9. Object Oriented Programming

Totnogtoe leerden we eigenlijk *gestructureerd programmeren* wat een programmeerparadigma is uit de jaren zestig. Hierbij schrijven we code gebruik makend van methoden, loops en beslissingsstructuren. Op zich blijft dit een erg nuttige manier van programmeren. Wanneer we echter bij complexere applicaties komen dan merken we dat met gestructureerd programmeren we redelijk snel tot minder intuïtieve en soms nodeloos complexe code aanbelanden.

Dat moet dus anders kunnen. Komt u binnen, **Object georiënteerd programmeren** (**OOP**). OOP is een manier van programmeren die voortbouwt op gestructureerd programmeren, maar die toelaat veel krachtigere applicaties te ontwikkelen.

Bij OOP draait alles rond **klassen en objecten** die intern nog steeds gestructureerde code zullen bevatten (loops, methoden en beslissingsstructuren), maar die onze code (hopelijk) een pak overzichtelijker en minder complex gaan maken. Dankzij OOP gaan we onze code meer modulair, leesbaarder en onderhoudsvriendelijker maken én tegelijkertijd zal ze veel krachtiger worden en daardoor complexere zaken eenvoudiger kunnen "oplossen".

> Hier zijn we weer!
>
> 
>
> Ik zet "oplossen" tussen aanhalingstekens. Net zoals alles binnen dit domein ben jij als programmeur uiteindelijk degene die het boeltje moet oplossen. Code, programmeerparadigma's en bibliotheken zijn niet meer dan nuttig gereedschap in jouw arsenaal van programmeertools. Als jij beslist om een hamer als zaag te gebruiken, tja, dan houd ik m'n hart vast voor het resultaat. Dit geldt ook voor de technieken die je nog in dit boek gaat leren: ze zijn "een tool", niets meer. Jij zal ze nog steeds zo optimaal mogelijk moeten leren gebruiken. Uiteraard is het doel van dit boek je zo duidelijk mogelijk het verschil én de bruikbaarheid van de verschillende nieuwe technieken aan te leren.

Toen C# werd ontwikkeld in 2001 was één van de hoofddoelen van de programmeertaal om *"een eenvoudige, moderne, objectgeoriënteerde programmeertaal voor algemene doeleinden"* te worden. **C# is van de grond af opgebouwd met het OOP programmeerparadigma als primaire drijfveer**.

Wanneer we nieuwe programma's in C# ontwikkelden dan zagen we hier reeds bewijzen van. Zo zagen we steeds het keyword `class` bovenaan staan, telkens we een nieuw project aanmaakten:

```
1  namespace WorldDominationTool
2  {
3      internal class Program
4      {
```

De klasse `Program` zorgt ervoor dat ons programma voldoet aan de C# afspraken die zeggen dat alle C# code in klassen moet staan.

Duizend mammoeten en sabeltandtijgers! Ik dacht dat ik nu wel mee zou zijn met alles wat C# me zou voorschotelen. Helaas, wolharige neushoorn-kaas, niet dus. Ik ga een voorspelling doen: van alle hoofdstukken in dit boek, wordt dit hoofdstuk hetgene waar je het meest je tanden op gaat stuk bijten. Hou dus vol, geef niet te snel op en kom geregeld hier terug. Succes gewenst!

Een wereld zonder OOP: Pong

Om de kracht van OOP te demonstreren gaan we een applicatie van lang geleden (deels) herschrijven gebruik makende van de kennis van gestructureerd programmeren. We gaan de arcadehal klassieker "Pong" namaken, waarbij we als doel hebben om een balletje alvast op het scherm te laten botsen. Een rudimentaire oplossing zou de volgende kunnen zijn:

```
1  Console.CursorVisible = false;
2  int balX = 20;
3  int balY = 20;
4  int VectorX = 2;
5  int VectorY = 1;
6  while (true)
7  {
8      //Xvector van richting veranderen aan de randen
9      if (balX + VectorX >= Console.WindowWidth || balX+VectorX < 0)
10     {
11         VectorX = -VectorX;
12     }
13     balX = balX + VectorX; //X positie updaten
14     //Yvector van richting veranderen aan de randen
15     if (balY + VectorY >= Console.WindowHeight || balY+VectorY < 0)
16     {
17         VectorY = -VectorY;
18     }
19     balY = balY + VectorY; //Y positie updaten
20     //Output naar scherm sturen
21     Console.SetCursorPosition(balX, balY);
22     Console.Write("O");
23     System.Threading.Thread.Sleep(50); //50 ms wachten
24     Console.Clear();
25 }
```

Hopelijk begrijp je deze code. Test ze maar eens in een programma. Zoals je zal zien krijgen we een balletje (`"O"`) dat over het scherm vliegt en telkens van richting verandert wanneer het aan de randen van het

applicatievenster komt. De belangrijkste informatie zit in de variabelen `balX`, `balY` die de huidige positie van het balletje bevatten. Voorts zijn ook `VectorX` en `VectorY` belangrijk: hierin houden we bij in welke richting (en met welke snelheid) het balletje beweegt (een zogenaamde bewegingsvector).

Extra balletjes?

Dit soort applicatie in C# schrijven met behulp van gestructureerde programmeer-concepten is redelijk eenvoudig. Maar wat als we nu 2 balletjes nodig hebben? Laten we arrays even links laten liggen en het gewoon eens naïef oplossen. Al na enkele lijnen kopiëren merken we dat onze code ongelooflijk rommelachtig gaat worden en we bijna iedere lijn moeten dupliceren:

```
Console.CursorVisible = false;
int balX = 20;
int balY = 20;
int vectorX = 2;
int vectorY = 1;

int bal2X = 10;
int bal2Y = 8;
int vector2X = 2;
int vector2Y = -1;

while (true)
{
    if (balX + vectorX >= Console.WindowWidth || balX+ vectorX < 0)
    {
        vectorX = -vectorX;
    }
    if (bal2X + vector2X >= Console.WindowWidth || bal2X + vector2X < 0)
    {
        vector2X = -vector2X;
    }

    balX = balX + vectorX;
    bal2X = bal2X + vector2X;
    //enzovoort
```

Bijna iedere lijn code moeten we verdubbelen. Arrays zouden dit probleem deels kunnen oplossen, maar we krijgen dan in de plaats de complexiteit van werken met arrays op ons bord, wat voor 2 balletjes misschien wat overdreven is én de code ook weer wat minder leesbaar maakt.

Een wereld met OOP: Pong

Uiteraard zijn we nu eventjes gestructureerd programmeren aan het demoniseren, dit is echter een bekend 21e eeuws trucje om je punt te maken.

Wanneer we Pong vanuit een OOP paradigma willen aanpakken dan is het de bedoeling dat we werken met klassen en objecten. Net zoals aan de start van dit boek ga ik je ook nu even in het diepe gedeelte van het bad gooien. Wees niet bang, ik zal je er tijdig uithalen (en je zal versteld staan hoeveel code je eigenlijk zult herkennen).

Om Pong in OOP te maken hebben we eerst een klasse nodig waarin we ons balletje gaan beschrijven, zonder dat we al een balletje hebben. En dat ziet er zo uit:

```
class Balletje
{
    //Eigenschappen
    public int X { get; set; }
    public int Y { get; set; }
    public int VectorX { get; set; }
    public int VectorY { get; set; }

    //Methoden
    public void Update()
    {
        if (X + VectorX >= Console.WindowWidth || X + VectorX < 0)
        {
            VectorX = -VectorX;
        }
        X = X + VectorX;


        if (Y + VectorY >= Console.WindowHeight || Y + VectorY < 0)
        {
            VectorY = -VectorY;
        }
        Y = Y + VectorY;
    }

    public void TekenOpScherm()
    {
        Console.SetCursorPosition(X, Y);
        Console.Write("O");
    }
}
```

De code voor een nieuwe klasse schrijf je best in een apart bestand in je project. Klik bovenaan in de menu balk op “Project” en kies dan “Add class…”. Geef het bestand de naam “Balletje.cs”.

Bijna alle code van zonet hebben we hier geïntegreerd in een `class Balletje`, maar er zit duidelijk een nieuw sausje over. Vooral aan het begin zien we onze 4 variabelen terugkomen in een nieuw kleedje, namelijk als eigenschappen oftewel properties (herkenbaar aan de `get` en `set` keyword, waarover later meer). Maar al bij al lijkt de code grotendeels op wat we al kenden. En dat is goed nieuws. OOP gooit de vorige hoofdstukken niet in de vuilbak, het gaat als het ware een extra laag over het geheel leggen. Let ook op het essentiële woordje `class` bovenaan, daar draait alles natuurlijk om: **klassen en objecten**.

Een klasse is een blauwdruk van een bepaalde soort 'dingen' of objecten. Objecten zijn de "echte" dingen die werken volgens de beschrijving van de klasse. Ja ik heb zonet 2x hetzelfde verteld, maar het is essentiëel dat je het verschil tussen de termen **klasse** en **object** goed begrijpt.

Laten we eens een **balletje-object** in het leven roepen. In de main schrijven we daarom dit:

```
1  Console.CursorVisible = false;
2  Balletje bal1 = new Balletje();
3  bal1.X = 20;
4  bal1.Y = 20;
5  bal1.VectorX = 2;
6  bal1.VectorY = 1;
```

Ok, interessant. Die `new` heb je al gezien wanneer je met `Random` ging werken en de code erna is ook nog begrijpbaar: we stellen eigenschappen van het nieuwe `bal1` object in. En nu komt het! Kijk hoe eenvoudig onze volledig `main` nu is geworden:

```
1  static void Main(string[] args)
2  {
3      Balletje bal1 = new Balletje();
4      bal1.X = 20;
5      bal1.Y = 20;
6      bal1.VectorX = 2;
7      bal1.VectorY = 1;
8
9      while (true)
10     {
11         bal1.Update();
12         bal1.TekenOpScherm();
13
14         System.Threading.Thread.Sleep(50);
15         Console.Clear();
16     }
17 }
```

De loopcode is herleid tot 2 aanroepen van **methoden op het `bal1` object**: `.Update()` en `.TekenOpScherm`.

Run deze code maar eens. Inderdaad, deze code doet exact hetzelfde als hiervoor. Ook nu krijgen we 1 balletje dat op het scherm over en weer botst.

