

1.0.0

H10: Klassen en objecten

OOP Intro



OOP

Object Oriented Programming of korter OOP is een techniek afkomstig van higher level programmeertalen zoals Java, C#, VB.NET, ... en vindt zijn oorsprong bij de programmeertaal Smalltalk, die het eerst de term Object Oriented Programming introduceerde.

In recentere jaren heeft deze techniek echter ook zijn weg gevonden naar scripting talen zoals Python, Ruby, Perl en zelfs PHP.

OOP streeft ernaar om een project zo structureel mogelijk op te bouwen in objecten. Dit heeft voor de programmeur het grote voordeel dat code vanaf nu in logische componenten wordt opgedeeld en veel makkelijker te hergebruiken is.

Om het concept van objecten te illustreren wordt vaak een voorwerp uit het dagelijks leven als voorbeeld gebruikt. Neem bijvoorbeeld een auto. De auto is het object en dit object heeft zowel geassocieerde data als functionaliteit. Deze data stellen eigenschappen of onderdelen van het object voor. Een eigenschap van de auto kan de kleur, de lengte, het aantal kilometers op de teller, of zelfs zijn huidige locatie of snelheid zijn. Deze zaken worden geïmplementeerd met ofwel instantievariabelen, ofwel *properties*. De functionaliteit is iets dat het object kan doen. Deze wordt geïmplementeerd met instantiemethodes. Als je een auto als object ziet, kunnen deze bijvoorbeeld starten, versnellen of remmen zijn.

Auto's zijn een veelgebruikt voorbeeld, maar eigenlijk schrijf je er zelden objecten voor tenzij je een soort simulator wil maken. Objecten hoeven niet altijd tastbare zaken te zijn. Je kan bijvoorbeeld ook een lening bij de bank voorstellen als een object. Dit object heeft dan onder andere volgende eigenschappen:

- een begunstigde, d.w.z. de persoon die de lening heeft aangegaan
- een restsaldo, d.w.z. het bedrag dat je moet betalen
- een looptijd, d.w.z. de periode waarbinnen de lening afbetaald moet zijn
- een intrestvoet, die uiteindelijk bepaalt hoe veel je bovenop het oorspronkelijke bedrag betaalt

Maar met een lening kunnen we ook bepaalde zaken doen die we wel in een computersysteem zouden implementeren. Deze zaken vormen de functionaliteit, dus de verzameling instantiemethodes, van het lening-object:


- een afbetaling verwerken
- de uiteindelijk af te betalen som berekenen
- het afbetaald kapitaal en de afbetaalde interest na een bepaald aantal maanden berekenen

Dus voor zaken die (minstens) zo uiteenlopend zijn als een auto en een lening hebben we een combinatie van data en functionaliteit. Zo zit een object in een programmeertaal er ook uit.

Black-box principe

Een belangrijk concept bij OOP is het **black-box** principe waarbij we de afzonderlijke objecten en hun werking als "zwarte dozen" gaan beschouwen. Dat wil zeggen dat we (meestal) niet gaan kijken hoe ze in elkaar zitten. 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 **publieke interface** of gewoonweg **interface** noemen. Deze zaken kan je 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 **encapsulatie** genoemd. Het doel van OOP is andere programmeurs (en jezelf) zoveel mogelijk af te schermen van de interne werking van je code. Net als bij een auto, moet je alleen weten wat elk stukje van de interface klaarspeelt, maar hoeft je niet te weten hoe dit allemaal is geïmplementeerd.

 Encapsulatie wordt vaak een van de vier pijlers van objectgeoriënteerd programmeren genoemd. De andere zijn abstractie, overerving en polymorfisme. Deze komen allemaal later in de cursus aan bod.

Klassen en objecten

Een elementair aspect binnen OOP is het verschil beheersen 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. Zo hebben we bijvoorbeeld de klasse van de leningen. Er zijn leningen van €50.000, er zijn leningen van €10.000.000. Er zijn er met een intrestvoet van 0.2% en er zijn er met een intrestvoet van 2%. Er zijn er met een looptijd van 5 jaar en er zijn er met een looptijd van 20 jaar. Het zijn allemaal leningen, maar ze hebben allemaal andere eigenschappen.

 Als je vertrouwd bent met relationele databanken: een klasse is vergelijkbaar met een tabel of entiteitstype, terwijl een object vergelijkbaar is met een record of een entiteit.

