info):

- `IEnumerable` (and `IEnumerable`): for use with foreach and LINQ
- `IDisposable`: for resources requiring cleanup, used with `using`
- `IQueryable`: lets you execute requests against queriable data sources.
- `INotifyPropertyChanged` : For data binding to UI classes in WPF, winforms and silverlight
- `IComparable` and `IComparer`: for generalized sorting *`IEquatable` and `IEqualityComparer`: for generalized equality
- `IList` and `ICollection`: for mutable collections
- `IDictionary`: for lookup collections

Sorteren met `Array.Sort` en de `IComparable` interface

Stap 1: Het probleem

Indien je een array van objecten hebt en je wenst deze te sorteren via `Array.Sort` dan dienen de objecten de `IComparable` interface te hebben.

We willen een array van landen kunnen sorteren op grootte van oppervlakte.

Stel dat we de klasse `Land` hebben:

```
class Land
{
    public string Naam {get;set;}
    public int Oppervlakte {get;set;}
    public int Inwoners{get;set;}
}
```

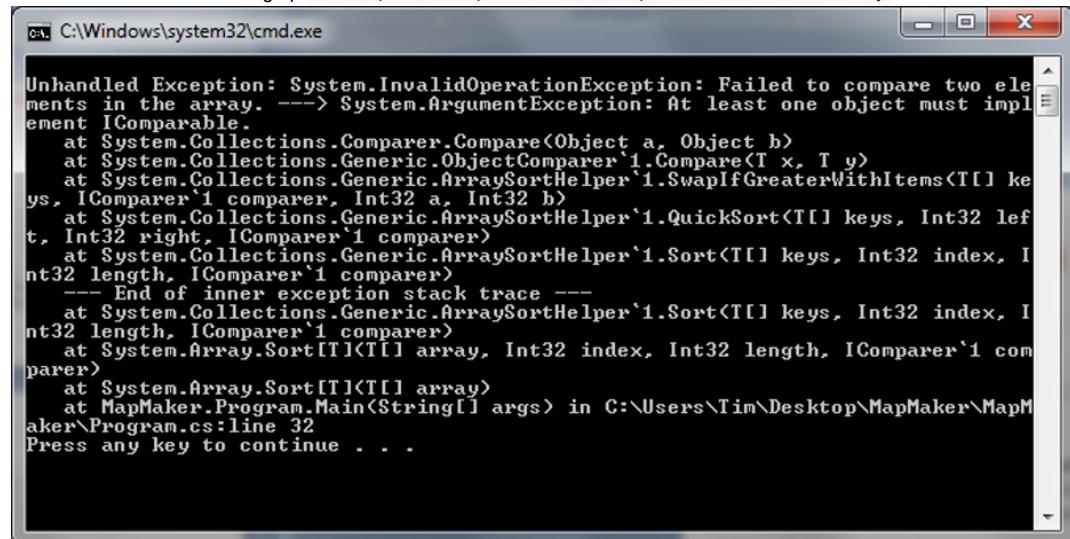
We plaatsen 3 landen in een array:

```
Land[] eurolanden = new Land[3];
eurolanden[0]= new Land() {Naam="Belgie", Oppervlakte= 5, Inwoners=2000};
eurolanden[1]= new Land() {Naam="France", Oppervlakte= 7, Inwoners=2500};
eurolanden[2]= new Land() {Naam="Nederland", Oppervlakte= 6, Inwoners=1800};
```

Wanneer we nu proberen:

```
Array.Sort(eurolanden);
```

Dan treedt er een uitzondering op: `InvalidOperationException: Failed to compare two elements in the array`



```
C:\Windows\system32\cmd.exe

Unhandled Exception: System.InvalidOperationException: Failed to compare two elements in the array. ---> System.ArgumentException: At least one object must implement IComparable.
   at System.Collections.Comparer.Compare(Object a, Object b)
   at System.Collections.Generic.ObjectComparer`1.Compare(T x, T y)
   at System.Collections.Generic.ArraySortHelper`1.SwapIfGreaterWithItems<T[]> keys, IComparer`1 comparer, Int32 a, Int32 b)
   at System.Collections.Generic.ArraySortHelper`1.QuickSort<T[]> keys, Int32 left, Int32 right, IComparer`1 comparer)
   at System.Collections.Generic.ArraySortHelper`1.Sort<T[]> keys, Int32 index, Int32 length, IComparer`1 comparer)
   --- End of inner exception stack trace ---
   at System.Collections.Generic.ArraySortHelper`1.Sort<T[]> keys, Int32 index, Int32 length, IComparer`1 comparer)
   at System.Array.Sort[T](T[] array, Int32 index, Int32 length, IComparer`1 comparer)
   at System.Array.Sort[T](T[] array)
   at MapMaker.Program.Main(String[] args) in C:\Users\Tim\Desktop\MapMaker\MapMaker.cs:line 32
Press any key to continue . . .
```

Stap 2: IComparable onderzoeken

We kunnen dit oplossen door de `IComparable` interface in de klasse `Land` te implementeren. We bekijken daarom eerst de documentatie van deze interface ([hier](#)). De interface is beschreven als:

```
interface IComparable
{
    int CompareTo(Object obj);
}
```

OPGELET: Deze interface bestaat al in .NET en mag je dus niet opnieuw in code schrijven!

Daarbij moet de methode een int teruggeven als volgt: | Waarde | Betekenis | | ----- |:-----| | Getal kleiner dan 0 | Huidig object komt **voor** het `obj` dat als parameter werd meegegeven | | 0 | Huidig object komt op **dezelfde** positie als `obj` werd meegegeven | | Getal groter dan 0 | Huidig object komt **na** het `obj` dat als parameter werd meegegeven |

Stap 3: IComparable in Land implementeren

We zorgen er nu voor dat `Land` deze interface implementeert. Daarbij willen we dat de landen volgens oppervlakte worden gesorteerd :

```
class Land: IComparable
{
    //...

    public int CompareTo(object obj)
    {
        Land temp= (Land)obj; //Zetten de parameter om naar land

        if(Oppervlakte > temp.Oppervlakte) return 1;
        if(Oppervlakte < temp.Oppervlakte) return -1;

        return 0;
    }
}
```

Nu zal de Sort werken! `Array.Sort(eurolanden);`

Stel dat vervolgens nog beter willen sorteren: we willen dat landen met een gelijke oppervlakte, op hun inwoners gesorteerd worden:

```
public int CompareTo(object obj)
{
    Land temp= (Land)obj; //Zetten de parameter om naar land

    if(Oppervlakte > temp.Oppervlakte) return 1;
    if(Oppervlakte < temp.Oppervlakte) return -1;
    if(this.Inwoners > temp.Inwoners) return 1;
    if(this.Inwoners < temp.Inwoners) return -1;
```

```
    return 0;  
}
```

Why should I care?



Als ik niet overtuigend genoeg was over het nut van interfaces in het vorige hoofdstuk, dan hoop ik dat bovenstaande voorbeelden je al een beetje hebben kunnen doen proeven van de kracht van interfaces. Gedaan met ons druk te maken wat er allemaal in een klasse gebeurt. Werk gewoon 'tegen' de interfaces van een klasse en we krijgen de ultieme black-box revelatie (see what I did there? :p)!

Figures with interfaces

Gebruik je [Rechthoek-klasse uit de Figuren oefening](#) die je eerder hebt aangemaakt. Maak een List aan waarin je een 10 rechthoek-objecten plaatsen, allen met een verschillende grootte. Zorg ervoor dat je nu je rechthoeken met de Sort()-methode kan sorteren op oppervlakte.

Toon de werking aan in een klein voorbeeld programma. Ignore dit stuk, trying to fix reference (mr Dams): [abstracte klassen in overerving](#)

[Rechthoek-klasse uit de Figuren oefening](#)

[abstracte klassen in overerving](#)

Game

Zie onderaan pagina voor minimale klasse-hiërarchie en interfaces.

Maak een spel dat als volgt werkt:

- De speler dient met zijn pion de overkant van het veld te bereiken.
- Het veld bestaat uit 20 bij 20 vakjes. Op ieder vakje kan maximum één mapelement staan:
 - De speler zelf
 - Een monster
- Een rots
- Een speler kan niet door rotsen of monsters wandelen.
- Een speler kan in zijn beurt telkens één vakje bewegen OF naar rechts schieten:
 - Indien geschoten wordt dan zal het mapelement op het vakje rechts van de speler vernietigd worden (rots of monster)
- Monsters kunnen ook bewegen. In de beurt van de monsters beweegt ieder monster in een willekeurige richting indien er geen rotsen of spelers LINKS van het monster staan.
 - Indien er WEL een rots of speler LINKS van het monster staat dan schiet het monster en vernietigt het de speler of rots.
- Enkel RockDestroyer monsters kunnen schieten. De setup van het spel bestaat uit volgende stappen:
 - Maak een 20 bij 20 array aan en plaats bepaalde hoeveelheid monsters en rotsen op de kaart, behalve op kolom 0.
 - Plaats de speler op de plek 0,10 in de array (midden van kolom 0)
 - Doorloop de volgende stappen tot er winnaar is

Iedere beurt van het spel bestaat uit volgende stappen:

1. Vraag speler om input (bewegen in 1 van de 4 richtingen OF schieten)
2. Voer actie van speler uit
3. Kijk of speler overkant van kaart heeft bereikt, zo ja: gewonnen!
4. Beweeg ieder monster op de kaart in een willekeurige richting
5. Beweeg iedere RockDestroyer OF laat RockDestroyer schieten

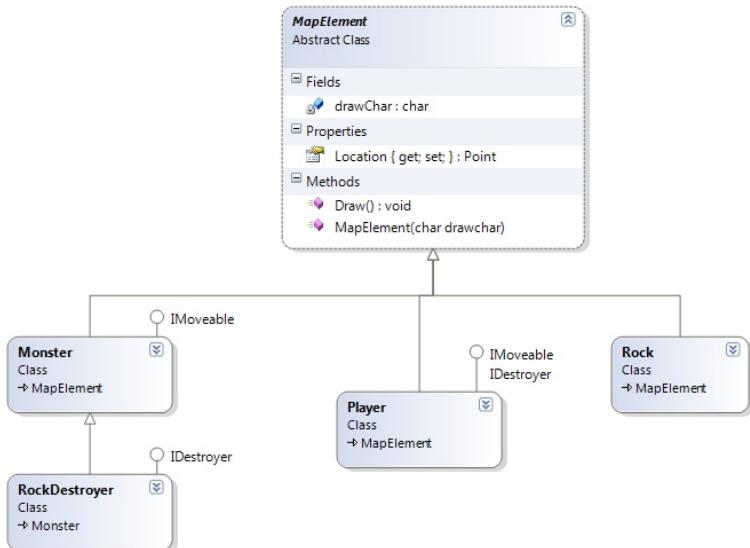
Stel de speler voor door een X, een rots door O , een monster door M een RockDestroyer door D.

Extra's:

Voorgaande beschrijving is een 'minimale' beschrijving. Voorzie extra functionaliteit naar believen zoals:

- Speler heeft levens
- Monsters hebben levens
- Andere soort monsters (bv slimmere)
- Meerdere levels met telkens andere/meer monsters bijvoorbeeld
- Meerdere spelers
- Verder schieten, of schieten in andere richtingen.