En nu - abracadabra - kijk goed hoe eenvoudig onze code blijft als we 2 balletjes nodig hebben:

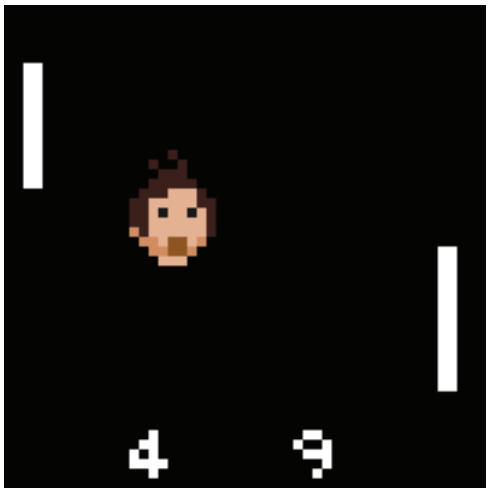
```
Console.CursorVisible = false;
Balletje bal1 = new Balletje();
bal1.X = 20;
bal1.Y = 20;
bal1.VectorX = 2;
bal1.VectorY = 1;

Balletje bal2 = new Balletje();
bal2.X = 10;
bal2.Y = 8;
bal2.VectorX = 2;
bal2.VectorY = -1;

while (true)
{
    bal1.Update();
    bal2.Update(); //zo simpel!
    bal1.TekenOpScherm();
    bal2.TekenOpScherm(); //wow, zooo simpel :)
    System.Threading.Thread.Sleep(50);
    Console.Clear();
}
```

Dit is de volledige code om 2 balletjes te hebben. Hoe mooi is dat?!

De kracht van OOP zit hem in het feit dat we de logica IN DE OBJECTEN ZELF plaatsen. De objecten zijn met andere woorden verantwoordelijk om hun eigen gedrag uit te voeren gebaseerd op externe impulsen en hun eigen interne toestand. In onze main zeggen we aan beide balletjes “update je zelf eens” , gevolgd door “teken je zelf eens”.



Een artistieke benadering van hoe Pong er vroeger uitzag.

Wanneer we 3 of meer balletjes zouden nodig hebben dan zullen we best arrays in de mix moeten gooien. Onze code blijft echter véél eenvoudiger én krachtiger dan wanneer we in het voorgaande enkel de kennis gebruikten die we totnogtoe hadden. Omdat we toch al in het diepe eind zitten, zal ik hier toch al eens tonen hoe we 100 balletjes op het scherm kunnen laten botsen (we gaan `Random` gebruiken zodat er wat willekeurigheid in de balletjes zit):

```
1  const int AANTAL_BALLETJES = 100;
2  Random r = new Random();
3  Balletje[] veelBalletjes = new Balletje[AANTAL_BALLETJES];
4  for (int i = 0; i < veelBalletjes.Length; i++) //balletjes aanmaken
5  {
6      veelBalletjes[i] = new Balletje();
7      veelBalletjes[i].X = r.Next(10, 20);
8      veelBalletjes[i].Y = r.Next(10, 20);
9      veelBalletjes[i].VectorX = r.Next(-2, 3);
10     veelBalletjes[i].VectorY = r.Next(-2, 3);
11 }
12
13 while (true)
14 {
15     for (int i = 0; i < veelBalletjes.Length; i++)
16     {
17         veelBalletjes[i].Update(); //update alle balletjes
18     }
19     for (int i = 0; i < veelBalletjes.Length; i++)
20     {
21         veelBalletjes[i].TekenOpScherm(); //teken alle balletjes
22     }
23     System.Threading.Thread.Sleep(50);
24     Console.Clear();
25 }
```

De reden dat we 2 loops gebruiken, in plaats van 1, is omdat we in de update fase eerst alle objecten willen updaten (soms ten opzichte van andere objecten) voor we alles terug op het scherm tekenen. Anders kan het zijn dat je vreemde effecten te zien krijgt als je bijvoorbeeld balletjes tegen elkaar wil laten wegbotsen.

Ok, zwem maar snel naar de kant. We gaan al het voorgaande van begin tot einde uit de doeken doen! Leg die handdoek niet te ver weg, we gaan hem nog nodig hebben.

Draai deze pagina pas om wanneer je uitgeslapen bent. Je opperste concentratie zal voor de volgende 2 pagina's vereist zijn!

9.1 Klassen en objecten

Een elementair aspect binnen OOP is het verschil begrijpen tussen een klasse en een object.

Wanneer we meerdere objecten gebruiken van dezelfde soort dan kunnen we zeggen dat deze objecten allemaal deel uitmaken van een zelfde klasse. **Het OOP paradigma houdt ook in dat we de echte wereld gaan proberen te modeleren in code**. OOP laat namelijk toe om onze code zo te structureren zoals we dat ook in het echte leven doen. Alles (objecten) om ons heen behoort tot een bepaalde klasse die alle objecten van dat type beschrijven.

Neem eens een kijkje aan een druk kruispunt waar fietsers, voetgangers, auto's en allerlei andere zaken samenkomen[1]. Het is een erg hectisch geheel, toch kan je alles dat je daar ziet *classificeren*. We zien bijvoorbeeld allemaal mens-objecten die tot de klasse van de Mens behoren, maar ook:

- Alle mensen hebben gemeenschappelijke eigenschappen (binnen deze beperkte context van een kruispunt): ze bewegen of staan stil (gedrag), ze hebben een bepaalde kleur van jas (eigenschap).
- Alle auto's behoren tot een klasse Auto. Ze hebben gemeenschappelijke zaken zoals: ze hebben een bepaald bouwjaar (eigenschap), ze werken op een bepaalde vorm van energie (eigenschap) en ze staan stil of bewegen (gedrag).
- Ieder verkeerslicht behoort tot de klasse VerkeersLicht.
- Fietsers behoren tot de klasse Fietser.

Definitie klasse en object

Volgende 2 definities druk je best af op een grote poster die je boven je bed hangt:

- **Een klasse** is als een **blauwdruk** (of prototype) dat het gedrag en toestand beschrijft van alle objecten van deze klasse.
- Een individueel **object** is een **instantie** van een klasse en heeft een eigen *toestand*, *gedrag* en *identiteit*.

Objecten zijn instanties met een eigen levenscyclus die wordt gekenmerkt door:

- **Gedrag**: deze wordt beschreven door de **methoden** in de klasse.
- **Toestand**: deze kan wijzigen door zijn eigen gedrag, of het gedrag van externe impulsen en wordt bepaald door **datavelden** die beschreven staan in de klasse (properties en instantievariabelen).
- **Identiteit** : een unieke naam van object zodat andere object ermee kunnen interageren.

Je zou dit kunnen vergelijken met het grondplan voor een huis dat tien keer in een straat zal gebouwd worden. Het plan is de *klasse*. De effectieve huizen die we, gebaseerd op dat grondplan, bouwen zijn de instanties of objecten van deze klasse en hebben elk een eigen toestand (ander type bakstenen, wel of geen zonnepannelen) en gedrag (rolluiken gaan open als de zon opkomt).

De klasse beschrijft het algemene **gedrag** van de individuele objecten. Dit gedrag wordt meestal bepaald door de interne staat van ieder object op zichzelf, de zogenaamde **eigenschappen**. Nemen we het

[1]Dit voorbeeld is gebaseerd op de inleiding van het inzichtvolle boek "Handboek objectgeoriënteerd programmeren" door Jan Beurghs (EAN: 9789059406476)

voorbeeld van de klasse Auto: de huidige snelheid van een individueel auto-object is mogelijks gebaseerd op het merk (eigenschap) van die auto, alsook welke energiebron (eigenschap) die auto heeft.

Voorts kunnen objecten ook beïnvloed worden door 'de buitenwereld': naast de interne staat van ieder object, leven de objecten natuurlijk in een bepaalde context, zoals een druk kruispunt. Andere objecten op dat kruispunt kunnen invloed hebben op wat een auto-object doet. Met andere woorden: we kunnen 'van buiten uit' vaak ook het gedrag en de interne staat van een object aanpassen. We hebben dit reeds zien gebeuren in het Pong-voorbeeld: de interne staat van ieder individueel balletjes-object is z'n positie alsook z'n richtingsvector. De buitenwereld, in dit geval onze `Main` methode kon echter de objecten manipuleren:

- Het gedrag van een balletje konden we aanpassen met behulp van de `Update` en `TekenOpScherm` methode.
- De interne staat via de eigenschappen die zichtbaar zijn aan de buitenwereld (dankzij het `public` keyword) .

Wanneer je later de specificaties voor een opdracht krijgt en snel wilt ontdekken wat potentiële klassen zijn, dan is het een goede tip om op zoek te gaan naar de zelfstandige naamwoorden (*substantieven*) in de tekst. Dit zijn meestal de objecten en/of klassen die jouw applicatie zal nodig hebben.

95% van de tijd zullen we in dit boek de voorgaande definitie van een klasse beschrijven, namelijk de blauwdruk voor de objecten die er op gebaseerd zijn. Je zou kunnen zeggen dat de klasse een fabriekje is dat objecten kan maken.
Echter, wanneer we het `static` keyword zullen bespreken gaan we ontdekken dat heel af en toe een klasse ook als een soort object door het leven kan gaan. Heel vreemd allemaal!

Abstractie principe

Een belangrijk concept bij OOP is het **Black-box** principe waarbij we de afzonderlijke objecten en hun werking als zwarte dozen gaan beschouwen.

Neem het voorbeeld van de auto: deze is in de echte wereld ontwikkeld volgens het blackbox-principe. De werking van de auto kennen tot in het kleinste detail is niet nodig om met een auto te kunnen rijden. De auto biedt een aantal zaken aan de buitenwereld aan (het stuur, pedalen, het dashboard), wat we de "**interface**" noemen, die je kan gebruiken om de interne staat van de auto uit te lezen of te manipuleren. Stel je voor dat je moest weten hoe een auto volledig werkte voor je ermee op de baan kon…

Binnen OOP wordt dit blackbox-concept **abstractie** genoemd. Het doel van OOP is andere programmeurs (en jezelf) zoveel mogelijk af te schermen van de interne werking van je klasse code. Vergelijk het met de methoden uit hoofdstuk 7: "if it works, it works" en dan hoef je niet in de code van de methode te gaan zien wat er juist gebeurt telkens je de methode wil gebruiken.