Objecten van dezelfde soort volgen wel dezelfde regels en hebben in dat opzicht dezelfde functionaliteit. Maar de eigenschappen van object bepalen ook het resultaat van de instantiemethodes. Eerder zagen we dat de uiteindelijk af te betalen som berekend kan worden door een lening. Deze som zal per definitie hoger zijn van een lening met een hoge rentevoet dan voor een lening met een lage rentevoet (als de andere eigenschappen gelijk zijn).

Samengevat:

- **Een klasse** is een beschrijving en verzameling van dingen (objecten) met soortgelijke eigenschappen en gedrag.
- Een individueel **object** is een **instantie** (*exemplaar, voorbeeld, verschijningsvorm*) van een klasse

Klassen en objecten aanmaken

In C# kunnen we geen objecten aanmaken voor we een klasse hebben gedefinieerd die de algemene eigenschappen (properties) en werking (methoden) beschrijft.

Klasse maken

Een heel elementaire klasse heeft de volgende vorm:

```
class ClassName
{
    // hier komen de data en functionaliteit
}
```

We maken hier normaal gebruik van Pascal case. Je kan ook nog enkele toevoegingen doen die niet zuiver data en functionaliteit zijn, maar die houden we voor verder in de cursus.

Volgende code beschrijft de klasse `Auto` in C#

```
class Auto
{
    // data kan bv. nummerplaat, bouwjaar of kilometerstand zijn
    // gedrag kan bv. bepalen van de verkoopwaarde zijn
}
```