Klasse-schema



`Location` is van het type `Point` (compositie). `Point` is een zelfgemaakte mini klasse die er als volgt uit (minimaal uitziet):

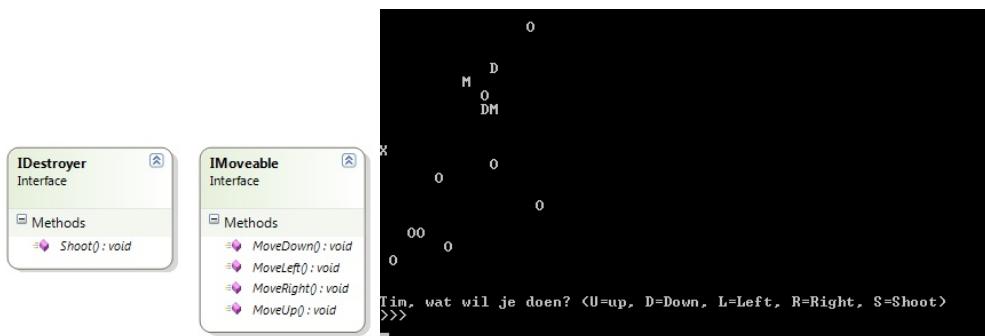
```

class Point
{
    public int X{get;set;}
    public int Y{get;set;}
}

```

En dus in je abstracte klasse `MapElement` zal iets krijgen in de trend van: `public Point Location {get;set;} = new Point();`

Enkele screenshots



Generics

Talen als C# bieden goede mogelijkheden voor strong typing. Dit houdt in dat de compiler tijdens het compileren weet van welk type een variabele is en daardoor welke methoden en properties daarvoor beschikbaar zijn. Dit is gewenst, omdat op deze wijze de compiler goed kan controleren of er geen methoden worden aangeroepen die niet bestaan voor een bepaald type. Hoe meer compile-time gecontroleerd kan worden, hoe minder risico er is op fouten (bv exceptions) die pas at run-time naar voren komen.

Run-time fouten zijn lastiger te detecteren, omdat deze alleen optreden op het moment dat de "foutie" regel code uitgevoerd wordt en dus alleen door middel van testen en gebruik van het systeem boven water kunnen komen wat dus vaak resulteert in bugs. Via generieke klassen kunnen we aan strong typing doen en zo al een groot deel van potentiele problemen at compile-time opvangen. [Bron](#)

Generieke methoden

Vaak schrijf je methoden die hetzelfde doen, maar waarvan enkel het type van de parameters en/of het returntype verschilt. Stel dat je een methode hebt die de elementen in een array onder elkaar toont. Je wil dit werkende hebben voor arrays van ints, string, etc. Zonder generics moeten we dan per type een methode schrijven:

```
public static void ToonArray(int[] array)
{
    foreach (var i in array)
    {
        Console.WriteLine(i);
    }
}

public static void ToonArray(string[] array)
{
    foreach (var i in array)
    {
        Console.WriteLine(i);
    }
}
```

Dankzij generics kunnen we nu het deel dat *generiek* moet zijn aanduiden (in dit geval met `T`) en onze methode eenmalig definiëren:

```
public static void ToonArray<T>(T[] array)
{
    foreach (T i in array)
    {
        Console.WriteLine(i);
    }
}
```

Generic types

In het volgende codevoorbeeld is te zien hoe een eigen generic struct in C# gedefinieerd en gebruikt kan worden. Merk het gebruik van de aanduiding `T`, deze geeft aan dat hier een type (zoals int, double, Student,etc.) zal worden ingevuld tijdens het compileren.

De typeparameter `T` wordt pas voor de specifieke instantie van de generieke klasse of type ingevuld bij het compileren. Hierdoor kan de compiler per instantie controleren of alle parameters en variabelen die in samenhang met het generieke type gebruikt worden wel kloppen.

De afspraak is om .NET een `T` te gebruiken indien het type nog dient bepaald te worden (dit is niet verplicht). [Meer info](#)

Voorbeeld generic type

We wensen een `struct` te maken die de locatie in X,Y,Z richting kan bewaren. We willen echter zowel `float`, `double` als `int` gebruiken om deze X,Y,Z coördinaten in bij te houden:

```
struct Location<T>
{
    public T X;
    public T Y;
```

```
    public T Z;  
}
```

We kunnen deze struct nu als volgt gebruiken:

```
static void Main(string[] args)  
{  
    Location<int> plaats;  
    plaats.X = 34;  
    plaats.Y = 22;  
    plaats.Z = 56;  
  
    Location<double> plaats2;  
    plaats2.X = 34.5;  
    plaats2.Y = 22.2;  
    plaats2.Z = 56.7;  
  
    Location<string> plaats3;  
    plaats3.X = "naast de kerk";  
    plaats3.X = "links van de bakker";  
    plaats3.Z = "onder het hotel";  
}
```

Why should I care?



Generics zijn een zeer krachtig concept van C#. De eerste maanden zal je vooral bestaande generieke klassen gebruiken die de .NET programmeurs reeds voor je hebben geprogrammeerd. Dankzij generics kunnen zij klassen schrijven die veel polyvalenter zijn omdat ze niet gebonden zijn aan een bepaald type. *Heb ik die klassen al gebruikt?* Wie weet. Maar als je straks de [generieke collections ontdekt](#) dan zal je direct verkocht zijn. Nog enkele hoofdstukken geduld dus!

Klassen en generics

Niets houdt ons nu tegen om generieke klassen te maken. We kunnen bijvoorbeeld volgende klasse maken die we kunnen gebruiken met eender welk type om de meetwaarde van een meting in op te slaan:

```
public class Meting<T>
{
    private T waarde;
    //constructor
    public Meting(T waardein)
    {
        waarde = waardein;
    }
    //Public property
    public T Waarde { get { return waarde; } set { waarde = value; } }
    public override string ToString()
    {
        return "Deze meting heeft als waarde:" + waarde.ToString();
    }
}
```

Een voorbeeldgebruik van dit nieuwe type kan zijn:

```
Meting<int> m1 = new Meting<int>(44);
Console.WriteLine(m1);

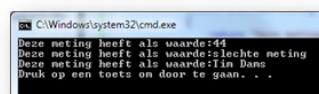
Meting<string> m2 = new Meting<string>("slechte meting");
Console.WriteLine(m2);
```

Of we zouden zelfs een extra klasse kunnen aanmaken genaamd Student

```
public class Student
{
    public string Naam;
    public Student(string n){Naam= n;} //Constructor
    public override string ToString()
    {
        return Naam;
    }
}
```

En dan als volgt een nieuwe meting met daarin een student aanmaken. Let goed op de constructor-aanroep van zowel Meting als Student!

```
Meting<Student> m3= new Meting<Student>(new Student("Tim Dams"));
Console.WriteLine(m3);
```



De uitvoer van dit programma zou dan zijn:

Meerdere types in generics

Zoals reeds eerder vermeld is de T aanduiding enkel maar een afspraak. Je kan echter zoveel T-parameters meegeven als je wenst. Stel dat je bijvoorbeeld een klasse wenst te maken waarbij 2 verschillende types kunnen gebruikt worden. De klassedefinitie zou er dan als volgt uit zien:

```
class DataBewaarder<Type1, Type2>
{
    private Type1 waarde1;
    private Type2 waarde2;
    //Constructor
    public DataBewaarder(Type1 w1, Type2 w2)
```

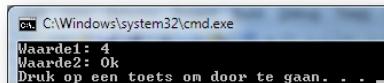
```

    {
        waarde1 = w1;
        waarde2 = w2;
    }
    public override string ToString()
    {
        return "Waarde1: " + waarde1 + "\nWaarde2: " + waarde2;
    }
}

```

Het gebruik ervan:

```
DataBewaarder<int, string> d1 = new DataBewaarder<int, string>(4, "Ok");
```



Met als uitvoer:

Constraints

We willen vaak voorkomen dat bepaalde types wel of niet gebruikt kunnen worden in je zelfgemaakte generieke klasse. Stel bijvoorbeeld dat je een klasse schrijft waarbij je de `CompareTo()` methode wenst te gebruiken. Dit gaat enkel indien het type in kwestie de `IComparable` interface implementeert. We kunnen als constraint ("beperking") dan opgeven dat de volgende klasse enkel kan gebruikt worden door klassen die ook effectief die interface implementeren (en dus de `CompareTo()`-methoden hebben). Het geel deel geeft de constraint aan. We zeggen dus letterlijk ("waar T overerft van `IComparable`"):

```

public class Wijziging<T> where T : IComparable
{
    public T vorigewaarde;
    public T huidigewaarde;
    public Wijziging(T vorig, T huidig)
    {
        vorigewaarde = vorig;
        huidigewaarde = huidig;
    }

    public bool IsGestegen()
    {
        if (huidigewaarde.CompareTo(vorigewaarde) > 0)
            return true;
        else return false;
    }
}

```

Volgende gebruik van deze klasse zou dan True op het scherm tonen:

```
Wijziging<double> w = new Wijziging<double>(3.4, 3.65);
Console.WriteLine(w.IsGestegen());
```

Mogelijke constraints

Verschillende zaken kunnen als constraint optreden. Naast de verplichting dat een bepaalde interface moet worden geïmplementeerd kunnen ook volgende constraints gelden([meer info](#)):

- Enkel value types
- Enkel klassen
- Moet default constructor hebben
- Moet overerven van een bepaalde klasse

Why should I care?



Goh, eerlijk is eerlijk: in je prille programmeursbestaan zal je niet heel vaak de nood zien om je eigen generieke klasse te schrijven. Laat staan dat je ook nog eens allerlei constraints gaat definiëren.

Collections

Nadelen van arrays

We hebben al eerder gezien hoe we arrays kunnen gebruiken om gegevens van hetzelfde type in één datastructuur voor te stellen. Hoewel arrays handig zijn, hebben ze ook een aantal nadelen:

- Je weet niet altijd van tevoren hoe groot de array zal worden, nochtans moeten we op voorhand ruimte reserveren door op te geven hoe groot de array moet zijn. Hierdoor kan het voorkomen dat we veel computergeheugen reserveren voor een array die mogelijk nooit volledig gevuld zal zijn.
- Omgekeerd kan het ook zijn dat we onze array te klein definiëren en we dus niet alle data kwijtraken in de array. We kunnen dit, omslachtig, oplossen door een nieuwe, grotere array te definiëren en daar de oude naar te kopiëren, inclusief te nieuwe data.
- Wanneer de array value-types bevat (int, double, struct, etc.) en deze array kopiëren naar een nieuwe array waar we vervolgens een waarde aanpassen, dan zal enkel de waarde in de gekopieerde array veranderen, niet in de originele. Indien we echter een array van objecten (referentietypes) maken en in een kopie een referentie aanpassen, dan zal deze ook aangepast worden in de originele array.