Kortom, hoe minder de buitenwereld moet weten om met een object te werken, hoe beter. Beeld je in dat je 10 lijnen code nodig had om een random getal te genereren. Niemand zou de klasse `Random` nog gebruiken. Dankzij de ontwikkelaar van deze klasse hoeven we maar 2 zaken te kunnen:

- Een `Random`-object aanmaken: `Random ranGen = new Random();`
- De `Next`-methode aanroepen om een getal uit het object te krijgen: `int getal = ranGen.Next();`. Wat er nu juist in die methode gebeurt boeit ons niet. It just works! Met dank aan abstractie en de kracht van OOP.

Objecten in de woorden van Steve Jobs

Steve Jobs, de oprichter van Apple, was een fervent fan van OOP. In een interview in 1994 voor het Rolling Stone magazine gaf hij volgende uitleg:

> "Objects are like people. They're living, breathing things that have knowledge inside them about how to do things and have memory inside them so they can remember things. And rather than interacting with them at a very low level, you interact with them at a very high level of abstraction, like we're doing right here.
>
> Here's an example: If I'm your laundry object, you can give me your dirty clothes and send me a message that says, "Can you get my clothes laundered, please." I happen to know where the best laundry place in San Francisco is. And I speak English, and I have dollars in my pockets. So I go out and hail a taxicab and tell the driver to take me to this place in San Francisco. I go get your clothes laundered, I jump back in the cab, I get back here. I give you your clean clothes and say, "Here are your clean clothes."
>
> You have no idea how I did that. You have no knowledge of the laundry place. Maybe you speak French, and you can't even hail a taxi. You can't pay for one, you don't have dollars in your pocket. Yet, I knew how to do all of that. And you didn't have to know any of it. All that complexity was hidden inside of me, and we were able to interact at a very high level of abstraction. That's what objects are. **They encapsulate complexity, and the interfaces to that complexity are high level**."

Objecten in de woorden van Bill Gates

En, omdat het vloeken in de kerk is om Steve Jobs in een C# boek aan het woord te laten, hier wat Microsoft-oprichter Bill Gates over OOP te zeggen had:

> "Another trick in software is to avoid rewriting the software by using a piece that's already been written, so called component approach which the latest term for this in the most advanced form is what's called Object Oriented Programming."

Ik zie dat je gereedsschapkist al aardig gevuld is. Zoals je misschien al gemerkt hebt aan deze sectie, zullen we vanaf nu ook geregeld minder "praktische" en eerder "filosofische" zaken tegenkomen. Maar wees gerust, je zal toch een grotere gereedsschapkist nodig hebben. Echter, net zoals een voorman niet alleen moet kunnen metsen en timmeren, maar ook stabiliteitsplannen begrijpen, zal ook jij moeten begrijpen wat de grotere ideeën achter bepaalde concepten zijn.

Zet nu je helm maar op, want in de volgende sectie gaan we wel degelijk onze handen lekker vuil maken!

9.2 OOP in C#

We kunnen in C# geen objecten aanmaken voor we een klasse hebben gedefinieerd dat de algemene eigenschappen (properties én instantievariabelen) en gedrag (methoden) beschrijft van die objecten.

Klasse maken

Een klasse heeft minimaal de volgende vorm:

```
1  class ClassName
2  {
3
4  }
```

De naam die je een klasse geeft moet voldoen aan de identifier regels uit hoofdstuk 2. Het is echter een goede gewoonte om **klassenamen altijd met een hoofdletter te laten beginnen**.

Volgende code beschrijft de klasse `Auto` in C#

```
1  class Auto
2  {
3
4  }
```

Binnen het codeblock dat bij deze klasse hoort zullen we verderop dan de werking via properties en methoden beschrijven.

Klassen in Visual Studio toevoegen

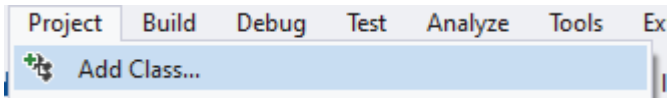
Je kan "eender waar" een klasse aanmaken in een project, maar het is een goede gewoonte om per klasse **een apart bestand** te gebruiken. Dit kan op 2 manieren.

Manier 1:

- In de Solution Explorer, rechterklik op je project.
- Kies "Add".
- Kies "Class..".
- Geef een goede naam voor je klasse.

Manier 2:

- Klik in de menubalk bovenaan op "Project".
- Kies "Add class..." .



Manier 2 is de snelste. Tip: of maak een eigen toetsenbord shortcut, dat is nog sneller natuurlijk.

Objecten aanmaken

Je kan nu objecten aanmaken van de klasse die je hebt gedefinieerd. Dit kan op alle plaatsen in je code waar je in het verleden ook al variabelen kon declareren, bijvoorbeeld in een methode of je `Main`-methode.

Je doet dit door eerst een variabele te definiëren en vervolgens een object te **instantiëren** met behulp van het `new` keyword. De variabele heeft als datatype `Auto`:

```
1 Auto mijnEersteAuto = new Auto();
2 Auto mijnAndereAuto = new Auto();
```

We hebben nu **twee objecten aangemaakt van het type Auto** die we verderop zouden kunnen gebruiken.

Let goed op dat je dus op de juiste plekken dit alles doet:

- Klassen maak je aan als aparte bestanden in je project.
- Objecten creëer je in je code op de plekken waar je deze nodig hebt, bijvoorbeeld in je `Main` methode bij een Console-applicatie.

De `new` operator

In het volgende hoofdstuk gaan we kijken wat er allemaal gebeurt in het geheugen wanneer we een object met `new` aanmaken. Het is echter nu al belangrijk te beseffen dat objecten niet kunnen gemaakt worden zonder `new`. De `new` operator vereist dat je aangeeft van welke klasse je een object wilt aanmaken, gevolgd door ronde haakjes (bijvoorbeeld `new Student()`). We roepen hier een constructor aan (zie verder) die het object in het geheugen zal aanmaken. Vervolgens geeft `new` een adres terug waar het object zich bevindt. Het is dit adres dat we vervolgens kunnen bewaren in een variabele die links van de toekenningsoperator (=) staat.

Test maar eens wat er gebeurt als je volgende code probeert te compileren:

```
1 Auto mijnEersteAuto = new Auto();
2 Auto mijnAndereAuto;
3 Console.WriteLine(mijnEersteAuto);
4 Console.WriteLine(mijnAndereAuto);
```

Je zal een `"Use of unassigned local variable`'mijnAndereAuto'` foutboodschap krijgen. Inderaad, je hebt nog geen object aangemaakt met `new` en `mijnAndereAuto` is dus voorlopig een lege doos (het heeft de waarde `null`).

Dit concept is dus fundamenteel verschillend van de klassieke *valuetypes* die we al kenden (`int`, `double`, enz.). Daar zal volgende code wél werken:

```
1 int balans;
2 Console.WriteLine(balans);
```

Klassen zijn gewoon nieuwe datatypes

In hoofdstuk 2 leerden we dat er allerlei datatypes bestaan. We maakten vervolgens variabelen aan van een bepaald datatype zodat deze variabele als inhoud enkel zaken kon bevatten van dat ene datatype.

Zo leerden we toen volgende datatypes:

- **Valuetypes** zoals `int`, `char` en `bool`.
- Het **`enum`** keyword liet ons toe om een nieuw datatype te maken dat maar een eindig aantal mogelijke waarden (values) kon hebben. Intern bewaarden variabelen van zo'n enum-datatype hun waarde als een `int`.
- **Arrays** waren het laatste soort datatypes. We ontdekten dat we arrays konden maken van eender welk datatype (valuetypes en enums) dat we tot dan kenden.

Wel nu, klassen zijn niet meer dan een nieuw soort datatypes. Kortom: telkens je een klasse aanmaakt, kunnen we in dat project variabelen en arrays aanmaken met dat datatype. We noemen variabelen die een klasse als datatype hebben **objecten**.

Het grote verschil dat deze objecten zullen hebben is dat ze vaak veel complexer zijn dan de eerdere datatypes die we kennen:

- Ze zullen meerdere "waarden" tegelijk kunnen bewaren (een `int` variabele kan maar één waarde tegelijkertijd in zich hebben).
- Ze zullen methoden hebben die we kunnen aanroepen om de variabele "voor ons te laten werken".

Het blijft ingewikkeld hoor. Heel boeiend om de theorie van een speer te leren, maar ik denk dat ik toch beter een paar keer met een speer naar een mammoet werp om echt te voelen wat OOP is.

Ik onthoud nu alvast "**klassen zijn gewoon een nieuwe vorm van complexere datatypes**" dan diegene die ik totnogtoe heb geleerd? Ok?

Correct. Er verandert dus niet veel. Enkel je variabelen worden krachtiger!

De anatomie van een klasse

We zullen nu enkele basisconcepten van klassen en objecten toelichten aan de hand van praktische voorbeelden.

Object methoden

Stel dat we een klasse willen maken die ons toelaat om objecten te maken die verschillende mensen voorstellen. We willen aan iedere mens kunnen zeggen "Praat eens".

We maken een nieuwe klasse `Mens` en plaatsen in de klasse een methode `Praat`:

```
1 class Mens
2 {
3     public void Praat()
4     {
5         Console.WriteLine("Ik ben een mens!");
6     }
7 }
```

We zien twee nieuwe aspecten:

- Het keyword **`static`** mag je **niet** voor een methode signatuur zetten (later ontdekken we wanneer dat soms wel moet) .
- Voor de methode plaatsen we **`public`** : dit is een *access modifier* die aangeeft dat de buitenwereld deze methode op het object kan aanroepen.

Je kan nu elders objecten aanmaken en ieder object z'n methode `Praat` aanroepen:

```
1 Mens joske = new Mens();
2 Mens alfons = new Mens();
3 joske.Praat();
4 alfons.Praat();
```

Er zal twee maal `Ik ben een mens!` op het scherm verschijnen. Waarbij telkens ieder object (`joske` en `alfons`) zelf verantwoordelijk was dat dit gebeurde.

Public en private access modifiers

De **access modifier** geeft aan hoe zichtbaar een bepaald deel van de klasse is. Wanneer je niet wilt dat "van buiten" een bepaalde methode kan aangeroepen worden, dan dien je deze als `private` in te stellen. Wil je dit net wel dat moet je er expliciet `public` voor zetten.

Test in de voorgaande klasse eens wat gebeurt wanneer je `public` vervangt door `private`. Inderdaad, je zal de methode `Praat` niet meer op de objecten kunnen aanroepen.

Wanneer je geen access modifier voor een methode zet in C# dan zal deze als `private` beschouwd worden. Dit geldt voor alle zaken waar je access modifiers voor kan zetten: niets ervoor zetten wil zeggen `private`.