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

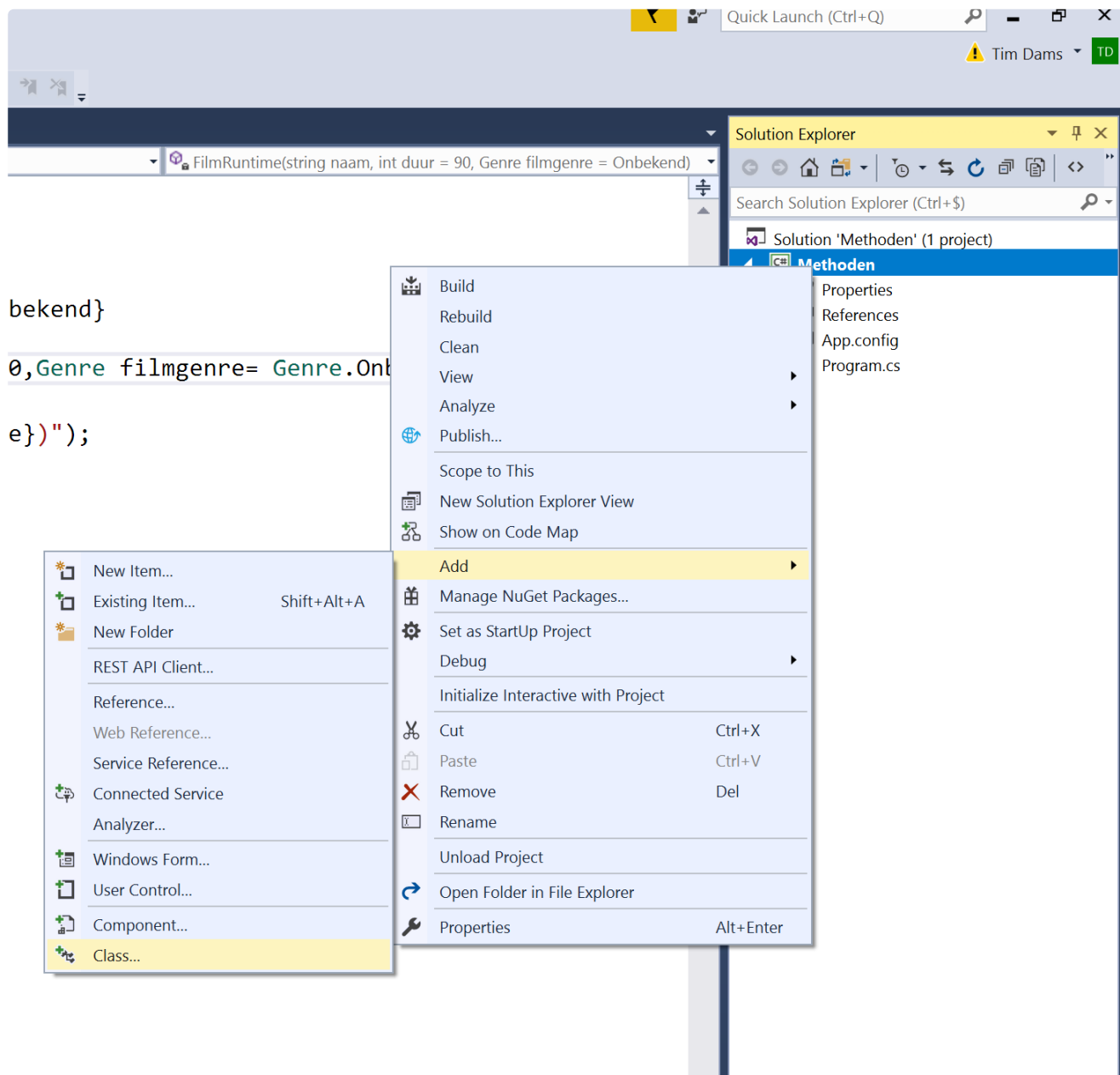
Klassen in Visual Studio

Je kan "eender waar" een klasse aanmaken, maar het is een goede gewoonte om per klasse een apart bestand te gebruiken. **Dit is ook de afspraak die wij zullen volgen.**

- In de solution explorer, rechterklik op je project
- Kies *Add*
- Kies *Class...*
- Geef een goede naam voor je klasse



De naam van je klasse moet voldoen aan de identifier regels die ook gelden voor het aanmaken van variabelen!



Klasse toevoegen in VS

Objecten aanmaken

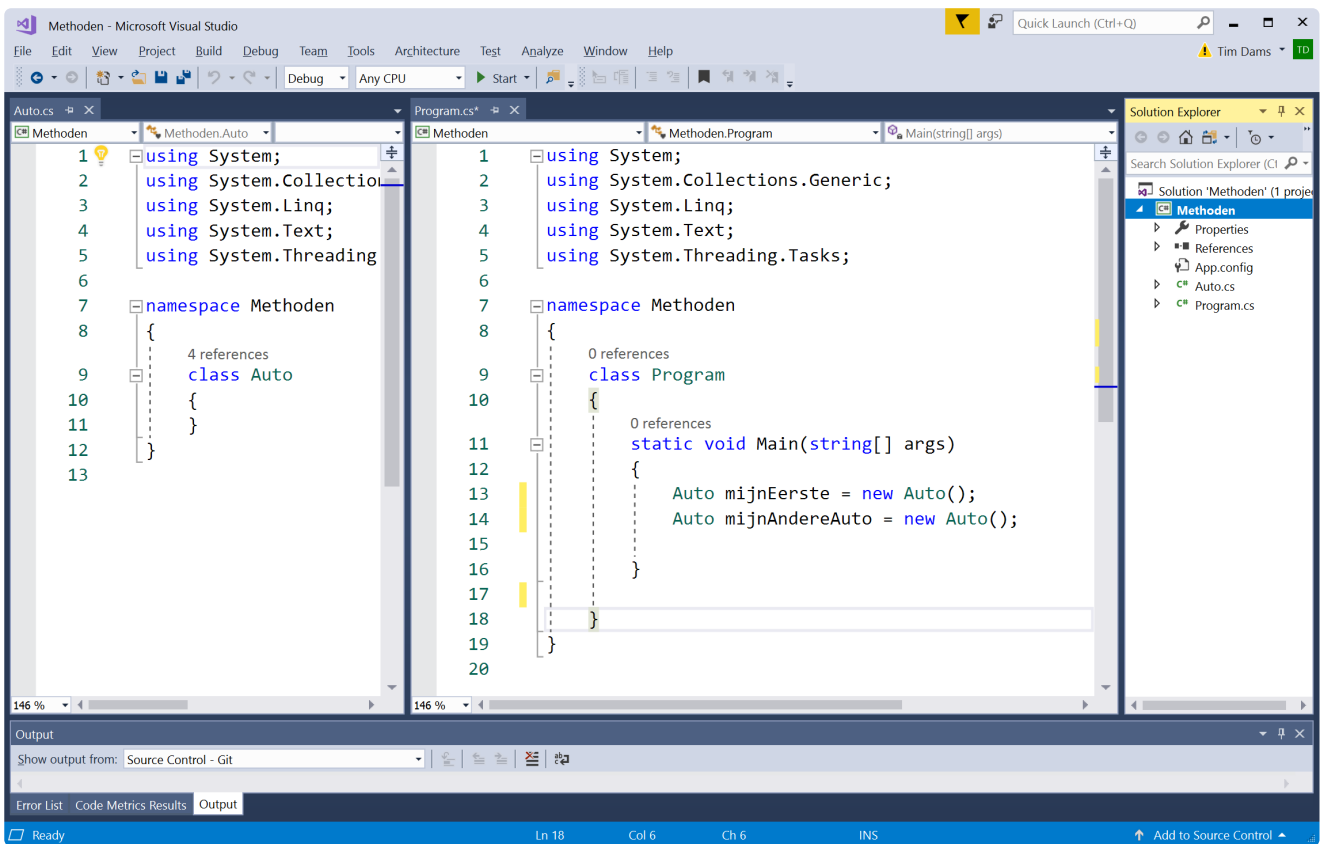
Je kan nu objecten aanmaken van de klasse die je hebt gedefinieerd. De klasse is een type en je kan dus ook variabelen aanmaken die bedoeld zijn om objecten van deze klasse in bij te houden. Je doet dit door eerst een variabele te definiëren met als type de naam van de klasse en vervolgens een object te **instantiëren** met behulp van het `new` keyword:

```
Auto mijnEerste = new Auto();
Auto mijnAndereAuto = new Auto();
```