Collecties: geavanceerde arrays

Collecties zijn speciale objecten waarin een reeks van andere objecten kan worden gerefereerd. Daar collecties zelf ook objecten zijn, heeft dit een paar gevolgen:

- Er moet eerst een instantie van een collectie aangemaakt worden voor je ze kan gebruiken (m.b.v. `new` keyword)
- Collecties hebben properties die onder andere het aantal objecten in collectie weergeeft.
- Collecties hebben methoden om onder andere objecten toe te voegen en verwijderen
- Daar collecties beschreven zijn in klassen kan je de typische object-georiënteerde eigenschappen hierop toepassen: je kan collecties overerven, encapsuleren (composiet), etc.

Er bestaan een hele resem collecties en je hebt er zelfs al van gebruikt, namelijk de `List<>`-klasse (weliswaar de generic variant, de niet-generieke versie heet `ArrayList`). Volgende tabel geeft een beschrijving van de meest gebruikte collecties die in de `System.Collections Namespace` zitten [Bron MSDN](#)

| Klasse | Beschrijving |
|-------------------------|---|
| <code>ArrayList</code> | Standaard collectie die dynamisch kan groeien (zelfde als de <code>List<></code> generieke variant). |
| <code>BitArray</code> | Array die enkel Booleans (true/false) bevat. Handig om dus een bit-voorstelling te doen. |
| <code>HashTable</code> | Ieder element bevat een waarde en een bijhorende sleutel als index. |
| <code>Queue</code> | Stelt een first-in, first-out (FIFO) lijst voor. |
| <code>SortedList</code> | Soortgelijk als <code>HashTable</code> , een sleutel stelt de index voor, elementen worden echter onmiddellijk gerangschikt . |
| <code>Stack</code> | Stelt een Last-in, first-out (LIFO) lijst voor. |

We zullen een aantal van de eerder vernoemde collectie-klassen verderop bekijken, maar we zullen onmiddellijk de generieke varianten bekijken omdat die veel handiger zijn. Het is namelijk zo dat generic collecties meestal bruikbaarder en veiliger zijn dan de gewone collecties die we zonet kort hebben beschreven. Generic collecties zijn 'strongly typed' en bieden een betere veiligheid op gebied van 'type safety', hun gebruik is echter quasi identiek dan dat van de niet generieke collecties.

Collection namespace

As je een collection-klasse wilt gebruiken, vergeet dan niet de `System.Collection` namespace toe te voegen. Indien je een generieke collection nodig hebt (zie verder) dan dien je de `System.Collections.Generic` namespace te gebruiken

Type-safety

Een niet-generieke collection is **niet type-safe**. Een generieke collection is dat wel.

Volgende 2 code-voorbeelden tonen dit.

In het niet-generieke geval zal deze code compileren maar **tijdens de uitvoer** zal de laatste lijn een fout (Exception) geven::

```
ArrayList nietGeneriekeList=new ArrayList();
string naam= "Tim";
nietGeneriekeList.Add(naam);
```

```
int leeftijd = (int)nietGeneriekeList[0];
```

Bij een generieke collection zal er bij soortgelijk code een compiler-error optreden (gedrag dat meestal wenselijk is) en de code zal dus niet gecompileerd kunnen worden:

```
List<string> generiekeList=new List<string>();  
string naam= "Tim";  
generiekeList.Add(naam);  
int leeftijd = (int)nietGeneriekeList[0]; // compilererror: cannot convert type string to int
```

Generic collections

Generic collecties zijn 'strongly typed' en bieden een betere veiligheid op gebied van 'type safety', hun gebruik is echter quasi identiek aan dat van de niet generieke collecties.

We hebben reeds 1 generic collectie-klasse veel gebruikt, namelijk de List<T>-klasse, waarbij we tussen de haakjes het type opgaven dat de List kan bevatten , bv List of List. Met andere woorden, je hebt reeds vorig academiejaar generics gebruikt zonder het goed en wel te beseffen!

Een volledig overzicht van alle mogelijk generic collections vind je hier terug [MSDN](#).

We beschrijven nu de werking van een aantal typische collecties, merk op dat deze werking quasi identiek is als die voor de niet-generische versie.

List collectie

Een List collectie is de meest standaard collectie die je kan beschouwen als een veiligere variant op een een doodnormale array. Via de Add(T item) methode kan je elementen toevoegen aan de lijst. In volgende voorbeeld maken we een List aan die objecten van het type string mag bevatten:

```
List<String> myStringList = new List<String>();
```

myStringList.Add("This is the first item in my list!"); Het leuke van een List is dat je deze ook kan gebruiken als een gewone array, waarbij je mbv de indexer elementen kan aanroepen. Stel bijvoorbeeld dat we een lijst hebben met minstens 4 strings in. Volgende code toont hoe we de string op positie 3 kunnen uitlezen en hoe we die op positie 2 overschrijven:

```
Console.WriteLine(myStringList[3]);  
myStringList[2] = "andere zin";
```

Interessante methoden en properties voorts zijn:

- `Count` : property die teruggeeft hoeveel elementen in de lijst aanwezig zijn.
- `Clear()` :methode die de volledige lijst leegmaakt
- `Insert()` : methode om element op specifieke plaats in lijst toe te voegen, bijvoorbeeld:

```
myStringList.Insert(3,"A fourth sentence");
```

voegt de string toe op de vierde (3+1) plek.

- `Contain(T item)` : geef als parameter een specifiek object mee (van het type dat de List<T> bevat) om te weten te komen of dat specifieke object in de List<T> terug te vinden is. Indien ja dan zal true worden teruggeven.

- `IndexOf(T item)` : geeft de index terug van het element item in de rij. Indien deze niet in de lijst aanwezig is dan wordt -1 teruggegeven.

Foreach loops

Je kan met een eenvoudige for of while-loop over een list-object itereren, maar het gebruik van een foreach-loop is toch handiger ([ook al eerder besproken hier](#)). Dit is dan ook de meestgebruikte operatie om eenvoudig (je kan ook Linq overwegen!) en snel een bepaald stuk code toe te passen op ieder element van de lijst:

```
List<int> integerList=new List<int>();  
integerList.Add(2);  
integerList.Add(3);  
integerList.Add(7);
```

```

foreach(int prime in integerList)
{
    Console.WriteLine(prime);
}

```

Queue<> collectie

Een queue (Ned. : rij) stelt een FIFO-lijst voor. Een queue stelt dus de rijen voor die we in het echte leven ook hebben wanneer we bijvoorbeeld aanschuiven aan een ticketverkoop. Met deze klasse kunnen we zo'n rij simuleren en ervoor zorgen dat steeds het 'eerste/oudste' element in de rij als eerste wordt behandeld. Nieuwe elementen worden achteraan de rij toegevoegd.

We gebruiken 2 methoden om met een queue te werken:

- `Enqueue(T item)` : Voeg een item achteraan de queue toe.
- `Dequeue()` : geeft het eerste element in de queue terug en verwijderd dit element vervolgens uit de queue.

Voorbeeld:

```

Queue<string> theQueue = new Queue<string>();
theQueue.Enqueue("I arrived here first!");
theQueue.Enqueue("I arrived second.");
theQueue.Enqueue("I'm last");

Console.WriteLine(theQueue.Dequeue());
Console.WriteLine(theQueue.Dequeue());

```

Dit zal op het scherm tonen:

```

I arrived here first!
I arrived here second.

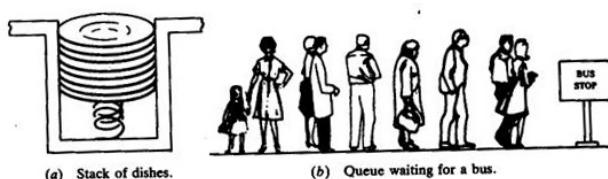
```

Peek

Een andere interessante methode is de `Peek()` methode; hiermee kunnen we kijken in de queue wat het eerste element is, zonder het te verwijderen.

Stack collectie

Daar waar een queue first-in-first-out is, is een stack last-in first out. M.a.w. het recentst toegevoegde element zal steeds vooraan staan en als eerste verwerkt worden. Je kan dit vergelijken met een stapel waar je steeds bovenop een nieuw object legt.



We gebruiken volgende 2 methoden om met een stack te werken:

- `Push(T item)` : plaats een nieuw item bovenop de stapel
- `Pop()` : geeft het bovenste element in de stack terug en verwijderd vervolgens dit element van de stack

```

Stack<string> theStack = new Stack<string>();
theStack.Push("I arrived here first!");
theStack.Push("I arrived second.");
theStack.Push("I'm last");

Console.WriteLine(theStack.Pop());
Console.WriteLine(theStack.Pop());

```

Dit zal dus het volgende resultaat geven:

```

I'm last
I arrived second.

```

Dictionary collectie

In de niet generieke-collections heb je de HashTable, de generische versie is de dictionary. In een dictionary wordt ieder element voorgesteld door een index en de waarde van het element. De sleutel moet een unieke waarde zijn zodat het element kan opgevraagd worden uit de dictionary aan de hand van deze sleutel.

Bij de declaratie van de dictionary dien je op te geven wat het type van de key zal zijn , alsook het type van de waarde (value). In het volgende voorbeeld maken we een dictionary van klanten, iedere klant heeft een unieke ID (de key) alsook een naam (die niet noodzakelijk uniek is):

```
Dictionary<int, string> customers = new Dictionary<int, string>();
customers.Add(123, "Tim Dams");
customers.Add(6463, "James Bond");
customers.Add(666, "The beast");
customers.Add(700, "James Bond");
```

Merk op dat je niet verplicht bent om een int als key te gebruiken, dit mag eender wat zijn, zelfs een klasse.

We kunnen nu met behulp van een foreach-loop alle elementen tonen (merk op dat de foreach-constructie voor eender welke type collectie kan gebruikt worden om doorheen een array te lopen):

```
foreach (var item in customers)
{
    Console.WriteLine(item.Key + "\t:" + item.Value);
}
```

We kunnen echter ook een specifiek element opvragen aan de hand van de key. Stel dat we de waarde (naam) van de klant met key (id) gelijk aan 123 willen tonen:

```
Console.WriteLine(customers[123]);
```

De key werkt dus net als de index bij gewone arrays, alleen heeft de key nu geen relatie meer met de positie van het element in de collectie.

Eender welk type voor key en value

De key kan zelfs een string zijn en de waarde een ander type. In het volgende voorbeeld hebben we eerder een klasse Student aangemaakt. We maken nu een student aan en voegen deze toe aan de studentenLijst. Vervolgens willen we de leeftijd van een bepaalde student tonen op het scherm en vervolgens verwijderen we deze student:

```
Dictionary<string, Student> studentenLijst = new Dictionary<string, Student>();
Student stud= new Student();
stud.Naam= "Tim";stud.Leeftijd=24;
studentenLijst.Add("AB12",stud);
Console.WriteLine(studentenLijst["AB12"].Leeftijd);
studentenLijst.Remove("AB12");
```

Why should I care?



Als je die vraag moet stellen dan heb je niet veel begrepen van arrays en dit hoofdstuk. Arrays zijn fijn en krachtig, toch zal je vaak allerlei extra methoden beginnen schrijven om het werken met je array wat eenvoudiger te maken. Van zodra je code schrijft om sneller elementen aan arrays toe te voegen, verwijderen, verzetten, etc. dan wordt het tijd om te overwegen om een of andere generieke collection te gebruiken.