Volgende twee methoden-signaturen zijn dus identiek:

```
private void NiemandMagDitGebruiken()
{
    //...
}

void NiemandMagDitGebruiken()
{
    //...
}
```

Het is een héél slechte gewoonte om géén access modifiers voor iedere methode te zetten. Maak er dus een gewoonte van dit steeds ogenblikkelijk te doen.

Test volgende klasse eens, kan je de methode `VertelGeheim` vanuit de Main op `joske` aanroepen?

```
class Mens
{
    public void Praat()
    {
        Console.WriteLine("Ik ben een mens!");
    }

    private void VertelGeheim()
    {
        Console.WriteLine("Ik ben verliefd op Anneke");
    }
}
```

Naast `private` (het meest beschermd) en `public` (het meest open) zijn er nog een aantal access modifiers die allemaal de toegang tot een klasse meer of minder kunnen inperken. Verderop in het boek zullen we nog `protected` (enkel zichtbaar voor overgeërfde klassen) bekijken. Maar weet dat ook nog `internal` (enkel binnen dezelfde *assembly*) , `protected internal` en `private protected` bestaan.
In dit boek gaan we ons beperken tot `private`, `protected` en `public`. Je zal echter merken dat VS 2022 standaard met `internal` werkt, wat voor ons niet echt uitmaakt.

Reden van private

Waarom zou je bepaalde zaken `private` maken?

De code binnenin een klasse kan overal aan binnen de klasse zelf. Stel dat je dus een erg complexe publieke methode hebt, en je wil deze opsplitsen in meerdere delen, dan ga je die andere delen `private` maken. Dit voorkomt dat programmeurs die je klasse later gebruiken, stukken code aanroepen die helemaal niet bedoeld zijn om rechtstreeks aan te roepen.

Volgende voorbeeld toont hoe je binnenin een klasse andere zaken van de klasse kunt aanroepen: we roepen in de methode `Praat` de methode `VertelGeheim` aan (die `private` is voor de buitenwereld, maar niet voor de code binnen de `Praat`-methode).

```
1  class Mens
2  {
3      public void Praat()
4      {
5          Console.WriteLine("Ik ben een mens!");
6          VertelGeheim();
7      }
8
9      private void VertelGeheim()
10     {
11         Console.WriteLine("Ik ben verliefd op Anneke");
12     }
13 }
```

Als we nu elders een object laten praten als volgt:

```
1  Mens rachid = new Mens();
2  rachid.Praat();
```

Dan zal de uitvoer worden:

```
Ik ben een mens!
Ik ben verliefd op Anneke
```

Met behulp van de **dot-operator** (`.`) kunnen we aan alle informatie die ons object aanbiedt aan de buitenwereld. Ook dit zagen we reeds toen we een `Random`-object hadden: we konden maar een handvol zaken aanroepen op zo'n object, waaronder de `Next` methode.

Het is natuurlijk een beetje vreemd dat nu al onze objecten zeggen dat ze verliefd zijn op Anneke. Dit is niet het smurfendorp met maar 1 meisje! Dit gaan we verderop oplossen. *Stay tuned!*

Instantievariabelen

Voorlopig doen alle objecten van het type `Mens` hetzelfde. Ze kunnen praten en zeggen hetzelfde. We weten echter dat objecten ook een interne staat hebben die per object individueel is (we zagen dit reeds toen we balletjes over het scherm lieten botsen: ieder balletje onthield z'n eigen richtingsvector en positie). Dit kunnen we dankzij **instantievariabelen** (ook wel **datavelden** of **datafields** genoemd) oplossen. Dit zullen variabelen zijn waarin zaken kunnen bewaard worden die verschillen per object.

Stel je voor dat we onze mensen een geboortejaar willen geven. Ieder object zal zelf in een instantievariabele bijhouden wanneer ze geboren zijn (het vertellen van geheimen zullen we verderop behandelen):

```
1  class Mens
2  {
3      private int geboorteJaar = 1970; //instantievariabele
4
5      public void Praat()
6      {
7          Console.WriteLine("Ik ben een mens! ");
8          Console.WriteLine($"Ik ben geboren in {geboorteJaar}.");
9      }
10 }
```

Enkele belangrijke concepten:

- De instantievariabele `geboorteJaar` zetten we private: we willen niet dat de buitenwereld het geboortejaar van een object kan aanpassen. Beeld je in dat dat in de echte wereld ook kon. Dan zou je naar je kameraad kunnen roepen "Hey Adil, jouw geboortejaar is nu 1899! Ha!" Waarop Adil vloekend verandert in een steenoud mannetje.
- We geven de variabele een beginwaarde `1970`. Alle objecten zullen dus standaard in het jaar 1970 geboren zijn wanneer we deze met `new` aanmaken.
- We kunnen de inhoud van de instantievariabelen lezen (en veranderen) vanuit andere delen in de code. Zo gebruiken we `geboorteJaar` in de tweede lijn van de `Praat` methode. Als je die methode nu zou aanroepen dan zou het geboortejaar van het object dat je aanroept mee op het scherm verschijnen.

We moeten ook dringend enkele extra niet-officiële identifier regels in het leven roepen:

- Klassenamen en methoden in klassen beginnen altijd met een hoofdletter.
- Alles dat `public` is in een klasse begint ook met een hoofdletter.
- Alles dat `private` is begint met een kleine letter (of liggend streepje), tenzij het om een methode gaat, die begint altijd met een hoofdletter.

Dit zijn geen officiële regels, maar afspraken die veel programmeurs onderling hebben gemaakt. Het maakt de code leesbaarder.

Wat?! Ik ben hier niet voor jou? Omdat je geen `goto` hebt gebruikt?! Flink hoor. Maar daarvoor ben ik hier niet. Ik zag je wel denken: "Als ik nu die instantievariabele ook eens `public` maak."

Niet doen. Simpel! **Instantievariabele mogen NOOIT `public` gezet worden**. De C# standaard laat dit weliswaar toe, maar dit is één van de slechtste programmeerdingen die je kan doen. Wil je toch de interne staat van een object kunnen aanpassen dan gaan we dat via **properties** en **methoden** kunnen doen, wat we zo meteen gaan uitleggen. Zie dat ik hier niet te vaak tussenbeide moet komen. Dank!

Ok, we zullen maar luisteren naar meneer de agent. Stel nu dat we een verjongingsstraal hebben waarmee we het geboortejaar van de mensen steeds met 1 jaar kunnen verhogen (en ze dus een jaar jonger maken).

```
1  class Mens
2  {
3      private int geboorteJaar = 1970;
4      public void Praat()
5      {
6          Console.WriteLine("Ik ben een mens! ");
7          Console.WriteLine($"Ik ben geboren in {geboorteJaar}.");
8      }
9      public void StartVerjongingskuur()
10     {
11         Console.WriteLine("Jeuj. Ik word jonger!");
12         geboorteJaar++;
13     }
14 }
```

Zoals al gezegd: **Ieder object zal z'n eigen geboortejaar hebben**.

Die laatste opmerking is een kernconcept van OOP: ieder object heeft z'n eigen interne staat die kan aangepast worden individueel van de andere objecten van hetzelfde type. We zullen dit testen in volgende voorbeeld waarin we 2 objecten maken en enkel 1 ervan verjongen. Kijk wat er gebeurt:

```
1  Mens elvis = new Mens();
2  Mens bono = new Mens();
3  elvis.StartVerjongingskuur();
4  elvis.Praat();
5  bono.Praat();
```

Als je voorgaande code zou uitvoeren zal je zien dat het geboortejaar van Elvis verhoogd en niet die van Bono wanneer we `StartVerjongingskuur` aanroepen. Zoals het hoort!

De uitvoer zal zijn:

```
Jeuj. Ik word jonger!
Ik ben een mens!
Ik ben geboren in 1971.
Ik ben een mens!
Ik ben geboren in 1970.
```

"Ja maar, nu pas je toch het geboortejaar van buiten aan via een methode, ook al gaf je aan dat dit niet de bedoeling was want dan zou je Adil ogenblikkelijk erg jong kunnen maken."

Correct. Maar dat was dus maar een voorbeeld. De hoofdreden dat we instantievariabelen niet zomaar `public` mogen maken is om te voorkomen dat de buitenwereld instantievariabelen waarden geeft die de werking van de klasse zouden stuk maken. Stel je voor dat je dit kon doen: `adil.geboortejaar = -12000;`

Dit kan nefaste gevolgen hebben voor de klasse.

Daarom gaan we de toegang tot instantievariabelen als het ware controleren door deze enkel via properties en methoden toe te laten. We zouden dan bijvoorbeeld het volgende kunnen doen:

```
1  class Mens
2  {
3      private int geboorteJaar = 1970;
4
5      public void VeranderGeboortejaar(int geboorteJaarIn)
6      {
7          if(geboorteJaarIn >= 1900)
8              geboorteJaar = geboorteJaarIn;
9      }
```

Mooi he. Zo voorkomen we dus dat de buitenwereld illegale waarden aan een variabele kan geven (in dit geval kan dus niet voor 1900 geboren zijn). **Objecten zijn verantwoordelijk voor zichzelf** en moeten zichzelf dus ook beschermen zodat de buitenwereld niets met hen doet dat hun eigen werking om zeep helpt.

Andere lieven

We kunnen nu het probleem oplossen dat al onze mensen verliefd zijn op Anneke. Volgende code toont dit:

```
class Mens
{
    private string lief = "niemand";

    public void VeranderLief(string nieuwLief)
    {
        lief = nieuwLief;
    }
    public void Praat()
    {
        Console.WriteLine("Ik ben een mens!");
        VertelGeheim();
    }

    private void VertelGeheim()
    {
        if( lief != "niemand")
            Console.WriteLine($"Ik ben verliefd op {lief}");
        else
            Console.WriteLine("Ik ben op niemand verliefd.");
    }
}
```

Nu kunnen we dus “Temptation Island - de OOP editie” beginnen:

```
Mens deelnemer1 = new Mens();
Mens deelnemer2 = new Mens();
deelnemer1.Praat();
deelnemer2.Praat();

deelnemer2.VeranderLief("phoebe");
deelnemer1.Praat();
deelnemer2.Praat();

deelnemer1.VeranderLief("camilla");
deelnemer1.Praat();
deelnemer2.Praat();
```

De uitvoer van voorgaande code zal zijn:

```
Ik ben een mens!
Ik ben op niemand verliefd.
Ik ben een mens!
Ik ben op niemand verliefd.
Ik ben een mens!
Ik ben op niemand verliefd.
Ik ben een mens!
Ik ben verliefd op phoebe
Ik ben een mens!
Ik ben verliefd op camilla
Ik ben een mens!
Ik ben verliefd op phoebe
```

Klasse "`Studenten`" of "`Student`"?