We hebben nu **twee objecten aangemaakt van het type Auto**.

Let goed op dat je dus op de juiste plekken dit alles doet (bekijk de onderstaande screenshot):

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



basics oop same in vv

i Je hebt dus in het verleden ook al objecten aangemaakt met `new`. Telkens je met `Random` werkt deed je dit al. Dit wil zeggen dat er dus in .NET ergens reeds een voorgeprogrammeerde klasse `Random` bestaat met de interne werking.

DateTime: leren werken met objecten

✓ Kennisclip. Klein verschil met de pagina tot de twee oranje checkboxes, dus lees die goed.

DateTime

Het .NET gegevenstype `DateTime` is de ideale manier om te leren werken met objecten. Het is een nuttig en toegankelijk gegevenstype. Je kan er je iets bij voorstellen, maar het is ook een beetje abstract.

! Waarom spreken we hier over "gegevenstype" en niet over "klasse"? Omdat klassen in .NET reference types zijn. `DateTime` is echter een value type, dus technisch gezien is het een "struct" en geen "klasse". Wij zullen zelf geen structs schrijven, maar het verschil met klassen is uiterst klein in C#. Instanties van zowel klassen als structs zijn objecten.

! Zegt het verschil tussen value types en reference types je niets meer? Kijk dan terug naar [deze pagina](#).

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 via de **constructor**

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:

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


`DateTime` op het begin van regel 1 is het type van de variabele. Dit type drukt uit met wat voor data we te maken hebben. Dit is net dezelfde redenering als bij het gebruik van `string` in `string mijnTekst = "hello world";`

Met constructor

Via de constructor kunnen we beginwaarden meegeven bij het maken van een nieuw object. Er zijn 11 manieren waarop dit kan zoals je [hier kan zien](#).

Enkele voorbeelden:

```
DateTime birthday = new DateTime(1982, 3, 18); //year, month, day
DateTime someMomentInTime = new DateTime(2017, 1, 18, 10, 16, 34 ); //year, month, day, hour,
```

 Je hebt eerder al met constructoren gewerkt: herinner je `new Random()`. Hiermee maakte je eigenlijk een object aan dat willekeurige getallen kon genereren.

DateTime methoden

Ieder `DateTime` object dat je aanmaakt heeft een hoop nuttige methoden.

Add Methods

Deze methoden kan je gebruiken om een bepaalde aantal dagen, uren, minuten en zo voort aan je huidige object toe te voegen. Al deze methoden geven steeds een **nieuw DateTime object** terug dat je moet bewaren wil je er iets mee doen:

- `AddDays`
- `AddHours`
- `AddMilliseconds`
- `AddMinutes`
- `AddMonths`
- `AddSeconds`
- `AddTicks`
- `AddYears`

Een voorbeeld:

```
DateTime timeNow= DateTime.Now;
DateTime nextWeek= timeNow.AddDays(7);
```

(voorgaande kan ook in 1 lijn: `DateTime nextWeek= DateTime.Now.AddDays(7)`)

Uiteraard mag je ook een bestaand object overschrijven met het resultaat van deze methoden:

```
DateTime someTime= new DateTime(2019, 4, 1);  
// much later...  
someTime = someTime.AddYears(10);  
Console.WriteLine(someTime);
```

DateTime properties

Properties zijn een zeer uniek aspect van C#. Ze geven toegang tot de data van een object, maar op een gecontroleerde manier. We zullen deze nog tot in den treure leren maken.