KlasOrganizer Deluxe II

Maak een klasse `KlasOrganizer` die een lijst van studenten als public member heeft (gebruik `List<Student>`) en verwerk deze nieuwe klasse in het project [Student Organizer Deluxe](#). Een KlasOrganizer object bevat alle studenten van een klas en heeft 1 publieke methode `GenereerOverzicht()` . Dit overzicht zal het klasgemiddelde per vak teruggeven, alsook het aantal studenten, etc.

Deze klasse bevat voorts een variabele die de klas bewaard (van type enum `Klas`).

Is en As

We introduceren twee nieuwe C# keywords: `is` en `as`.

Is keyword

Het `is` keyword is een operator die je kan gebruiken om te weten te komen of:

- Een object van een bepaalde type is
- Een object een bepaalde interface bevat

`is` geeft een bool terug.

Is voorbeeld 1

Stel dat we volgende drie klassen hebben:

```
class Vehicle {}  
  
class Car: Vehicle{}  
  
class Person {}
```

Een Car **is** een Vehikel. Een Person **is geen** Vehikel.

Stel dat we enkele variabelen hebben als volgt:

```
Car myCar= new Car();  
Person rambo= new Person();
```

We kunnen nu de objecten met `is` bevragen of ze van een bepaalde type zijn:

```
if(myCar is Vehicle)  
{  
    Console.WriteLine("The first object is a Vehicle");  
}  
if(rambo is Vehicle)  
{  
    Console.WriteLine("The second object is a Vehicle");  
}
```

De uitvoer zal worden: `The first object is a Vehicle`.

Is voorbeeld 2

Ook kunnen we dus `is` gebruiken om te weten of een klasse een specifieke interface heeft. Stel:

```
interface IDeletable{ ...};  
  
class Book: IDeletable { ... };  
  
class Person { ... };
```

In actie:

```
Person tim= new Person();  
Book gameofthrones = new Book();  
  
if(gameofthrones is IDeletable)  
{  
    Console.WriteLine("I can delete game of thrones");  
}  
if(tim is IDeletable)  
{  
    Console.WriteLine("I can delete tim");  
}
```

Ouput: I can delete game of thrones .

As keyword met voorbeeld

Wanneer we objecten van het ene naar het andere type willen omzetten dan doen we dit vaak met behulp van [casting](#):

```
Student fritz= new Student();
Mens jos = (Mens)fritz;
```

Het probleem bij casting is dat dit niet altijd lukt. Indien de conversie niet mogelijk is zal een Exception gegenereerd worden en je programma zal crashen als je niet aan [Exception handling doet](#).

Het `as` keyword lost dit op. Het keyword zegt aan de compiler 'probeer dit object te converteren. Als het niet lukt, zet het dan op `null` in plaats van een Exception te werpen.

De code van daarnet herschrijven we dan naar:

```
Student fritz= new Student();
Mens jos =fritz as Mens;
```

Indien nu de casting niet lukt (omdat "Student" misschien geen childklasse van "Mens" blijkt te zijn) dan zal `jos` de waarde `null` hebben gekregen.

We kunnen dan vervolgens bijvoorbeeld schrijven:

```
Student fritz= new Student();
Mens jos =fritz as Mens;
if(jos!=null)
{
    //Doe Mens-zaken
}
```

Why should I care?



De `is` en `as` keywords laten toe om meer dynamische code te schrijven. Mogelijk weet je niet op voorhand wat voor datatype je code zal moeten verwerken en wordt polymorfisme je oplossing. Maar dan? Dan komen `is` en `as` to the rescue!

Je, dank zij polymorfisme, gevuld lijst van objecten van allerhande typen wordt nu beheersbaarder. Je kan nu, met `is` een element bevragen of hij van een bepaald type is. Vervolgens kan je met `as` het element even 'omzetten' naar z'n effectieve type (en dus meer doen dan wat hij kan in de vermomming van z'n eigen basistype).

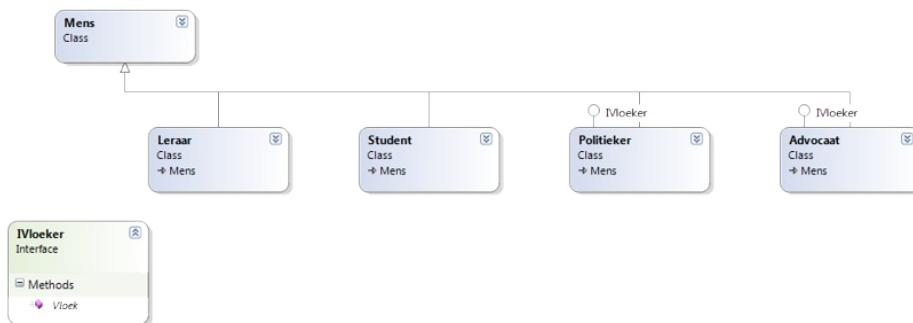
Polymorfisme, interfaces en is/as

De eigenschappen van polymorfisme en interfaces combineren kan zeer krachtige tot zeer krachtige code resulteren wanneer we de `is` en `as` keywords gebruiken. We tonen dit in een voorbeeld.

Vloekende mensen: Opstart

De idee is de volgende: mensen kunnen spreken. Leraren, Studenten, Politieker, en ja zelfs Advocaten zijn mensen. Echter, enkel Politiekers en Advocaten hebben ook de interface `IVloeker` die hen toelaat eens goed te vloeken. Brave leerkrachten en studenten doen dat niet. We willen een programma dat lijsten van mensen bevat waarbij we de vloekers kunnen doen vloeken zonder complexe code te moeten schrijven.

We hebben volgende klasse-structuur:



Als basse-is klasse `Mens` hebben we dan:

```
public class Mens
{
    public void Spreek()
    {
        Console.WriteLine("Hoi!");
    }
}
```

Voorts definiëren we de interface:

```
interface IVloeker
{
    void Vloek();
}
```

We kunnen nu de nodige child-klassen maken.

1. De niet vloekers: `Leraar` en `Student`
2. De vloekers: `Advocaat` en `Politieker`

```
class Leraar:Mens {} //moet niets speciaal doen

class Student:Mens{} //ook studenten doen niets speciaal

class Politieker: Mens, IVloeker
{
    public void Vloek()
    {
        Console.WriteLine("Godvermiljaardeerdeju, zei de politieker");
    }
}

class Advocaat: Mens, IVloeker
{
    public void Vloek()
    {
        Console.WriteLine("SHIIIT, zei de advocaat");
```

```
    }  
}
```

Vloekende mensen: Het probleem

We maken een array van mensen aan:

```
Mens[] mensjes = new Mens[4],  
mensjes[0]= new Leraar();  
mensjes[1]= new Politieker();  
mensjes[2]= new Student();  
mensjes[3]= new Advocaat();  
  
for(int i =0; i<mensjes.Length; i++)  
{  
    //NOW WHAT?  
}
```

Het probleem: hoe kan ik in de array van mensen (bestaande uit een mix van studenten, leraren, advocaten en politiekers) s enkel de vloekende mensen laten vloeken?

Oplossing 1 - Vloekende mensen: `is` to the rescue

De eerste oplossing is door gebruik te maken van het `is` keyword. We zullen de lijst doorlopen en steeds aan het huidige object vragen of dit object de `IVloeker` interface bezit, als volgt:

```
for(int i =0; i<mensjes.Length; i++)  
{  
    if(mensjes[i] is IVloeker)  
    {  
        //NOW WHAT ?  
    }  
    else  
    {  
        mensjes[i].Spreek();  
    }  
}
```

Vervolgens kunnen we binnen deze if het huidige object tijdelijk omzetten ([casten](#)) naar een `IVloeker` object en laten vloeken:

```
if(mensjes[i] is IVloeker)  
{  
    IVloeker tijdelijk= (IVloeker)mensjes[i];  
    tijdelijk.Vloek();  
}
```

Oplossing 2 - Vloekende mensen: `as` to the rescue

Het `as` keyword kan ook een toffe oplossing geven. Hierbij zullen we het object proberen omzetten via `as` naar een `IVloeker`. Als dit lukt (het object is verschillend van `null`) dan kunnen we het object laten vloeken:

```
for(int i =0; i<mensjes.Length; i++)  
{  
    IVloeker tijdelijk = mensjes[i] as IVloeker;  
    if(tijdelijk !=null)  
    {  
        tijdelijk.Vloek();  
    }  
    else  
    {  
        mensjes[i].Spreek();  
    }  
}
```

Why should I care



In de echte C# wereld houdt men van interfaces. Het is veel beter om 'tegen interfaces' te programmeren dan tegen klassen. Het maakt je code veel beter qua onderhoud en complexiteit.

Met `is` en `as` kan je snel objecten van eender welk type bevragen:

- "Heb *jj* interface *x*?" m.b.v. `is` (bv. `if(object is ISomething)`)
- "Wordt *jj* eens even interface *x*!" m.b.v. `as` (bv. `object as ISomething`)

Objecten testen op gelijkheid: the missing manual

In dit hoofdstuk gaan we dieper in hoe we objecten kunnen vergelijken op gelijkheid met behulp van de `Equals` methode die in `System.Object` gedefinieerd wordt. We hebben dit reeds behandeld in het hoofdstuk over [System.Object](#). We hebben nu echter voldoende bagage dankzij de voorgaande hoofdstukken om een complete oplossing te vinden.

We zullen stap voor stap opbouwen en motiveren waarom dit de enige correcte manier is.

Als leuk zij-effect krijgen we het feit dat deze uitleg aardig wat reeds opgedane kennis vereist: we kunnen deze zaken dus ineens herhalen en toelichten in de context van de `Equals` methode.

Heap en stack

C# programma's gebruiken twee soorten geheugens zoals we reeds in [dit hoofdstuk](#) lazen

- De stack
- De heap

De volledige werking van deze geheugens gaan we niet terug herhalen. Voor ons belangrijk is te weten dat variabelen van built-in .NET types (int , char, etc.) in de stack worden bewaard. Objecten daarentegen worden in de heap bewaard. Indien je een object aanmaakt met de new operator en deze bewaard in een lokale variabele zoals hier:

```
Point punt1 = new Point();
```

dan zal in de heap een `Point` object worden aangemaakt. Vervolgens krijgt een kleine variabele genaamd `punt1` in de stack het adres naar dat aangemaakte object. `punt1` zelf bevat dus niet meer dan een geheugenadres én dus niet het eigenlijke punt. We zullen dit verder op nodig hebben.

Objecten vergelijken

Value types en de == operator

Wanneer we twee variabele van een van de ingebouwde .NET types (behalve string) met elkaar vergelijken (int, char, etc.) dan kunnen we schrijven:

```
int getal1 = 4;
int getal2= 5;
if(getal1 == getal2)
{
    Console.WriteLine("Getallen zijn gelijk");
}
```

Dit werkt omdat de variabelen `getal1` en `getal2` in het geheugen de effectieve waarden 4 en 5 bezitten. **Maar wat gebeurt er indien we twee objecten met elkaar op deze manier vergelijken?**

Objecten en de == operator

Stel dat we een klasse Point hebben dat we gebruiken om een 2-dimensionaal punt voor te stellen:

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

We zouden nu kunnen denken dat volgende code `Punten zijn gelijk` op het scherm zal tonen:

```
Point punt1= new Point();
punt1.X=3;
punt1.Y=5;
```

```

Point punt2= new Point();
punt2.X=3;
punt2.Y=5;

if(punt1==punt2)
{
    Console.WriteLine("Punten zijn gelijk");
}

```

Echter, objecten worden we weten dat objecten "by reference" in het geheugen worden bewaard. Wat dit wil zeggen is dat de variabelen `punt1` en `punt2` in het geheugen niet het volledig object bevatten. Ze bevatten enkel een geheugenadres (pointer, referentie) naar een andere plaats (in de heap) waar het volledige object zich bevindt.