Veel beginnende programmeurs maken fouten op het correct kunnen onderscheiden wat de klassen en wat de objecten in hun opgave juist zijn. Het is altijd belangrijk te begrijpen dat een klasse weliswaar beschrijft hoe alle objecten van dat type werken, maar op zich gaat die beschrijving steeds over 1 object uit de verzameling. *Say what now?!*

Als je een klasse `Student` hebt, dan zal deze eigenschappen hebben zoals `Punten`, `Naam` en `Geboortejaar`. Als je een klasse `Studenten` daarentegen hebt, dan is dit vermoedelijk een klasse die beschrijft hoe een groep studenten moet werken in je applicatie. Mogelijk zal je dan properties hebben zoals `KlasNaam`, `AantalAfwezigen`, enz. Kortom, eigenschappen over de groep, niet over 1 student.

"`Level`" of "`Level1`"?

Een andere veelgemaakte fout is klassen te schrijven, die maar exact één object kan en moet creëren (dit heet een *singleton*). Stel je voor dat je een spel maakt waarin verschillende levels zijn. Een logische keuze zou dan zijn om een klasse `Level` te maken (niét `Levels`) die properties heeft zoals `MoeilijkheidsGraad`, `HeeftGeheimeGrotten`, `AantalVijanden`, enz.

Vervolgens kunnen we dan instanties maken: *1 object stelt 1 level in het spel voor.* De speler kan dan van level naar level gaan en de code start dan bijvoorbeeld telkens de `BeginLevel` methode:

```
1  Level level1 = new Level();
2  level1.BeginLevel();
```

Wat dus niet mag zijn **klassen** met namen zoals `level1`, `level2`, enz. Vermoedelijk hebben deze klasse 90% gelijkaardige code en is er dus een probleem met wat we de *architectuur* van je code zouden kunnen noemen. Of duidelijker: je snapt niet wat het verschil is tussen klassen en objecten!
Objecten met namen zoals `level1` en `level2` zijn wél dus toegestaan, daar ze dan vermoedelijk allemaal van het type `Level` zijn. **Maar opgelet: als je variabelen hebt die een genummerd zijn (bv. `bal1`, `bal2`, enz.) dan is de kans groot dat je vervolgens een array van objecten nodig hebt** (wat we in hoofdstuk 12 uit de doeken zullen doen).

9.3 Properties

We zagen zonet dat instantievariabelen nooit `public` mogen zijn om te voorkomen dat de buitenwereld onze objecten 'vult' met slechte zaken. Het voorbeeld waarbij we vervolgens een methode `StartVerjongingskuur` gebruikten om op gecontroleerde manier toch aan de interne staat van objecten te komen is één oplossing, maar een nogal *oldschool* oplossing.

Deze manier van werken - methoden gebruiken om instantievariabelen aan te passen of uit te lezen- is wat voorbij gestreefd binnen C#. Onze programmeertaal heeft namelijk het concept **properties** (*eigenschappen*) in het leven geroepen die toelaten op een veel eenvoudigere manier aan de interne staat van objecten te komen.

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.

Star Wars en de nood aan properties

In het Star Wars universum heb je goede oude "Darth Vader". Hij behoort tot de mysterieuze klasse van de Sith Lords. Deze lords lopen met een geheim rond: ze hebben een zogenaamde Sithnaam, een naam die ze enkel mogen bekend maken aan andere Sith Lords, maar aan niemand anders. Voorts heeft een Sith Lord ook een hoeveelheid energie (*The Force*) waarmee hij kattekwaad kan uithalen. Deze energie mag natuurlijk nooit onder nul gezet worden.

We kunnen voorgaande als volgt schrijven:

```
1  class SithLord
2  {
3      private int energie;
4      private string sithName;
5  }
```

Het is uit den boze dat we eenvoudige instantievariabelen (`energie` en `name`) `public` maken. Zouden we dat wel doen dan kunnen externe objecten deze geheime informatie uitlezen!

```
1  SithLord palpatine = new SithLord();
2  Console.WriteLine(palpatine.sithName); //dit zal niet werken dankzij private
```

We willen echter wel van buiten uit het energie-level van een sithLord kunnen instellen. Maar ook hier hetzelfde probleem: wat als we de energie-level op -1000 instellen? Terwijl energie nooit onder 0 mag gaan.

Properties lossen dit probleem op.

2 soorten properties

Er zijn 2 soorten properties in C#:

- **Full properties**: deze stijl van properties verplicht ons véél code te schrijven, maar we hebben ook volledige controle over wat er gebeurt.
- **Autoproperties** zijn exact het omgekeerde van full properties: weinig code, maar ook weinig controle.

We behandelen eerst full properties, daar autoproperties een soort afgeleide van full properties zijn (bepaalde aspecten van full properties worden bij autoproperties achter de scherm verstopt zodat jij als programmeur er geen last van hebt).

In één van de volgende versies van C# (normaal versie 11) zal er nog een derde type verschijnen: de zogenaamde semi-auto properties. Een - je raadt het nooit- propertytype dat zich tussen beide bestaande types zal bevinden. De details en exacte gebruik ervan worden nog besproken op github (**github.com/dotnet/csharplang/issues/140**) door de ontwikkelaars, dus het is nog te vroeg om deze al op te nemen in dit boek.

Full properties

Properties herken je aan de `get` en `set` keywords in een klasse. Een property is een beschrijving van wat er moet gebeuren indien je informatie uit (**`get`**) een object wilt halen of informatie net in (**`set`**) een object wilt plaatsen.

In volgende voorbeeld maken we een property, genaamd `Energie` aan. Deze doet niets anders dan rechtstreeks toegang tot de instantievariabele `energie` te geven:

```
1  class SithLord
2  {
3      private int energie;
4
5      public int Energie
6      {
7          get
8          {
9              return energie;
10         }
11         set
12         {
13             energie = value;
14         }
15     }
16 }
```

Dankzij voorgaande code kunnen we nu buiten het object de property `Energie` gebruiken als volgt:

```
SithLord Vader = new SithLord();
Vader.Energie = 20; //set
Console.WriteLine($"Vaders energie is {Vader.Energie}"); //get
```

Laten we eens inzoomen op de full property code.

Full property: identifier en datatype

De eerste lijn van een full property beschrijft de naam (identifier) en datatype van de property: `public int Energie`

Een property is altijd `public` daar dit de essentie van een property net is "de buitenwereld gecontroleerde toegang tot de interne staat van een object geven".

Vervolgens zeggen we wat voor **datatype** de property moet zijn en geven we het een naam die moet voldoen aan de identifier regels van weleer. Voor de buitenwereld zal een property zich gedragen als een gewone variabele, met de naam `Energie` van het type `int`.

Indien je de property gaat gebruiken om een instantievariabele 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 `Energie` i.p.v. `energie`).

Full property: get gedeelte

Indien je wenst dat de property data **naar buiten** kan sturen, dan schrijven we de get-code. Binnen de accolades van de `get` schrijven we wat er naar buiten moet gestuurd worden.

```
get
{
    return energie;
}
```

Dit werkt dus identiek aan een methode met een returntype. **Het element dat je met `return` teruggeeft 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 `energie` uitlezen via de property en het get-gedeelte, bijvoorbeeld `int uitgelezen = palpatine.Energie;`.

We mogen eender wat doen in het get-gedeelte (net zoals bij methoden) zolang er finaal maar iets uitgestuurd wordt m.b.v. `return`. We gaan hier verderop meer over vertellen, want soms is het handig om *getters* te schrijven die de data transformeren voor ze uitgestuurd wordt.

Full property: set gedeelte

In het set-gedeelte schrijven we de code die we moeten hanteren indien men van buiten een waarde aan de property wenst te geven om zo een instantievariabele aan te passen.

```
set
{
    energie = value;
}
```

De waarde die we van buiten krijgen (als een parameter zeg maar) zal altijd in een lokale variabele **`value`** worden bewaard binnenin de set-code. Deze zal van het type van de property zijn.

Deze `value` parameter is een essentiëel onderdeel van de `set` syntax en kan je niet hernoemen.

Vervolgens kunnen we `value` toewijzen aan de interne variabele indien gewenst: `energie = value;`. Uiteraard kunnen we die toewijzing dus ook gecontroleerd laten gebeuren, wat we zo meteen zullen uitleggen.

We kunnen vanaf nu van buitenaf waarden toewijzen aan de property en zo `energie` toch bereiken: `palpatine.Energie = 50;`.

Je bent niet verplicht om een property te maken wiens naam overeen komt met een bestaande instantievariabele (**maar dit wordt ten stelligste afgeraden**). Dit mag dus ook:

```
class Auto
{
    private int benzinePeil;

    public int FuelLevel
    {
        get { return benzinePeil; }
        set { benzinePeil = value; }
    }
}
```

Visual Studio heeft een ingebouwde snippet om snel een full property, inclusief een bijhorende private instantievariabele, te schrijven. **Typ `propfull` gevolgd door twee maal op de tab-toets te duwen**.

Full property met toegangscontrole

De full property `Energie` heeft nog steeds het probleem dat we negatieve waarden kunnen toewijzen (via de `set`) die dan vervolgens zal toegewezen worden aan `energie`.