Enkele nuttige properties van `DateTime` zijn:

- `Date`
- `Day`
- `DayOfWeek`
- `DayOfYear`
- `Hour`
- `Millisecond`
- `Minute`
- `Month`
- `Second`
- `Ticks`
- `TimeOfDay`
- `Today`
- `UtcNow`
- `Year`

Properties gebruiken

Alle properties van `DateTime` zijn read-only. Je kan een bestaande datum dus niet aanpassen (maar je kan wel een nieuwe datum baseren op een bestaande datum).

Een voorbeeld:

```

DateTime moment = new DateTime(1999, 1, 13, 3, 57, 32, 11);
// Year gets 1999.
int year = moment.Year;
// Month gets 1 (January).
int month = moment.Month;
// Day gets 13.
int day = moment.Day;
// Hour gets 3.
int hour = moment.Hour;

// Minute gets 57.
int minute = moment.Minute;
// Second gets 32.
int second = moment.Second;
// Millisecond gets 11.
int millisecond = moment.Millisecond;

```

Uiteraard mag je ook deze properties gebruiken om direct naar het scherm te schrijven:

```

DateTime now = DateTime.Now;
Console.WriteLine($"The current day is {now.DayOfWeek}");

```

Datum en tijd formatteren

Je hebt een invloed op hoe `DateTime` objecten naar `string` worden opgezet. Omzetten naar een `string` is functionaliteit van het object. Je doet dit dus via een **objectmethode**. Deze gebruik je zoals de statische methoden van eerder, maar je roept ze op voor een specifiek object.

Je kan bepalen hoe de omzetting naar `string` moet gebeuren door extra *formatter syntax* mee te geven. Dit zie je in volgende voorbeeld:

```

DateTime now = DateTime.Now;
WriteLine(now.ToString("d")); // short date
WriteLine(now.ToString("D")); // long date
WriteLine(now.ToString("F")); // full date and time
WriteLine(now.ToString("M")); // month and day
WriteLine(now.ToString("o")); // date en time separated by T and time zone at the end
WriteLine(now.ToString("R")); // RFC1123 date and time
WriteLine(now.ToString("t")); // short time
WriteLine(now.ToString("T")); // long time
WriteLine(now.ToString("Y")); // year and month

```

Custom format

Wil je nog meer controle over de output dan kan je ook zelf je formaat specificeren.

Dit wordt hier volledig uit de doeken gedaan.

Localized time

De manier waarop `DateTime` objecten worden getoond (via `ToString`) is afhankelijk van de landinstellingen van je systeem. Soms wil je dit echter op een andere manier tonen. Je doet dit door mee te geven volgens welke **culture** de tijd en datum getoond moet worden.

Dit vereist dat je eerst een `CultureInfo` object aanmaakt en dat je dan meegeeft:

```
DateTime now = DateTime.Now;  
CultureInfo russianCI = new CultureInfo("ru-RU");  
Console.WriteLine($"Current time in Russian style is: {now.ToString("F", russianCI)}");
```

Culture names

Een lijst van alle cultures in .NET kan je [hier terugvinden](#).

Opgelet, enkel indien een specifieke culture op je computer staat geïnstalleerd zal je deze kunnen gebruiken.

`CultureInfo` is een klasse waar je je misschien al iets minder bij kan voorstellen dan `DateTime`. Een `CultureInfo` stelt een aantal afspraken voor een bepaalde cultuur voor. De data is de combinatie van taal en regio. De functionaliteit bestaat er dan in om zaken zoals de munt, het gebruikte formaat voor datums,... in die taal en die regio te produceren. Ook hier geldt dus het black box principe: je kan code schrijven die bijvoorbeeld het juiste datumformaat voor Rusland gebruikt wanneer je zelf niet weet wat dat formaat is.

Static method

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.

Parsing time

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

```
string date_string = "8/11/2016"; //dit zou dus ook door gebruiker kunnen ingetypt zijn  
DateTime dt = DateTime.Parse(date_string);  
Console.WriteLine(dt);
```


Zoals je ziet roepen we `Parse` aan op `DateTime` en dus niet op een specifiek object. Dit is logisch omdat je geen bestaande datum nodig hebt om een nieuwe datum te parsen.