Wanneer we dus de expressie `punt1==punt2` schrijven dan zal de inhoud van die 2 variabelen worden vergeleken, zijnde de 2 adressen. Daar beide variabelen naar een ander adres wijzen zal deze test dus fout teruggeven. Als we 1 extra lijn voor de if toevoegen:

```

punt1 = punt2;

if(punt1==punt2)
{
    Console.WriteLine("Punten zijn gelijk");
}

```

Dan zal de test wel `true` teruggeven: we hebben vlak voor de test gezegd dat er in de variabele `punt1` het geheugenadres moet komen dat ook in `punt2` staat. Met andere woorden: zowel `punt1`, als `punt2` bevatten nu hetzelfde adres, namelijk dat van het object met waarden `X=3` en `Y=5`. Het object met `Y=5` zijn we kwijt door de garbage collector: die heeft gezien dat er geen enkele variabele meer wijst naar dat object en heeft het dus verwijderd. De variabelen `punt1` en `punt2` zijn nu dus wel gelijk: ze hebben dezelfde inhoud, namelijk hetzelfde adres naar hetzelfde object.

Objecten vergelijken zonder overerving

Hoe kunnen we dan wel 2 objecten vergelijken? Hiervoor dienen we, manueel, alle properties en private fields te vergelijken met elkaar van beide objecten. Althans, jij als programmeur moet beslissen wanneer 2 objecten gelijk zijn. Mogelijk vind je dat 2 punten gelijk zijn als ze beide dezelfde X-waarde hebben ongeacht de Y-waarde. Maar wij prefereren natuurlijk dat zowel de X als de Y-waarde dezelfde is en kunnen dus schrijven:

```
if(punt1.X== punt2.X && punt1.Y== punt2.Y)
```

We zouden dit dan in een handige `static methode` kunnen plaatsen :

```

static bool VergelijkPunten(Point p1, Point p2)
{
    if(punt1.X== punt2.X && punt1.Y== punt2.Y)
    {
        return true;
    }
    return false;
}

```

Nog leuker zou natuurlijk zijn als we de vergelijking **in** het object zelf kunnen doen. We zouden dan aan een object kunnen vragen "beste punt1, is het volgende punt dat ik als parameter meegeef gelijk aan jou of niet?". Hiermee leggen we de verantwoordelijkheid bij het object en zorgen we dat alles mooi geencapsuleerd en samen blijft. Onze uitgebreide Point-klasse wordt dan:

```

class Point
{
    public int X {get;set;}
    public int Y {get;set;}

    public bool IsbitPuntGelijk(Point anderePunt)
    {
        if(X== anderePunt.X && Y== anderePunt.Y)
        {
            return true;
        }
        return false;
    }
}

```

Ieder object van het type Point bevat dus nu een methode `IsDitPuntGelijk` waarbij we een ander punt meegeven als parameter. Daar de test in het punt zelf wordt gedaan hebben we onmiddellijke toegang tot alle, al dan niet, private variabelen van het punt in kwestie.

Elders kunnen we dus schrijven:

```
if(punt1.IsDitPuntGelijk(punt2))
```

In de wereld waar we overerving nog niet kennen zou dit een mooi einde zijn van de oefening...maar we kennen overerving en gaan dus een stapje verder.

System.Object: De grondlegger van alles

Uit een [vorige hoofdstuk](#) weten we dat alle klassen overerven van `System.Object` en dat deze een methode `Equals` bevat. Deze werd speciaal toegevoegd om objecten op gelijkheid te testen. We moeten echter de implementatie zelf schrijven, daar .NET niet kan voorspellen hoe jij vindt dat objecten dezelfde zijn.

Wat we vorige keer niet zagen is dat er twee versies van de `Equals` methode beschikbaar zijn in `System.Object`:

- Een static versie
- Een gewone object-methode versie

```
object object1= new object();
object object2= new object();

//object methode versie
if(object1.Equals(object2))
    Console.WriteLine("gelijk!");

//static versie
if(Object.Equals(object1, object2))
    Console.WriteLine("gelijk!");
```

Equals als virtual methode

Wij gaan ons nu concentreren op de eerste, niet-statistische, `Equals` methode. Bekijken we de signature van de `Equals` methode in `System.Object` dan zien we:

```
public virtual bool Equals(object obj)
```

Met andere woorden, deze methode is `virtual` gemaakt zodat andere klasse deze methode kunnen override'n. Laten we dit eerst eens **niet** doen. Daar `Point` van `System.Object` overerft kunnen we schrijven:

```
Point punt1= new Point();
punt1.X=3;
punt1.Y=5;

Point punt2= new Point();
punt2.X=3;
punt2.Y=5;

if(punt1.Equals(punt2))
{
    Console.WriteLine("Punten zijn gelijk");
}
```

Standaard zal de `Equals` methode van `System.Object` simpelweg kijken of beide objecten naar hetzelfde geheugenadres wijzen (zoals eerder reeds aangehaald). Dit is hier niet het geval. Volgende code uitvoeren zal wél ``Punten zijn gelijk'' op het scherm tonen:

```
punt1=punt2;
if(punt1.Equals(punt2))
{
    Console.WriteLine("Punten zijn gelijk");
}
```

`System.Object` Weet natuurlijk niet welke andere klassen allemaal de originele Equals methode zal gebruiken en kan dus ook moeilijk voor al die andere klassen de nodige code voorzien. We zullen dus in onze `Point` klasse de `Equals` methode moeten herschrijven waarbij we ze zullen laten werken zoals we willen: *punten zijn gelijk indien zowel hun X als hun Y waarde dezelfde is.*

Wanneer we een bestaande methode willen overriden dan moeten we **EXACT DEZELFDE SIGNATURE** overnemen van de originele methode. De signature van de originele methode is `virtual bool Equals(object obj)`.

We zijn dus verplicht om deze methode zo over te nemen in onze `Point` klasse, waarbij we het `virtual` keyword natuurlijk vervangen door `override`:

```
class Point: System.Object
{
    public int X {get;set;}
    public int Y {get;set;}

    public override bool Equals (object obj)
    {
        return true;
    }
}
```

Daar we stevast `true` returnen hierboven zal onderstaande code altijd in de if gaan. Alle `Point` objecten zouden gelijk zijn, ongeacht of dat nu is of niet:

```
if(punt1.Equals(punt2))
```

Merk op dat dit dezelfde code is als voor we de `Equals` methode override'n. Echter, aangezien we de `Equals` methode *we/* override'n zullen we dus de implementatie uitvoeren die in de `Point` klasse staat, en niet die van `System.Object`.

Polymorfisme duikt op

Maar nu komt het nieuwe element om de hoek kijken: *hoe kunnen we nu binnen onze nieuwe Equals methode de punten vergelijken?* We krijgen binnen de `Equals` methode een parameter van het type `Object` binnen...

Als we het volgende schrijven dan begint Visual Studio te huilen:

```
public override bool Equals (object obj)
{
    if(X=obj.X && Y= obj.Y) //BAAAAD CODE
        return true;
    return false
}
```

Inderdaad. De `obj` parameter is van de `Object` klasse, en deze klasse heeft geen `x` en `y` properties. We zullen dus die `obj` parameter naar een punt moeten transformeren. Dit kan op twee manieren:

- Via casting: `Point t= (Point)obj;`
- Via `as`: `Point t= obj as Point;`

Laten we even veronderstellen dat we de `Equals` methode in onze `Point` klasse altijd gebruiken om 2 `Points` te vergelijken niets anders: we weten dan dat de `obj` parameter ook kan aanschouwd worden als een `Point`. En we kunnen dus de `obj` parameter casten naar een (tijdelijk) `Point`:

```
public override bool Equals (object obj)
{
    Point tijdelijk= (Point)obj;
    if(X=tijdelijk.X && Y= tijdelijk.Y)
        return true;
    return false
}
```

We maken dus een tijdelijke variabele aan en zetten daarin het adres van de binnenkomende `obj` object. Als we namelijk naar `obj` zouden gaan zien in het geheugen (de heap) dan zouden we daar effectief een object van het type `System.Object` zien staan, maar vlak erachter staan de `X` en de `Y`-waarden. We zeggen dus eigenlijk: *"beste variabele tijdelijk, jij verwijst naar het geheugenadres van de obj parameter, maar ik weet dat die obj-parameter van het type Point is... Kijk dus maar verder in het geheugen en beschouw de obj parameter als een Point.*

We kunnen dus nu obj vergelijken met het punt zelf. Wanneer we dus schrijven: `if(punt1.Equals(punt2))` .

Dan zal de `Equals` methode op het `punt1` uitgevoerd worden. `x` en `y` bevatten met andere woorden de waarden van `punt1`. `Punt2` in dit geval zal als object `obj` de methode binnengaan.

Verbetering: is to the rescue

Het is natuurlijk gevvaarlijk om te veronderstellen dat we de `Equals` methode op de ``Point'' klasse altijd correct gaan gebruiken. Stel dat we zouden schrijven:

```
if(punt1.Equals("mijn locatie is hier"))
```

dan gaan we dus proberen om een Point te vergelijken met een `string` . We vergelijken appelen met peren.

Het is dus aan te raden om een controle(s) in te bouwen in de `Equals` methode: voor we `obj` gaan casten naar een `Point` gaan we eerst controleren of deze wel kan gecast worden.

Met andere woorden: we gaan vragen wat het echte type van obj is.

```
public override bool Equals (object obj)
{
    if(obj is Point)
    {
        Point tijdelijk= (Point)obj;
        if(X==tijdelijk.X && Y== tijdelijk.Y)
            return true;
    }
    return false
}
```

Enkel indien het type van `obj` dezelfde is als het type van het object waarbinnen we de ``Equals'' methode aanroepen zullen we verder werken. Als we deze controle niet zouden doen dan zou deze lijn:

```
Point tijdelijk = (Point)obj;
```

proberen om "mijn locatie is hier" om te zetten (casten) naar een Point, wat zou resulteren in een `InvalidOperationException` .

Checken op null

Als finale check moeten we ook controleren of we geen null-object als parameter aan de methode meegeven. Mogelijk probeer je een bestaand object te vergelijken met een nog niet geïnstantieerd object en dan krijgen we een `NullReferenceException` .

Onze finale `Equals` methode wordt:

```
public override bool Equals (object obj)
{
    if(obj != null)
    {
        if(obj is Point)
        {
            Point tijdelijk= (Point)obj;
            if(X==tijdelijk.X && Y== tijdelijk.Y)
                return true;
        }
    }
    return false
}
```

Why should I care



Let's be honest. Als je aan dit punt en geen flauw benul hebt waarom je in godsnaam je hier ies van moet aantrekken, wel dan wordt het dringend tijd om deze cursus van voor naar achter, links naar rechts en onder tot boven terug door te nemen ;).