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. Daar de `value` variabele de waarde krijgt die we aan de property van buiten af geven, kunnen we deze dus controleren en , indien nodig, bijvoorbeeld niet toewijzen. Volgende voorbeeld zal enkel de waarde toewijzen indien deze groter of gelijk aan `0` is:

```
public int Energie
{
    get
    {
        return energie;
    }
    set
    {
        if(value >= 0)
            energie = value;
    }
}
```

Volgende lijn zal dus geen effect hebben:
`palpatine.Energie = -1;`

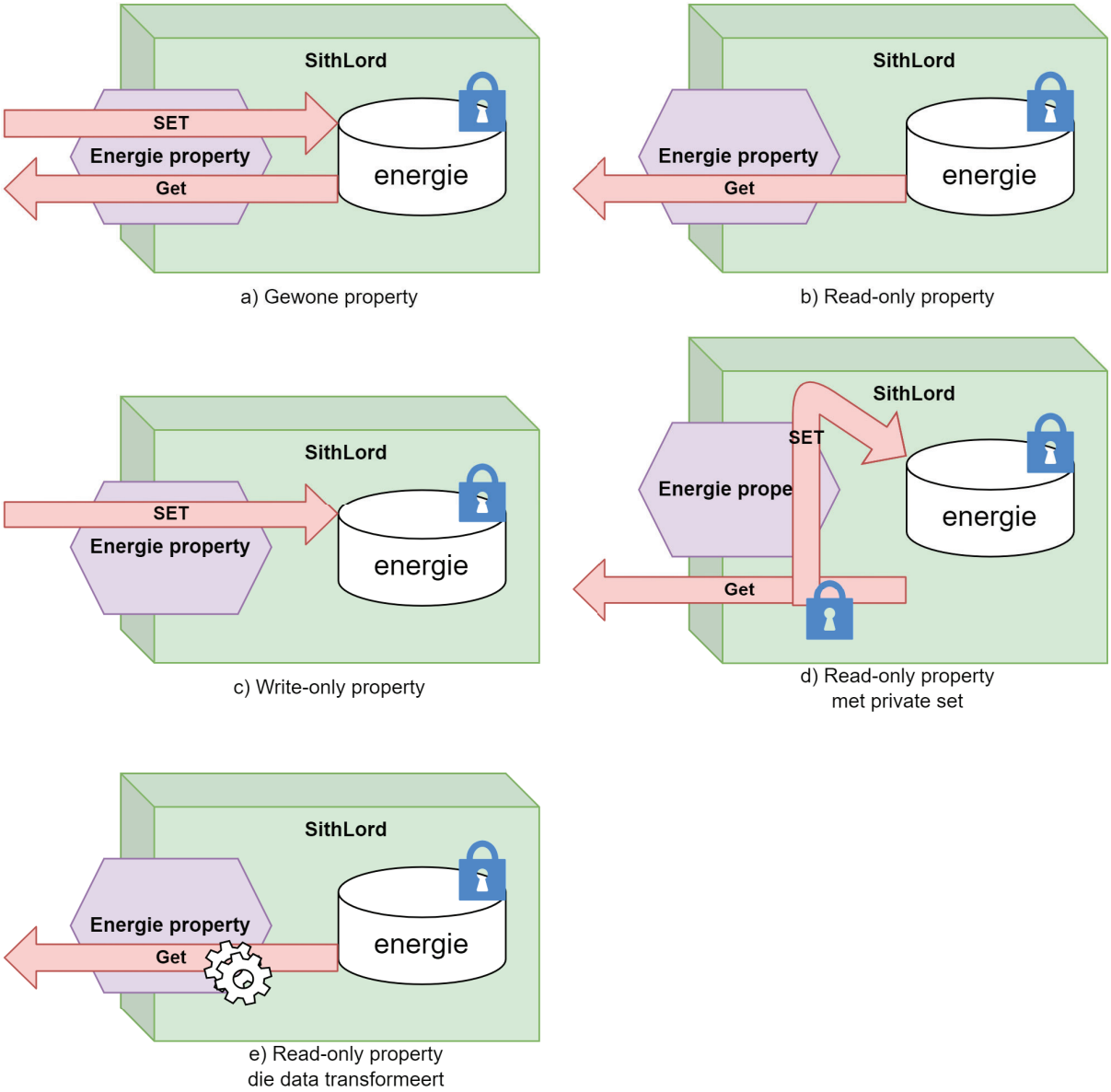
We kunnen de code binnen `set` (en `get`) zo complex maken als we willen.

Probeer wel steeds de OOP-principes te hanteren wanneer je met properties werkt: in de `get` en `set` van een property mogen enkel die dingen gebeuren die de verantwoordelijkheid van de property zelf zijn. Je gaat dus bijvoorbeeld niet controleren of een andere property geen illegale waarden krijgt, daar is die andere property voor verantwoordelijk.

Property variaties

We zijn niet verplicht om zowel de `get` en de `set` code van een property te schrijven. Dit laat ons toe om een aantal variaties te schrijven:

- **Write-only property**: heeft geen `get`.
- **Read-only property**: heeft geen `set`.
- **Read-only property met private `set`** (het omgekeerde , een private `get`, zal je zelden tegenkomen).
- **Read-only property die data transformeert**: om interne data in een andere vorm uit je object te krijgen.



De verschillende full properties mooi opgelijst.

Write-only property

Dit soort properties zijn handig indien je informatie naar een object wenst te sturen dat niet mag of moet uitgelezen kunnen worden. Het meest typische voorbeeld is een property `Pincode` van een klasse `BankRekening`.

```
public int Energie
 {
     set
     {
         if(value >= 0)
             energie = value;
     }
 }
```

We kunnen dus enkel `energie` een waarde geven, maar niet van buiten uitlezen.

Read-only property

Letterlijk het omgekeerde van een write-only property. Deze gebruik je vaak wanneer je informatie uit een object wil kunnen uitlezen uit een instantievariabele dat NIET door de buitenwereld mag aangepast worden.

```
public int Energie
 {
     get
     {
         return energie;
     }
 }
```

We kunnen enkel `energie` van buiten uitlezen, maar niet aanpassen.

Het `readonly` keyword heeft andere doelen en wordt NIET gebruikt in C# om een readonly property te maken.

Read-only property met private set

Soms gebeurt het dat we van enkel voor de buitenwereld de property read-only willen maken. We willen echter intern (in de klasse zelf) nog steeds controleren dat er geen illegale waarden aan private instantievariabelen worden gegeven. Op dat moment definiëren we een read-only property met een private setter:

```
1    public int Energie
2     {
3         get
4         {
5             return energie;
6         }
7         private set
8         {
9             if(value >= 0)
10                energie = value;
11        }
12    }
```

Van buiten zal enkel code werken die de `get` van deze property aanroept, bijvoorbeeld:

```
Console.WriteLine(palpatine.Energie);
```

Code die de `set` van buiten nodig heeft (bv. `palpatine.Energie = 65;`) zal een fout geven ongeacht of deze geldig is of niet.

Het is een goede gewoonte om **altijd** via de properties je interne variabele aan te passen en niet rechtstreeks via de instantievariabele zelf. Dit is zo'n nuttige tip dat we op de volgende pagina de voorman hier ook nog even over aan het woord gaan laten.

Lukt het een beetje? Properties zijn in het begin wat overweldigend, maar geloof me: ze zijn zowat dé belangrijkste bewoners in de .NET/C# wereld.

Nu even goed opletten: indien we **IN** het object de instantievariabelen willen aanpassen dan is het een goede gewoonte om dat **via de property** te doen (ook al zit je in het object zelf en heb dus eigenlijk de property niet nodig). Zo zorgen we ervoor dat de bestaande controle in de property niet wordt omzeilt. Kijk zelf naar volgende **slechte** codevoorbeeld:

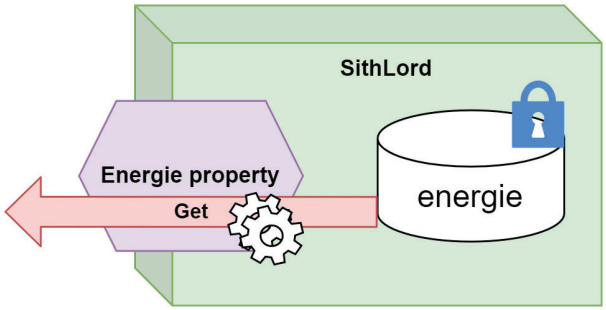
```
1  class SithLord
2  {
3      private int energie;
4      private string sithName;
5      public void ResetLord(int resetWaarde)
6      {
7          energie = resetWaarde;
8      }
9      public int Energie
10     {
11         get
12         {
13             return energie;
14         }
15         private set
16         {
17             if(value >= 0)
18                 energie = value;
19         }
20     }
21 }
```

De nieuw toegevoegde methode `ResetLord` willen we gebruiken om de lord z'n energie terug te verlagen. Als we deze methode met een negatieve waarden aanroepen zullen we alnsog `energie` op een verkeerde waarde instellen. Nochtans is dit een illegale waarde volgens de set-code van de property.

We moeten dus in de methode ook expliciet via de property gaan om bugs te voorkomen en dus gaan we in `ResetLord`schrijven naar de property `Energie` én niet rechtstreeks naar de instantievariabele `energie`:

```
1  public void ResetLord(int resetWaarde)
2  {
3      Energie = resetWaarde; // Energie i.p.v. energie
4  }
```

Read-only properties die transformeren



Transformerende properties: Erg nuttig, maar vaak wat stiefmoederlijk behandeld.

Je bent uiteraard niet verplicht om voor iedere instantievariabele een bijhorende property te schrijven. Omgekeerd ook: mogelijk wil je extra properties hebben voor data die je 'on-the-fly' kan genereren dat niet noodzakelijk uit een instantievariabele komt. Stel dat we volgende klasse hebben:

```
class Persoon
{
    private string voornaam;
    private string achternaam;
}
```

We willen echter ook soms de volledige naam of emailadres krijgen, beide gebaseerd op de inhoud van de instantievariabelen `voornaam` en `achternaam`. Via een read-only property die transformeert kan dit:

```
class Persoon
{
    private string voornaam;
    private string achternaam;
    public string VolledigeNaam
    {
        get
        {
            return $"{voornaam} {achternaam}";
        }
    }
    public string Email
    {
        get
        {
            return $"{voornaam}@ziescherp.be";
        }
    }
}
```

Autoproperties

Automatische eigenschappen (**autoproperties** , soms ook *autoprops* genoemd) laten toe om snel properties te schrijven zonder dat we de achterliggende instantievariabele moeten beschrijven.

Heel vaak wil je heel eenvoudige variabelen aan de buitenwereld van je klasse beschikbaar stellen. Omdat je instantievariabelen echter niet `public` mag maken, moeten we dus properties gebruiken die niets anders doen dan als doorgeefluik fungeren. Autoproperties doen dit voor ons: het zijn vereenvoudige full properties waarbij de achterliggende instantievariabele onzichtbaar voor ons is.