IsLeapYear

Deze nuttige methode geeft een `bool` terug om aan te geven het meegegeven object eens schrikkeljaar is of niet:

```
DateTime today = DateTime.Now;
bool isLeap = DateTime.IsLeapYear(today.Year);
if(isLeap == true) {
    Console.WriteLine("This year is a leap year");
}
```

Dit is logisch omdat je geen volledige bestaande datum nodig hebt. Je wil gewoon iets zeggen over een jaartal, terwijl een `DateTime` ook een dag, maand, uur,... heeft. Dus dit heeft niet veel te maken met een specifieke `DateTime`, maar heeft duidelijk wel met de klasse te maken.

 Af en toe is het een kwestie van smaak van de auteurs of een methode statisch is of niet. Maar meestal is er een duidelijke "beste" keuze.

TimeSpan

Je kan `DateTime` objecten ook bij mekaar optellen en aftrekken. Deze bewerking geeft echter **geen** `DateTime` object terug, maar een `TimeSpan` object. Dit is een object dat dus aangeeft hoe groot het verschil is tussen de 2 `DateTime` objecten:

```
DateTime today = DateTime.Today;
DateTime borodino_battle = new DateTime(1812, 9, 7);
TimeSpan diff = today - borodino_battle;
WriteLine("{0} days have passed since the Battle of Borodino.", diff.TotalDays);
```

Oefening

Klokje

Maak een applicatie die bestaat uit een oneindige loop. De loop zal iedere seconde pauzeren:

`System.Threading.Thread.Sleep(1000);` . Vervolgens wordt het scherm leeg gemaakt en wordt de huidige tijd getoond. Merk op dat ENKEL de tijd wordt getoond, niet de datum.

Verjaardag

Maak een applicatie die aan de gebruiker vraagt op welke dag hij jarig is. Toon vervolgens over hoeveel dagen z'n verjaardag dan zal zijn.

Enumeraties: nog een eigen datatype

Enum datatypes

Met klassen kan je dus complexe concepten zoals een lening modelleren in code. Soms hoeft het niet zo complex, maar is het toch nuttig om wat meer structuur te voorzien dan met heel brede datatypes zoals `int` en `string`. Met een `string` kan je bijvoorbeeld om het even welke tekst voorstellen, dus oneindig veel mogelijkheden. Met een `byte` kan je een getal tussen 0 en 255 voorstellen. Met de andere getaltypes kan je een getal in een ander bereik voorstellen. Soms matchen de waarden die je kan voorstellen niet met de mogelijkheden in het probleemdomein.

Als je de mogelijkheden in het probleemdomein op voorhand kan vastleggen, is het vaak handig gebruik te maken van een **enumeratie (enum)**. Een enumeratie is **een opsomming van alle mogelijke waarden** voor een variabele van een bepaald type.

Een voorbeeld: weekdays

Stel dat je een programma moet schrijven dat afhankelijk van de dag van de week iets anders moet doen. Zonder enums zou je dit 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

Beide hebben tekortkomingen, omdat ze niet goed genoeg overeenstemmen met de mogelijkheden in de werkelijkheid.

Slechte oplossing 1: Met ints

De waarde van de dag staat in een variabele `int dagKeuze`. We bewaren er 1 in 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. ? We kunnen daar wel conditionele code voor voorzien, maar de compiler zal ons niet waarschuwen als we dat vergeten!
- De code is niet erg leesbaar. `dagKeuze==2` ? Was dat nu maandag, dinsdag of woensdag (want misschien beginnen we te tellen vanaf zondag, of misschien beginnen we te tellen vanaf 0). Je kan wel afspraken maken binnen je team, maar het is altijd mogelijk een afspraak te vergeten.

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.WritLine("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"` bewaart dan zal de `if` al niet werken. Iedere schrijffout of variant zal falen.

Enumeraties: het beste van beide werelden

Het keyword `enum` geeft aan dat we een nieuw datatype maken. Gelijkaardig aan wat we voor een klasse doen, dus. Wanneer we dit nieuwe type hebben gedefinieerd kunnen we dan ook variabelen van dit nieuwe type aanmaken (dat is waarom we spreken van een "datatype"). Anders dan bij een klasse beperken we tot alleen de opgesomde opties.