Dierentuin advanced

Voeg een filter toe aan de dierentuin applicatie uit een eerder hoofdstuk. Namelijk: e. Filter praten: er wordt gevraagd welke dieren moeten praten (Koe, Slang of Varken) vervolgens zullen enkel die dieren praten (tip: "is" operator uit les van gisteren)

All In One

Volgende hoofdstukken bevatten grotere projecten waarin wordt getracht meerdere technieken uit de vorige hoofdstukken te combineren.
Het doel van dit hoofdstuk is tweeledig:

1. Op een andere manier tonen hoe specifieke C# elementen in een 'realistische' programma's kunnen gebruikt worden
2. Aantonen dat, indien je tot hier bent geraakt, je al een aardig hoop skills hebt om grote, complexe applicaties te maken die verder gaan dan de standaard oefeningen die je na ieder hoofdstuk hebt gemaakt.

Volgt nu een beschrijving van de belangrijkste technieken die je in de projecten hierna zal tegenkomen:

- [OO Textbased Game](#): Klassen en objecten, Enum, List<>, Compositie
- [War Simulator](#): Klassen en objecten, Overerving, Abstract, List<>, Polymorfisme, Compositie, Interfaces
- [Map Maker](#): Klassen en objecten, Overerving, Abstract, List<>, Polymorfisme, Compositie, Interfaces

Text-gebaseerd spel mbv OO technieken

De prijs voor meest sexy titel gaan we niet winnen. Maar naar de aloude traditie van de klassieke tekst-gebaseerde adventure-games (zie [hier](#)) zullen we een eenvoudig object georiënteerd framework maken dat ons toelaat snel onze eigen games te maken. De nadruk van dit artikel ligt daarbij niet tot het creëren van een perfecte imitatie, maar wel om aan te tonen dat met een beetje object georiënteerde inzichten we tot zeer propere, herbruikbare én onderhoudbare code kunnen komen.

We maken uiteraard het spel in een Console-applicatie.

Main

We willen de `Main()` methode van Program.cs zo leeg mogelijk laten. Daarom zullen we de meeste functionaliteit verpakken in een klasse GameManager. Het enige dat we dan nog hoeven te doen in onze main is een loop starten die steeds 3 zaken zal doen:

1. Huidige locatie beschrijven
2. Aan gebruiker tonen welke acties hij kan uitvoeren
3. Gewenste actie van gebruiker verwerken en uitvoeren

In code behelst dit:

```
static void Main(string[] args)
{
    Console.WriteLine("Welkom bij AP Adventure. Een avontuur voor moedige en minder moedige studenten. Ben je er klaar voor?");

    GameManager theGame= new GameManager();

    //Start gameloop
    while(!theGame.Exit)
    {
        //Beschrijf kamer
        theGame.DescribeLocation();
        //Toon acties
        theGame.ToonActies();
        //Lees actie uit
        theGame.VerwerkActie();
    }
}
```

Een bool property `Exit` binnen het GameManager object zal ons toelaten om de loop te stoppen wanneer het spel wordt beëindigd.

GameObject

Doorheen de verschillende locaties zullen elementen te vinden zijn. We beschrijven deze als GameObjects:

```
class GameObjects
{
    public string Name { get; set; }
    public string Description { get; set; }

    public void Describe()
    {
        Console.WriteLine(Name+" "+Description+".");
    }
}
```

Location

De gebruiker kan van locatie naar locatie gaan. Een locatie bestaat uit een aantal zaken:

- Een beschrijving en titel
- Een lijst van GameObjecten (items) die zich op deze locatie bevinden
- Een lijst van Exits, namelijk de richtingen waar de gebruiker naar toe kan gaan die aansluiten op een andere locatie
- Een eerste versie van onze locatie klasse is dan:

```

class Location
{
    public Location()
    {
        Exits = new List();
        ObjectsInRoom= new List();
    }
    public string Title { get; set; }
    public string Description { get; set; }

    public List Exits { get; set; }
    public List ObjectsInRoom { get; set; }

    public void Describe()
    {
        Console.WriteLine(Title + "\n*****");
        Console.WriteLine(Description );
        Console.WriteLine("Voorts zie je ook nog:");
        foreach (var objects in ObjectsInRoom)
        {
            objects.Describe();
        }
        Console.WriteLine("\n*****");
    }

    //...
}

```

Merk op dat we gebruik maken van `List` ipv arrays.

Exit

Iedere exit in een locatie definieert minstens 2 zaken:

- De richting waar deze uitgang zich bevindt (Noord, Oost,Zuid, West)
- Een referentie naar het locatie-object waar deze uitgang toegang tot verschafft

We krijgen dus al:

```

class Exit
{
    public Exit()
    {
        NeedsObject= new List<GameObjects>();
    }
    public Directions ExitDirection { get; set; }
    public Location GoesToLocation { get; set; }

    //...
}

```

Waarbij `Directions` een eigen gemaakt enum-type is:

```

enum Directions
{
    North, South, West, East
}

```

Van locatie veranderen

Binnen de locatie klasse voegen we een methode toe die de GameManager kan gebruiken om te weten te komen naar welke locatie de gebruiker gaat. De methode aanvaardt een `Direction` (i.e. de richting waarin de gebruiker wenst te gaan) en zal een referentie naar het location-object teruggeven waarnaar de gebruiker zal bewegen. Indien de richting waarin hij wenst te bewegen niet geldig is dan tonen we dit op het scherm:

```

public Location GetLocationOnMove(Directions direction)
{
    foreach (var exit in Exits)
    {
        if (exit.ExitDirection == direction)

```

```

    {
        return exit.GoesToLocation;
    }
}
Console.WriteLine("Dat is geen geldige richting");
return this;
}

```

Wanneer dus een exit wordt gevonden in de Exits lijst die voldoet aan de meegegeven `Direction` dan geven we een referentie terug naar de bijhorende locatie (`GoesToLocation`). Wordt er geen exit gevonden en bereiken we dus het einde van de foreach lus dan verschijnt de tekst op het scherm en geven we een referentie terug naar de huidige locatie.

GameObjects als vereisten om exit te gebruiken

Stel nu dat we soms willen dat een bepaalde locatie pas bereikt kan worden indien de gebruiker reeds een bepaald `GameObject` in zijn bezit heeft. Hiervoor moeten we 2 zaken aanpassen:

- We beschrijven in de `Exit` klasse welk object(en) nodig zijn om deze exit te mogen gebruiken
- We controleren of de speler het `GameObject` heeft wanneer deze naar een nieuwe locatie wil gaan mbv de `GetLocationOnMove()` methode.

De nieuwe, volledige `Exit` klasse wordt dan:

```

class Exit
{
    public Exit()
    {
        NeedsObject= new List<GameObjects>();
    }
    public Directions ExitDirection { get; set; }
    public Location GoesToLocation { get; set; }

    public List<GameObjects> NeedsObject { get; set; }

    public bool TestPassCondition(List<GameObjects> playerInventory)
    {
        int passCount = 0;
        for (int i = 0; i < NeedsObject.Count; i++)
        {
            if (playerInventory.Contains(NeedsObject[i]))
                passCount++;
        }

        if (passCount == NeedsObject.Count)
            return true;
        else
        {
            return false;
        }
    }
}

```

In deze ietwat knullige code tellen we dus of de speler alle GameObjecten in z'n inventory heeft (`playerInventory`) nodig om deze exit te gebruiken.

Deze methode `TestPassCondition` gebruiken we nu in de `GetLocationOnMove()`-methode in de `Location` klasse om te bepalen of de exit mag gebruikt worden. De methode wordt dan:

```

public Location GetLocationOnMove(Directions direction, List<GameObjects> playerInventory )
{
    foreach (var exit in Exits)
    {
        if (exit.ExitDirection == direction)
        {

            if(exit.TestPassCondition(playerInventory))
                return exit.GoesToLocation;
            else
            {
                Console.WriteLine("Je kan hier niet langs (je hebt niet alle vereiste items).");
                return this;
            }
        }
    }
}

```

```

        }
    }
}
Console.WriteLine("Dat is geen geldige richting");
return this;
}

```

GameManager

Rest ons nu enkel nog de `GameManager` klasse te maken. Ruw gezien is deze als volgt:

```

class GameManager
{
    public GameManager()
    {
        InitGame();
    }
    private Location currentLocation = null;
    public bool Exit { get; set; }

    public void DescribeLocation()
    {
        //...
    }

    public void VerwerkActie()
    {
        //...
    }

    public void ToonActies()
    {
        //...
    }

    private List<Location> GameLocation = new List<Location>();
    private List<GameObjects> Objects = new List<GameObjects>();
    private List<GameObjects> playerInventory= new List<GameObjects>();
    private void InitGame()
    {
        //...
    }
}

```

Wat ogenblikkelijk opvalt zijn:

- De 3 publieke methoden `DescribeLocation`, `VerwerkActie` **en** `ToonActies`
- Een private field `currentLocation` die een referentie bijhoudt naar de huidige locatie van de speler
- 3 lijsten met daarin de objecten die de speler heeft (`playerInventory`), alle objecten in het spel (`Objects`) en alle locaties in het spel (`GameLocation`)
- Een `InitGame()` methode waarin we alle gameobjecten, exits en locaties zullen aanmaken bij aanvang van het spel
- Een bool `Exit` zodat de externe gameloop weet wanneer het spel gedaan is

We zullen nu de afzonderlijke methoden invullen:

DescribeLocation()

```

public void DescribeLocation()
{
    this.currentLocation.Describe();
}

```

VerwerkActie()

```

string actie = Console.ReadLine();
bool error = false;
if (actie == "n")
    currentLocation = currentLocation.GettLocationOnMove(Directions.North, playerInventory);
else if (actie == "o")

```

```

        currentLocation = currentLocation.GettLocationOnMove(Directions.East, playerInventory);
    else if (actie == "w")
        currentLocation = currentLocation.GettLocationOnMove(Directions.West, playerInventory);
    else if (actie == "z")
        currentLocation = currentLocation.GettLocationOnMove(Directions.South, playerInventory);
    else if (actie == "e")
        Exit = true;
    else
    {
        error = true;
    }
    Console.Clear();
    if(error)
        Console.WriteLine("Onbekende actie");

```

We laten de speler dus toe door `n,o,w,z` in te typen dat gecontroleerd wordt naar welke nieuwe locatie zal gegaan worden. We passen hierbij de `currentLocation` property van de GameManager aan naar de, al dan niet nieuwe, locatie.