Zo kan je eenvoudig de volgende klasse `Persoon` herschrijven met behulp van autoproperties. De originele klasse mét full properties (we hebben de layout wat compacter gemaakt om een extra blad te besparen in dit boek):

```
1  public class Person
2  {
3      private string voornaam;
4      private int geboorteJaar;
5
6      public string Voornaam
7      {
8          get { return voornaam; }
9          set { voornaam = value; }
10     }
11
12     public int Geboortejaar
13     {
14         get { return geboorteJaar; }
15         set { geboorteJaar = value; }
16     }
17 }
```

De herschreven klasse met autoproperties wordt:

```
1  public class Person
2  {
3      public string Voornaam { get; set; }
4      public int Geboortejaar { get; set; }
5  }
```

Beide klassen hebben exact dezelfde functionaliteit, echter is de laatste klasse aanzienlijk korter en dus eenvoudiger om te lezen. **De private instantievariabelen zijn niét meer aanwezig.** C# gaat die voor z'n rekening nemen. Alle code zal dus via de properties moeten gaan.

Het is belangrijk te benadrukken dat de achterliggende instantievariabele onzichtbaar is in autoproperties en **onmogelijk** kan gebruikt worden. Alles gebeurt via de autoproperty, altijd.

Vaak zal je nieuwe klassen eerst met autoproperties beschrijven. Naarmate de specificaties dan vereisen dat er bepaalde controles of transformaties moeten gebeuren, zal je stelselmatig autoproperties vervangen door full properties.

Dit kan trouwens automatisch in VS: selecteer de autoprop in kwestie en klik dan vooraan op de schroevendraaier en kies “Convert to full property”.

Opgelet: Merk op dat de syntax die VS gebruikt om een full property te schrijven anders is dan wat we hier uitleggen. Wanneer je VS laat doen krijg je een oplossing met allerlei => tekens. Dit is zogenaamde **Expression Bodied Member syntax (EBM)**. We behandelen deze (nieuwere) C# syntax in de appendix.

Beginwaarden van autoproperties

Je mag autoproperties beginwaarden geven door de waarde achter de property te schrijven, als volgt:

```
public int Geboortejaar {get;set;} = 2002;
```

Al je objecten zullen nu als geboortejaar 2002 hebben wanneer ze geïnstantieerd worden.

Altijd auto-properties?

Merk op dat je autoproperties dus enkel kan gebruiken indien er geen extra logica in de property (bij de set of get) aanwezig moet zijn.

Stel dat je bij de setter van geboorteJaar wil controleren op een negatieve waarde, dan zal je dit zoals voorheen moeten schrijven en kan dit niet met een automatic property:

```
1 set
2 {
3     if( value > 0)
4         geboorteJaar = value;
5 }
```

Voorgaande property kan dus *NIET* herschreven worden met een automatic property.

Alleen-lezen autoproperties

Je kan autoproperties ook gebruiken om bijvoorbeeld een read-only property met private setter te definiëren. Als volgt:

```
public string Voornaam { get; private set; }
```

Een andere manier die ook kan wanneer we enkel een read-only property nodig hebben, is als volgt:

```
public string Voornaam { get; } = "Tim";
```

Hierbij zijn we dan wel verplicht om ogenblikkelijk deze property een beginwaarde te geven, daar we deze op geen enkele andere manier nog kunnen aanpassen.

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.
Via `propg` gevolgd door twee maal de tabtoets krijg je een autoproperty met private setter.

Methode of property?

Een veel gestelde vraag bij beginnende OOP-ontwikkelaars is: “Moet dit in een property of in een methode geplaatst worden?”

De regels zijn niet in steen gebeiteld, maar ruwweg kan je stellen dat:

- Betreft het een actie of gedrag: iets dat het object moet doen (tekst tonen, iets berekenen of aanpassen, enz.) dan plaats je het in een **methode**.
- Betreft het een eigenschap van het object, dan gebruik je een **property**.

9.4 OOP in de praktijk : DateTime

Doe die zwembroek maar weer aan! We gaan nog eens zwemmen.

Zoals je vermoedelijk al doorhebt hebben we met properties en methoden nog maar een tipje van de klasse-ijsberg besproken. Vreemde dingen zoals *constructors*, *static methoden*, *overerving* en arrays van objecten staan ons nog allemaal te wachten.

Om je toch al een voorsmaakje van de kracht van klassen en objecten te geven, gaan we eens kijken naar één van de vele klassen die je tot je beschikking hebt in C#. Je hebt al leren werken met bijvoorbeeld de `Random` klasse, maar ook al met wat speciale *static klassen* zoals de `Math`- en `Console`-bibliotheek die je kan gebruiken ZONDER dat je er objecten van moet aanmaken (het keyword `static` is daar de oorzaak van).

Nog zo'n handige ingebouwde klasse is de `DateTime` klasse, die, je raadt het nooit, toelaat om de tijd en/of datum in een object voor te stellen.

De .NET klasse `DateTime` is de ideale manier om te leren werken met objecten. Het is een nuttige en toegankelijk klasse.

DateTime objecten aanmaken

Er zijn 2 manieren om `DateTime` objecten aan te maken:

1. Door aan de klasse de huidige datum en tijd te vragen via `DateTime.Now`.
2. Door manueel de datum en tijd in te stellen met het `new` keyword en de **klasseconstructor** (een concept dat we binnen 2 hoofdstukken uit de doeken gaan doen)

DateTime.Now

Volgend voorbeeld toont hoe we een object kunnen maken dat de huidige datum tijd van het systeem bevat. Vervolgens printen we dit op het scherm:

```
1 DateTime currentTime = DateTime.Now;
2 Console.WriteLine(currentTime);
```

`DateTime.Now` is een zogenaamde **static property** wat verderop in het boek zal uitgelegd worden.

Met constructor en `new`

De constructor van een klasse laat toe om bij het maken van een nieuw object, beginwaarden voor bepaalde instantievariabelen of properties mee te geven. De `DateTime` klasse heeft meerdere constructors gedefiniëerd zodat je bijvoorbeeld een object kan aanmaken dat bij de start reeds de geboortedatum van de auteur bevat:

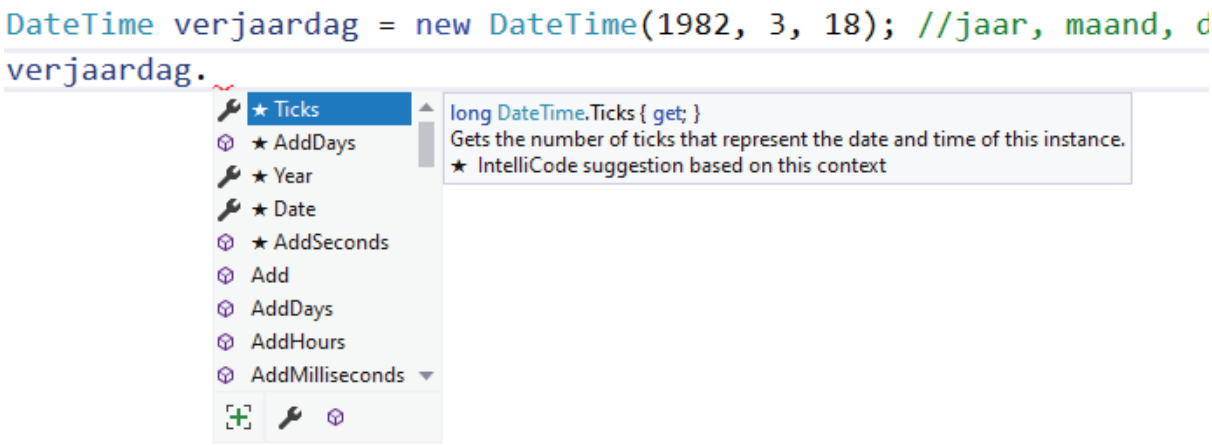
```
DateTime verjaardag = new DateTime(1981, 3, 18); //jaar, maand, dag
```

Ook is er een constructor om startdatum én -tijd mee te geven bij de objectcreatie:

```
1 //Volgorde: jaar, maand, dag, uur, minuten, seconden
2 DateTime trouwMoment = new DateTime(2017, 4, 21, 10, 00,34 );
```

DateTime methoden

Van zodra je een `DateTime` object hebt gemaakt zijn er tal van nuttige methoden die je er op kan aanroepen. Visual Studio is zo vriendelijk om dit te visualiseren wanneer we de dot-operator typen achter een object:



Iedere kubus stelt een methode voor. Iedere Engelse sleutel een property.

Add-methoden

De ingebouwde methoden beginnen allemaal met `Add`, gevolgd door wat er moet bijgevoegd worden: `AddDays`, `AddHours`, `AddMilliseconds`, `AddMinutes`, `AddMonths`, `AddSeconds`, `AddTicks`, `AddYears`.

Een *tick* is 100 nanoseconden, oftewel 1 tien miljoenste van een seconden. Dat lijkt een erg klein getal (wat het voor ons ook is) maar voor computers is dit het soort tijdsintervals waar ze mee werken.

Deze methoden kan je gebruiken om een bepaalde aantal dagen, uren, minuten op te tellen bij de huidige tijd en datum van een object.

Het object zal voor ons de "berekening" hiervan doen en vervolgens een **nieuw DateTime object** teruggeven dat je moet bewaren wil je er iets mee doen.

In volgende voorbeeld wil ik ontdekken op welke datum de wittebroodsweken van m'n huwelijk eindigen (pakweg 5 weken na de trouwdag).

```
1 DateTime eindeWitteBroodsweken = trouwMoment.AddDays(35);
2 Console.WriteLine(eindeWitteBroodsweken);
```

DateTime properties

Dit hoofdstuk heeft al aardig wat woorden verspild aan properties, en uiteraard heeft ook de `DateTime` klasse een hele hoop interessante properties die toelaten om de interne staat van een `DateTime` object te bewerken of uit te lezen.

Enkele nuttige properties van `DateTime` zijn: `Date`, `Day`, `DayOfWeek`, `DayOfYear`, `Hour`, `Millisecond`, `Minute`, `Month`, `Second`, `Ticks`, `TimeOfDay`, `Today`, `UtcNow`, `Year`.

Alle properties van DateTime zijn read-only en hebben dus een private setter die we niet kunnen gebruiken.

Een voorbeeld:

```
1 Console.WriteLine($"Einde in maand nr: {eindeWitteBroodsweken.Month}.");
2 Console.WriteLine($"Dat is een {eindeWitteBroodsweken.DayOfWeek}.");
```

Dit geeft op het scherm:

```
Je wittebroodsweken eindigen in maand nummer: 5.
Dat is een Friday.
```

Static methoden

Sommige methoden zijn `static` dat wil zeggen dat je ze enkel rechtstreeks op de klasse kunt aanroepen. Vaak zijn deze methoden hulpmethoden waar de individuele objecten niets aan hebben. We hebben dit reeds gebruikt, zonder het te weten, bij de `Math` en `Console`-klassen.