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, bijvoorbeeld: `Console.ForegroundColor = ConsoleColor.Red;`

Achter de schermen worden waarden van een enum type nog steeds voorgesteld als getallen. Maar je gebruikt deze getallen niet rechtstreeks en dat verkleint de risico's.

Zelf enum maken

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

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

Stap 1: het type definiëren

We maken eerst een enum type aan. **Wij zullen dit doen in een aparte file met dezelfde naam als het nieuwe datatype.** Bijvoorbeeld:

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

Vanaf nu kan je variabelen van het type `Weekdagen` aanmaken. Merk op dat er **geen puntkomma** voorkomt. **De syntax is ongeveer dezelfde als die voor de definitie van een klasse.**

Stap 2: variabelen van het type aanmaken en gebruiken.

We kunnen nu variabelen van het type `Weekdagen` aanmaken. Bijvoorbeeld:

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

En vervolgens kunnen we waarden aan deze variabelen toewijzen als volgt:

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

Kortom: we hebben variabelen zoals we gewoon zijn, het enige verschil is dat we nu beperkt zijn in de waarden die we kunnen toewijzen. Deze kunnen enkel de waarden zijn die in het type gedefinieerd werden. De code is nu ook een pak leesbaarder geworden.

Waarden van een enum type inlezen

Eens je met enums aan het werken bent in je programma, kan je gerust zijn dat deze een geldige waarde bevatten. Er is alleen het probleem dat je soms een van de mogelijkheden moet krijgen van de gebruiker. Een waarde van een enum type rechtstreeks intypen gaat niet. Net zoals wanneer je een getal inleest, is er een omzetting nodig. Het verschil is dat we hier een dubbele omzetting doen: van tekst naar getal en dan van getal naar enum waarde.

Je kan dit als volgt doen:

```
Weekdagen keuze;  
Console.WriteLine("Welke dag is het vandaag?");  
Console.WriteLine($"{(int) Weekdagen.Maandag}. {Weekdagen.Maandag}");  
Console.WriteLine($"{(int) Weekdagen.Dinsdag}. {Weekdagen.Dinsdag}");  
// enzovoort  
keuze = (Weekdagen) Convert.ToInt32(Console.ReadLine());
```

Dit werkt en je hoeft de achterliggende getallen voor de waarden niet te kennen. Er zijn meer geavanceerde mogelijkheden, maar dit volstaat op dit punt.

Klassen en objecten weergeven deel 1

✓ Kennisclip

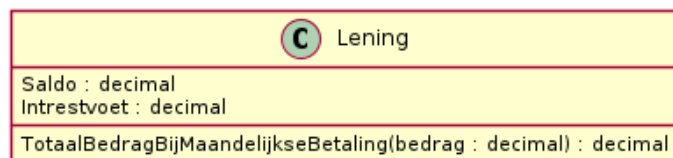
Klassen voorstellen

De data en functionaliteit van een klasse, en ook haar relatie tot andere klassen, wordt vaak voorgesteld in een **UML-klassendiagram**. Hoe je verschillende klassen met elkaar verbindt, houden we voor iets verderop. Ook enkele andere details volgen later. Wat we wel al kunnen vertellen:

- elke klasse wordt voorgesteld als een rechthoek, met bovenaan in die rechthoek de naam van de klasse
- in een tweede vakje worden dan de eigenschappen gegeven, gewoonlijk met hun datatype
- in een derde vakje worden dan de methoden gegeven, met hun parameters en hun returntype

i Het (return) type kan voor de naam van een attribuut of methode staan (zoals in C#), of het kan helemaal achteraan staan, voorafgegaan door een dubbele punt (zoals in TypeScript).

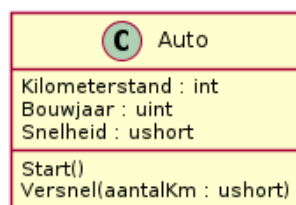
Voorbeeld 1: Lening



De "C" bovenaan staat voor "class". De meeste programma's tekenen deze niet.

Dit vertelt ons dat een object van klasse `Lening` beschikt over een saldo en een intrestvoet en het vertelt ons ook welke datatypes we gebruiken om deze voor te stellen. Bovendien kan een `Lening` zelf bepalen hoe veel we in totaal zullen moeten betalen als we elke maand een vast bedrag aflossen. Dit is functionaliteit van het object, die we zullen implementeren met een instantiemethode.