ToonActies()

```

public void ToonActies()
{
    Console.WriteLine("Mogelijke acties: (typ bijvoorbeeld n indien u naar het noorden wil)");
    Console.WriteLine("n= noord");
    Console.WriteLine("o= oost");
    Console.WriteLine("z= zuid");
    Console.WriteLine("w= west");

    Console.WriteLine("e=exit");
}

```

InitGame()

In deze methode definiëren nu de volledige spelinhoud. Wil je dus bijvoorbeeld dit spel uitbreiden met extra kamers en objecten, dan doe je dat in deze methode. Ter illustratie tonen we eerst hoe we 2 locaties aanmaken en deze aan elkaar hangen mbv de Exits (waarbij kamer één zich ten zuiden van kamer 2 bevindt)

```

private void InitGame()
{
    //Maak Locaties
    Location l1 = new Location()
    {
        Title = "De poort",
        Description =
            "Je staat voor een grote grijze poort die op een kier staat. Rondom je is prikkeldraad, je kan enkel naar het noorden, door de poort gaan. ";
    };

    Location l2 = new Location()
    {
        Title = "Receptie",
        Description =
            "De receptie....veel blijft er niet meer over van wat eens een bruisende omgeving was. Hier en daar zie je skeletten van , waarschijnlijk, enkele studenten. Een grote poort staat op een kier naar het zuiden. Je ziet twee deuren aan de westelijke en noordelijke zijde."
    };

    //Place exits
    l1.Exits.Add(new Exit() { ExitDirection = Directions.North, GoesToLocation = l2 });

    l2.Exits.Add(new Exit() { ExitDirection = Directions.South, GoesToLocation = l1 });

    //Voeg locatie toe
    GameLocation.Add(l1);
    GameLocation.Add(l2);

    currentLocation = l1;
}

```

Vergeet niet op het einde de 2 kamers toe te voegen aan de `GameLocation` lijst van de GameManager, alsook in te stellen wat de beginkamer is.

GamelInit met GameObject als conditie om kamer in te kunnen

Stel dat we even later in een kamer een sleutel plaatsen die als conditie dient om een andere kamer te kunnen openen. We schrijven dan in de `GameInit()` methode:

```
Location l6 = new Location()
{
    Title = "Gang",
    Description =
        "Een brede gang waar makkelijk 5 mensen schouder aan schooulder door kunnen. Zowel in het oosten als het westen zie je een deur."
};

Location l7 = new Location()
{
    Title = "Computerruimte",
    Description =
        "Eindelijk; je hebt het gehaald. De plek waar iedereen naar toe wil: het computerlabo!"
};

//Place objects in rooms
GameObjects keyto17 = new GameObjects() { Description = "Verroest en groot", Name = "Sleutel" };
15.ObjectsInRoom.Add(keyto17);
//...

16.Exits.Add(new Exit() { ExitDirection = Directions.West, GoesToLocation = 14, NeedsObject= new List<GameObjects>(){keyto17}});
16.Exits.Add(new Exit() { ExitDirection = Directions.East, GoesToLocation = 17 });

//Voeg locatie toe
//...
GameLocation.Add(16);
GameLocation.Add(17);
```

Een volledige GamelInit ter illustratie

We hebben nu de belangrijkste onderdelen geschreven. We tonen daarom een iets uitgebreider spel, demo zeg maar, waarin we alles gecombineerd in actie zien:

```
private void InitGame()
{
    //Maak Locaties
    Location l1 = new Location()
    {
        Title = "De poort",
        Description =
            "Je staat voor een grote grijze poort die op een kier staat. Rondom je is prikkeldraad, je kan enkel naar het noorden, door de poort gaan."
    };

    Location l2 = new Location()
    {
        Title = "Receptie",
        Description =
            "De receptie....veel blijft er niet meer over van wat eens een bruisende omgeving was. Hier en daar zie je skeletten van , waarschijnlijk, enkele studenten. Een grote poort staat op een kier naar het zuiden. Je ziet twee deuren aan de westelijke en noordelijke zijde."
    };

    Location l3 = new Location()
    {
        Title = "Koffieruime",
        Description =
            "Je staat in de koffieruimte achter de receptie. Menig pralinetje is hier vroeger met veel gusto opgesmikkeld. Een lege pralinedoos is het enige bewijs dat het hier ooit gezellig was. Een deur is de enige uitgang uit deze kamer naar het oosten."
    };

    Location l4 = new Location()
    {
        Title = "Tuin",
        Description =
            "Het is duidelijk herfst. Een kale boom en vele bruine bladeren op de grond...mistroosteriger kan eigenlijk niet. Je ziet een deur in het zuiden en in het westen en een grote klapdeur naar het noorden."
    };
}
```

```

Location l5 = new Location()
{
    Title = "Cafetaria",
    Description =
        "Ooit was dit een bruisende locati: veel eten, geroezemoes en licht door de grote ruiten. Nu enkel stof en lege tafel. Enkel een klapdeur is zichtbaar naar het zuiden."
};

Location l6 = new Location()
{
    Title = "Gang",
    Description =
        "Een brede gang waar makkelijk 5 mensen schouder aan schouder door kunnen. Zowel in het oosten als het westen zie je een deur.."
};

Location l7 = new Location()
{
    Title = "Computerraumte",
    Description =
        "Eindelijk; je hebt het gehaald. De plek waar iedereen naar toe wil: het computerlabo!"
};

//Place objects in rooms
GameObjects keyto17 = new GameObjects() { Description = "Verroest en groot", Name = "Sleutel" };
l5.ObjectsInRoom.Add(keyto17);

//Place exits
l1.Exits.Add(new Exit() { ExitDirection = Directions.North, GoesToLocation = l2 });

l2.Exits.Add(new Exit() { ExitDirection = Directions.South, GoesToLocation = l1 });
l2.Exits.Add(new Exit() { ExitDirection = Directions.West, GoesToLocation = l3 });
l2.Exits.Add(new Exit() { ExitDirection = Directions.North, GoesToLocation = l4 });

l3.Exits.Add(new Exit() { ExitDirection = Directions.East, GoesToLocation = l2 });

l4.Exits.Add(new Exit() { ExitDirection = Directions.South, GoesToLocation = l2 });
l4.Exits.Add(new Exit() { ExitDirection = Directions.West, GoesToLocation = l6 });
l4.Exits.Add(new Exit() { ExitDirection = Directions.North, GoesToLocation = l7 });

l5.Exits.Add(new Exit() { ExitDirection = Directions.South, GoesToLocation = l4 });

l6.Exits.Add(new Exit() { ExitDirection = Directions.West, GoesToLocation = l4, NeedsObject= new List<GameObjects>(){keyto17} });
l6.Exits.Add(new Exit() { ExitDirection = Directions.East, GoesToLocation = l7 }); //needs key condition

l7.Exits.Add(new Exit() { ExitDirection = Directions.East, GoesToLocation = l6 }); //Winning room

//Voeg locatie toe
GameLocation.Add(l1);
GameLocation.Add(l2);
GameLocation.Add(l3);
GameLocation.Add(l4);
GameLocation.Add(l5);
GameLocation.Add(l6);
GameLocation.Add(l7);

currentLocation = l1;
}

```

What's missing? Veel!

Maar een eerste uitdaging zou kunnen zijn: hoe kunnen we de speler objecten van de grond laten oprapen en in de inventaris plaatsen?
Dat raadsel laten we aan jou over over om op te lossen!

TODO: Under construction

Mapmaker "all-in-one-project"

Volgende hoofdstuk toont het gebruik van de belangrijkste OO concepten in een applicatie waarmee we een huis-plattegrond kunnen visualiseren. Het doel van dit hoofdstuk is zoveel mogelijk elementen van de voorbije hoofdstukken te integreren tot een groter werkend geheel.

Abstracte klasse

Eerst definiëren we een kleine hulpklasse Point die een punt in de ruimte voorstelt. We kunnen deze klasse ook gebruiken om een vector voor te stellen:

```
public class Point
{
    public Point(int inx, int iny)
    {
        x = inx;
        y = iny;
    }

    public int x
    {
        get{return x;}
        set{x=value;;}
    }
    public int y
    {
        get{return y;}
        set{y=value;;}
    }
}
```

We maken nu een abstracte klasse MapObject, die we vervolgens zullen gebruiken om over te erven zodat nieuwe klassen aangemaakt kunnen worden.

```
abstract public class MapObject
{
    private Point location;
    private double price ;
    private char drawChar;

    //Teken object in de console
    public abstract void Paint();
}
```

De variabele Price zal de prijs van het object bevatten, zodat we vlot kunnen berekenen wat de totale kostprijs van onze kaart zal zijn. Location bevat de coördinaten (x,y) waar het object in de console zal getekend moeten worden. drawChar geeft aan met welk karakter het item moet getoond worden.

Belangrijk: Merk op dat deze klasse minimaal is en allerlei essentiële zaken mankeert, zoals minstens een default constructor etc.

Indien je dit project dus in de praktijk wenst te gebruiken dan zal je nog zelf de nodige properties (of get/set-methoden, naar keuze) waarmee je toegang krijgt tot Location, Price krijgt en ook drawChar methode moeten schrijven.

Overerving

We maken de `MapObject` klasse expres abstract, we willen voorkomen dat deze klasse rechtstreeks als object in het programma kan gebruikt worden.

Laten we nu een nieuwe klasse aanmaken dat overerft van de abstract klasse `MapObject`

```
public class WallElement: MapObject
{
    public override void Paint()
    {}
```

```
}
```

De methode van Paint moeten we verplicht overiden (daar ze abstract was in de base klasse), voorts is het aan te raden om een default constructor te maken. De Paint-methode bevat zeer eenvoudigweg volgende 2 lijntjes code:

```
ConsoleCursorPosition(Location.x, Location.y);
Console.Write(DrawChar);
```

Elementen op het scherm

We kunnen nu in ons hoofdprogramma (main-methode) al direct elementen op het scherm brengen met bijvoorbeeld volgende code:

```
WallElement steen1= new WallElement();
steen1.Paint();
```

Dit geeft, als je een default constructor hebt gemaakt die automatisch ieder object op locatie (1,1) zet,, een sterretje op positie (1,1) op het scherm.

We zouden dus nu bijvoorbeeld meerdere stenen kunnen plaatsen (met verschillende prijs, naargelang de soort) en dan de totaalprijs opvragen.

Grotere objecten

We hebben nu een basis om andere zaken te maken. Stel dat we grotere objecten op het scherm wensen. We zouden dan kunnen definiëren dat de variabele Location het punt linksboven van het object bepaald. Volgende nieuwe object erft over van de MapObject en geeft een grotere figuur weer (vierkant, maar je kan natuurlijk je fantasie de vrije loop laten gaan):

```
class FurnitureElement: MapObject
{
    private int unitSize;
    public int UnitSize
    {
        get { return unitSize; }
        set { if (value > 0) unitSize = value; }
    }

    public override void Paint()
    {
        for (int i = Location.x; i < Location.x + UnitSize; i++)
        {
            for (int j = Location.y; j < Location.y + UnitSize; j++)
            {
                if (i < Console.WindowWidth && j < Console.WindowHeight)
                {
                    ConsoleCursorPosition(i, j);
                    Console.Write(DrawChar);
                }
            }
        }
    }
}
```

We kunnen dan eenvoudig weg allerlei meubels definiëren, zoals een zetel:

```
class ZetelElement: FurnitureElement
{
    public ZetelElement()
    {
        Price = 100;
        DrawChar = '+';
        UnitSize = 2;
    }
}
```

Of als je de zetel anders wil getekend zien (geen rechthoek bijvoorbeeld, maar iets dat meer op zetel trekt, dan voeg je nog volgende code toe:

```
public override void Paint()
```

```
{
    //Code om complexere zetel op scherm te tonen
}
```

Polymorfisme

We kunnen nu ongelooflijk veel objecten op het scherm tonen (laten we veronderstellen dat je een overloaded constructor telkens hebt geschreven), met behulp van een List-object, als volgt:

```
List<MapObject> allObjects= new List<MapObject>(); //lang leve polymorfisme

//Muurtje
for (int i = 0; i < 10 ; i++)
{
    Point tempLoc= new Point(2+i,3);
    WallElement tempWall= new WallElement(tempLoc,'=',10.0);
    allObjects.Add(tempWall);
}

//Zetel
allObjects.Add(new ZetelElement(new Point(6,8), 3 ));

//Teken alle objecten
for (int i = 0; i < allObjects.Count; i++)
{
    allObjects[i].Paint();
}
```

Dankzij polymorfisme kunnen we alle objecten die overgeërfd zijn van MapObject in de MapObject-list plaatsen. Wanneer we dan Paint aanroepen gebruiken we de implementatie van het object zelf indien aanwezig.