We behandelen `static` uitgebreid verderop in het boek.

De tijd uit een string inlezen

Parsen laat toe dat je strings omzet naar een `DateTime` object. Dit is handig als je bijvoorbeeld de gebruiker via `Console.ReadLine()` tijd en datum wilt laten invoeren:

```
1 string datumInvoer = Console.ReadLine();
2 DateTime datumVerwerkt = DateTime.Parse(datumInvoer);
3 Console.WriteLine(datumVerwerkt);
```

Indien je nu dit programma'tje zou uitvoeren en als gebruiker "8/11/2016" zou intypen, dan zal deze datum geparsed worden en in het object `datumVerwerkt` komen.

Zoals je ziet roepen we `Parse` aan op `DateTime` en dus niet op een specifiek object. Dat was ook zo reeds bijvoorbeeld bij `int.Parse` wat dus doet vermoeden dat zelfs het `int` datatype eigenlijk een klasse is!

IsLeapYear

Deze nuttige methode geeft een `bool` terug om aan te geven of de actuele parameter (type `int`) een schrikkeljaar voorstelt of niet:

```
1 DateTime vandaag = DateTime.Now;
2 if(DateTime.IsLeapYear(vandaag.Year))
3     Console.WriteLine("Dit jaar is een schrikkeljaar.");
```

TimeSpan

Je kan DateTime objecten ook bij elkaar optellen en aftrekken. Het resultaat van deze bewerking geeft echter niet een DateTime object terug, **maar een `TimeSpan` object**. Dit is nieuwe object van het type `TimeSpan` (wat dus een andere klasse is) dat aangeeft hoe groot het verschil is tussen de 2 DateTime objecten kunnen we als volgt gebruiken:

```
1 DateTime vandaag = DateTime.Today;
2 DateTime geboorteDochter = new DateTime(2009,6,17);
3 TimeSpan verschil = vandaag - geboorteDochter;
4 Console.WriteLine($"{verschil.TotalDays} dagen sinds geboorte dochter.");
```

Je zal de `DateTime` klasse in véél van je projecten kunnen gebruiken waar je iets met tijd, tijdsverschillen of datums wilt doen. We hebben de klasse in deze sectie echter geen eer aangedaan. De klasse is veel krachtiger dan we hier hebben doen uitschijnen. Het is een goede gewoonte als beginnende programmeur om steeds de documentatie van nieuwe klassen er op na te slaan. Wanneer je in je browser zoekt op "C#" gevolgd door de naam van de klasse dan zal je zo goed als zeker als eerste *hit* de officiële .NET documentatie krijgen op **docs.microsoft.com**.

Gaat het nog?! Dit was een stevig hoofdstuk he. We hebben zo maar eventjes 4 heel grote fasen doorlopen:

1. Eerst keken we hoe OOP ons kan helpen in een real-life voorbeeld, Pong. We schreven code die hier en daar herkenbaar was, maar op andere plaatsen totaal nieuw was.
2. Vervolgens namen we de mammoet bij de horens en bekeken we de theorie van OO, die ons vooral verwarde.
3. Gelukkig gingen we dan ogenblikkelijk naar de praktijk over en zagen we dat methoden en properties de kern van iedere klasse blijkt te zijn.
4. Als afsluiter gooiden we dan de `DateTime` klasse open om een voorproefje te krijgen van hoe krachtig een goedgeschreven klasse kan zijn.

Voor je verder gaat raad ik je aan om dit alles goed te laten bezinken én maximaal de komende oefeningen te maken. Het zal de beste manier zijn om de ietwat bizarre wereld van OOP snel eigen te maken.

9.5 Oefeningen

RapportModule

Ontwerp een klasse `Rapport` die je zal tonen wat je graad is gegeven een bepaald behaald percentage. Het enige dat je aan een `Rapport`-object moet kunnen geven is het behaalde percentage (int) dat wordt bijgehouden via een auto-property genaamd `Percentage`. Via een methode `PrintGraad` kan de behaalde graad weergegeven worden, gebaseerd op dit percentage. Dit zijn de mogelijkheden:

- Minder dan 50%: `"Niet geslaagd"`.
- Tussen 50% en 68% (68 incl.): `"Voldoende"`.
- Tussen 68% en 75% (75 incl.): `"Onderscheiding"`.
- Tussen 75% en 85% (85 incl.): `"Grote onderscheiding"`.
- Meer dan 85%: `"Grootste onderscheiding"`.

Test je klasse door enkele objecten in je `main` en te onderzoeken of deze de juiste graden op het scherm printen. Bijvoorbeeld:

```
1 Rapport mijnpunten = new Rapport();
2 mijnpunten.Percentage = 65;
3 mijnpunten.PrintGraad();
```

Bibliotheek*

Boeken in een bibliotheek mogen maximum 14 dagen uitgeleend worden. Schrijf een console-applicatie om de volgende gegevens te tonen door middel van een klasse `BibBoek`:

- de naam van de ontlener, die werd ingelezen (autoproperty).
- de datum van vandaag (autoproperty met private set). Gebruik uiteraard de `DateTime` klasse.
- de datum dat het boek ten laatste terug moet ingeleverd worden (readonly property).

Nummers

Maak een klasse `Nummers`. Deze klasse bevat 2 getallen (type `int`) die via een autoproperty kunnen aangepast worden. Er zijn 4 methoden:

- `Som`: geeft de som van beide getallen terug.
- `Verschil`: geeft het verschil van beide getallen terug.
- `Product`: geeft het product van beide getallen terug.
- `Quotient`: geeft de deling van het eerste door het tweede getal terug; toon een foutboodschap op het scherm indien er een deling door nul zal gebeuren.

Toon in je `main` aan dat je code werkt (op de volgende pagina tonen we een voorbeeld).

Volgende code zou bijvoorbeeld 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
```

BankManager*

Deel 1

We maken een `Rekening` klasse die kan gebruikt worden om de bankrekening van een klant voor te stellen. Deze heeft volgende zaken:

- Een instantievariabele van het type `int` genaamd `balans`. Deze variabele houdt het totale bedrag bij dat op de rekening staat.
- 2 autoproperties van type `string` namelijk `NaamKlant` en `RekeningNummer`.
- 1 readonly property `Balans` die de balans teruggeeft.

Voorzie 3 methoden:

1. `HaalGeldAf`: bepaald bedrag (als parameter type `int`) wordt van de `balans` verwijderd.
2. `StortGeld`: bepaald bedrag (als parameter type `int`) wordt op de rekening gezet en aan `balans` toegevoegd.
3. `ToonInfo`: het totale bedrag op de rekening wordt getoond op het scherm, alsook de naam van de klant en het rekeningnummer (*ook de staat wanneer je deel 2 hebt gemaakt wordt getoond*).

Pas de `HaalGeldAf` methode aan zodat als returntype het bedrag (`int`) wordt teruggegeven. Indien het gevraagde bedrag meer dan de `balans` 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 kon worden gegeven (error die verschijnt: `Rekening leeg nu.`)

Maak 2 instanties van het type `Rekening` aan en toon aan dat je geld van de ene Rekening aan de andere kunt geven, als volgt:

```
1 //rekening 2 geeft 300 euro aan rekening 1
2 rekening1.StortGeld(rekening2.HaalGeldAf(300));
```

Test je klasse.

1. Nieuwe klant aanmaken
2. Status van bestaande klant tonen
3. Geld op een bepaalde Rekening zetten
4. Geld van een bepaalde Rekening afhalen
5. Geld tussen 2 Rekeningen overschrijven

Voorzie extra functionaliteit naar keuze.

Deel 2

Voeg aan de `Rekening`-klasse een autoproperty,genaamd `Staat`, met private set toe van het type `RekeningStaat` toe, dat een enumeratie bevat. De Rekening kan in volgende staten zijn `Geldig`, `Geblokkeerd`. Een rekening is Geldig wanneer een nieuwe rekening wordt geopend.

Maak een bijhorende publieke methode waarmee je de Rekening van staat kunt veranderen. Deze methode (noem ze `VeranderStaat`) vereist geen parameters. Telkens je ze aanroept wordt de staat omgewisseld. Als dus het object momenteel op `Geldig` stond, dan wordt ze nu `Geblokkeerd` en omgekeerd.

Indien een persoon geld van of naar een Geblokkeerde rekening wil sturen dan zal er een error op het scherm verschijnen, namelijk `Gaat niet. Rekening geblokkeerd`. Idem bij de `StortGeld` methode.

Indien de `HaalGeldAf` methode wordt aangeroepen en er werd meer geld afgehaald dan de balans dan zal de rekening ook op Geblokkeerd gezet worden na het verschijnen van de foutboodschap (“Rekening leeg nu”).

Persoon

Ontwerp en implementeer een klasse Persoon met 2 autoproperties (`string`) `Achternaam` en `Voornaam`.

Voeg bovendien een full property `GeboorteDatum` toe (type `DateTime`). De geboortedatum kan enkel waarden tussen 1/1/1990 en vandaag (moment dat code wordt uitgevoerd) liggen. Indien dit niet de situatie is, wordt de huidige datum van uitvoeren gebruikt.

Definieer vervolgens een methode `BerekenLeeftijd`, die op basis van de ingestelde geboortedatum en de huidige datum (dat programma wordt uitgevoerd) de leeftijd van de persoon teruggeeft als `int`.

Verjaardag

Maak een applicatie die aan de gebruiker z'n geboortedatum vraagt (dag/maand, bv 18/3). Vervolgens toont het programma op welke dag hij volgend jaar jarig zal zijn (Monday, Tuesday, enz.) én hoeveel dagen hij nog moet wachten. Uiteraard gebruik je `DateTime` voor deze applicatie.

Je hebt geen eigen klasse nodig. Doel van deze oefening is dat je leert werken met de bestaande `DateTime` en `TimeSpan` klassen. Je mag dus alles in de Main schrijven.

Volgende methode geeft de naam (als `string`) van de dag terug in de taal van het systeem waarop je applicatie draait:

```
System.Globalization.DateTimeFormatInfo.CurrentInfo.GetDayName();
```

"'

Voorbeeld invoer en uitvoer:

```
Wanneer is je verjaardag (d/m, bv 18/3)
20/5
Je bent over 124 dagen jarig op een Dinsdag
```