Maken van een grafisch menu

Het maken van een semi-grafisch menu is verrassend eenvoudig.

Menu Tekenen

Volgende klasse toont een kadertje met wat tekst in:

```
public class Menu
{
    public Menu()
    {}

    public void ShowMenu()
    {
        //Tekenen
        TekenBalk(1);
        TekenOpties(2);
        TekenBalk(5);
    }

    private void TekenBalk(int hoogte)
    {
        for (int i = 0; i < Console.WindowWidth; i++)
        {
            Console.SetCursorPosition(i, hoogte);
            Console.Write('*');
        }
    }

    private void TekenOpties(int hoogte)
    {
        Console.SetCursorPosition(5,hoogte);
        Console.Write("A) Voeg zetel toe op willekeurige locatie");
    }
}
```

```

        Console.SetCursorPosition(5, hoogte+1);
        Console.WriteLine("B) Beweeg kaart naar beneden");
        Console.SetCursorPosition(5, hoogte+2);
        Console.WriteLine("Wat wilt u doen?...");
    }
}

```

Je kan dit dan als volgt oproepen in je main:

```

Menu menu= new Menu();
menu.ShowMenu();

```

Tekstverwerken van Menu

We geven onze lijst van objecten mee aan ons Menu zodat het Menu object nieuwe zaken aan de map kan toevoegen:

```

public void GetInput(List<MapObject> list)
{
    string input=Console.ReadLine();
    if(input=="A" || input=="a")
    {
        //Voeg randomzetel toe
    }
    if (input == "B" || input == "b")
    {
        //Beweeg kaart naar beneden
    }
}

```

We kunnen dan in de main volgende code plaatsen die constant het scherm hertekent en telkens op input van de gebruiker wacht:

```

List<MapObject> allObjects = new List<MapObject>();
Menu menu= new Menu();
do
{
    menu.ShowMenu();
    menu.GetInput(allObjects);
    Console.Clear();
    //Teken alle objecten
    for (int i = 0; i < allObjects.Count; i++)
    {
        allObjects[i].Paint();
    }
} while (true);

```

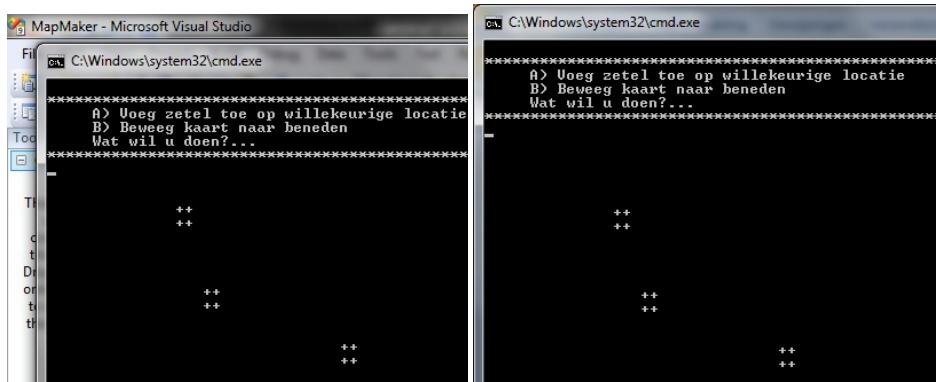
Map verplaatsen

De map verplaatsen is wederom verrassend eenvoudig. Stel dat je je map naar beneden wenst te verplaatsen als de B wordt ingedrukt; Je update gewoon de locatie van ieder object waarbij de y-positie gewoon met 1 wordt verhoogd:

```

if (input == "B" || input == "b")
{
    //Beweeg kaart naar beneden
    for(int i=0;i<list.Count;i++)
    {
        list[i].Location = new Point(list[i].Location.x, list[i].Location.y + 1);
    }
}

```



Composiet-klassen

Voorts kunnen we bijvoorbeeld nu meerdere klassen aanmaken (tafels, stoelen, deuren, etc) en dan een composiet-klasse aanmaken die bijvoorbeeld een volledig salon beschrijft, de code zou er dan als volgt kunnen uitzien:

```
public class SalonElement: MapObject
{
    private List<MapObject> elementen= new List<MapObject>();

    public SalonElement(Point salonLoc)
    {

        elementen.Add(new ZetelElement(new Point(2, 2), 3, '+'));
        elementen.Add(new ZetelElement(new Point(5, 9), 3, '+'));

        Location = salonLoc;
    }

    public override void Paint()
    {
        for (int i = 0; i < elementen.Count; i++)
        {
            elementen[i].Paint();
        }
    }
}
```

Merk op dat we rekening moeten houden met het feit dat de locatie van het salon het punt linksboven is, en dat dus de nieuwe locaties van de zetels vanaf dit punt hun oorsprong hebben. Althans dat willen we.. Als we in het main-programma dan schrijven:

```
SalonElement salon1= new SalonElement(new Point(6,5));
salon1.Paint();
```



Dan verschijnen onze zetels wel, maar niet op de locatie zoals we wilden (nu verschijnen de zetels op locatie (2,2) en (5,9), terwijl we liever hebben dat ze verschijnen op (2+6, 2+5) en (5+6, 9+5), dus rekening houdende met de locatie van het salon zelf

Interface

We willen nu ervoor zorgen dat wanneer we volgende code schrijven, dat ook alle elementen van het Salon mee verhuizen naar de nieuwe locatie:

```

List<MapObject> allObjects = new List<MapObject>();
allObjects.Add(new SalonElement(new Point(5, 5)));
allObjects[0].Paint();

//Verplaats salon
allObjects[0].Location= new Point(10,10);
allObjects[0].Paint();

```

Echter, dat gebeurt niet. De oplossing is een gevorderd principe, maar een beetje dat hopelijk het voordeel van een Interface laat zien.

We leggen een nieuwe interface **IComposite** vast die iedere composietklasse moet implementeren:

```

interface IComposite
{
    void UpdateElements(Point offset);
}

```

Ons **SalonElement** wordt dan volgende aanpassing:

```

public class SalonElement: MapObject,IComposite
{
    private List<MapObject> elementen= new List<MapObject>();

    public SalonElement(Point salonLoc)
    {
        ...
    }

    public override void Paint()
    {
        ...
    }

    public void UpdateElements(Point offset)
    {
        ...
    }
}

```

De **UpdateElements** methode zou er dan als volgt kunnen uitzien:

```

for (int i = 0; i < elementen.Count; i++)
{
    Point elementLoc = elementen[i].Location;
    elementLoc.x += offset.x;
    elementLoc.y += offset.y;
    elementen[i].Location = elementLoc;
}

```

Is/as

Tekens we dus **UpdateElements** aanroepen dan worden alle elementen die bij het object horen ook geüpdatet.

Nu rest ons nog één aanpassing, dat is ervoor zorgen dat deze methode ook effectief telkens wordt aangeroepen. De methode moet aangeroepen worden telkens we een aanpassing aan de **Location** van het **SalonElement** doen. Hierbij controleren we eerst of de locatie überhaupt al geïnitialiseerd is (anders is deze waarde gelijk aan 'null'). Vervolgens berekenen we de offset, dit is het verschil tussen de huidige en de nieuwe locatie van de composietklasse. Daar **Location** bij **MapObject** hoort, moeten we dus in die klasse een aanpassing doen. We bereiden daarom de **Location**property uit als volgt:

```

public Point Location
{
    get { return location; }
    set
    {
        Point prevloc = location;
        Point offset = new Point(1, 1);
        if (location != null)
        {

            offset.x = value.x - prevloc.x;
            offset.y = value.y - prevloc.y;
        }
    }
}

```

```

        }

        location = value;
        if (this is IComposite)
        {
            IComposite obj = this as IComposite;
            obj.UpdateElements(offset);
        }

    }
}

```

Deze code kan misschien wat toelichting gebruiken:

- Telkens we de set aanroepen van Location (dus in bijvoorbeeld allObjects[0].Location= new Point(10,10);) dan veranderen sowieso de locatie van het object naar de nieuwe waarde.
- We bewaren de huidige locatie zodat we de offset kunnen berekenen.
- Indien er een 'huidige locatie' is (location!=null) dan berekenen we de offset in de x en de y richting.
- Nu passen we de locatie van de composietklasse aan.
- Vervolgens kijken we of het object in kwestie (aangegeven met this, daar we in het object zelf zitten) de IComposite interface 'heeft'.
- Als dit zo is dan zetten we het object even om naar een IComposite-object zodat we de UpdateElements()-methode kunnen aanroepen.

We kunnen nu dus zelfs een volledig Huis als klasse beschrijven en zo verschillende soorten huizen definiëren. Telkens we dan een huis verplaatsen dan verplaatst de hele inboedel mee.

Belangrijk: Het gebruik van de interface is hier louter illustratief. Dit probleem kan je beter oplossen door een CompositeElement klasse aan te maken die overerft van MapObject. Deze klasse bevat dan een lijst van elementen en een UpdateElements methode. In MapObject controleer je dan of een object van het type CompositeElement is (ipv IComposite)

Een nog betere oplossing is die waarbij je gewoon direct zegt dat MapObject een lijst van elementen kan bevatten. Als een MapObject exact 1 element in zijn lijst bevat dan is de werking dezelfde als ervoor, maar nu kunnen we dus zonder veel code aanpassingen ook composiet objecten aanmaken.

Think about it.

EAICT Coding guidelines

Naamgeving

- **Duidelijke naam:** de identifier moet duidelijk maken waarvoor de identifier dient. Schrijf dus liever `gewicht` of `leeftijd` in plaats van `a` of `meuh`.
- **Camel casing:** gebruik camel casing indien je meerdere woorden in je identfier wenst te gebruiken. Camel casing wil zeggen dat ieder nieuw woord terug met een hoofdletter begint. Een goed voorbeeld kan dus zijn `leeftijdTimDams` of `aantalLeerlingenKlas1EA`. Merk op dat we liefst het eerste woord met kleine letter starten.
- **Prefereer cijfers en letters:** gebruik geen liggende streepjes of andere karakters die geen cijfers of letters zijn.
- **Private kleine letter, public hoofdletter:** private variabelen starten met een kleine letter (bv. `pagesBook`), public variabelen met een grote letter (bv. `SizeBook`).
- **Methoden met hoofdletter:** methoden starten steeds met een hoofdletter (bv. `OpenDataBase`).
- **Geen afkortingen:** Schrijf `GetWindowSize` in plaats van `GetWinSz`.