Voorbeeld 2: Auto

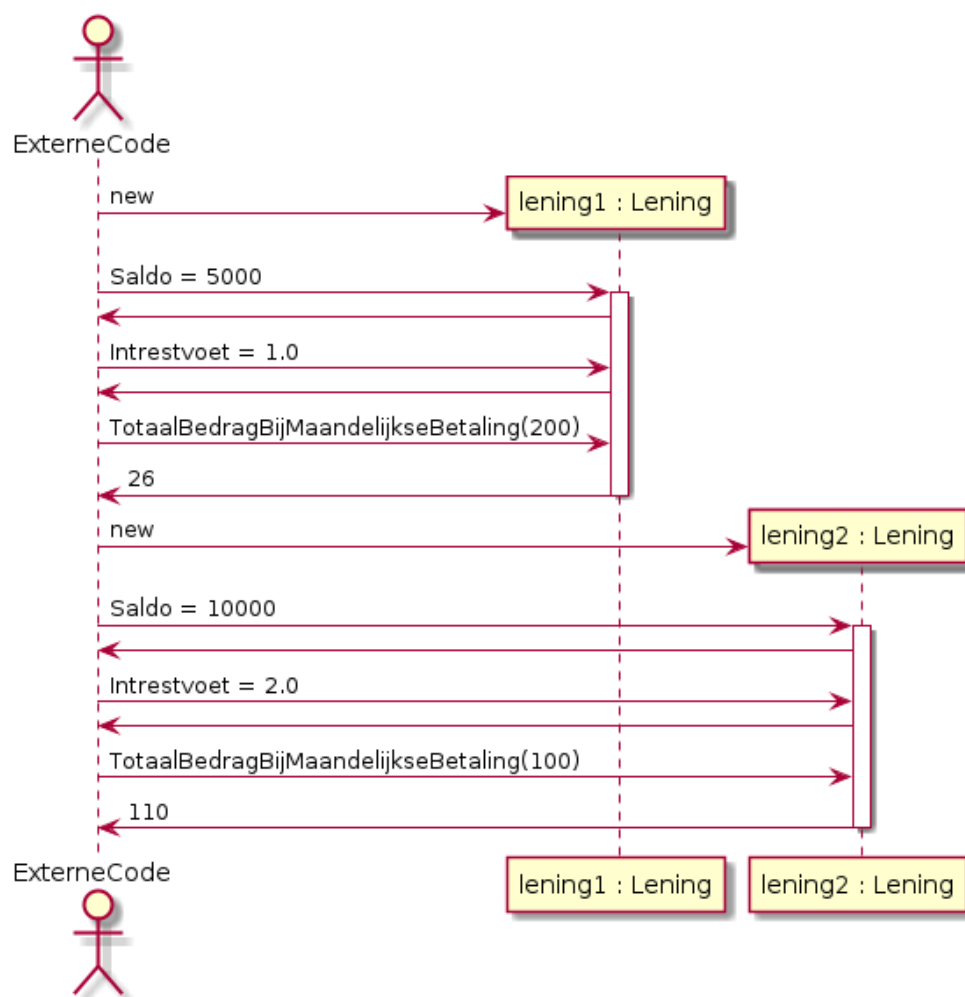


Dit vertelt ons dat elke `Auto` een eigen kilometerstand, bouwjaar en huidige snelheid heeft. Het is mogelijk de `Auto` te laten starten en om hem te laten versnellen met een geven aantal kilometer per uur.

Objecten voorstellen

Een UML-klassendiagram dient voor... klassen. Als je wil kijken naar objecten, gebruik je een **sequentiediagram**. We gaan hier nog niet erg diep op in, maar we maken alvast duidelijk dat je **op een klassendiagram geen objecten** ziet.

Een voorbeeld van een sequentiediagram voor een programma dat twee leningen aanmaakt (bijvoorbeeld een programma in een bank om een simulatie te maken voor een klant):



Het figuurtje met als naam `ExterneCode` stelt iets voor dat ons eigenlijk niet interesseert. Wat de pijlen en de berichten daarop technisch betekenen, maakt voorlopig ook niet zo uit. Het belangrijkste hier is dat er twee objecten van de klasse `Lening` zijn. Dit wordt extra in de verf gezet omdat ze twee verschillende namen hebben (`lening1` en `lening2`), maar allebei hetzelfde type na de dubbele punt (`Lening`). Op een klassediagram ga je nooit twee keer een box voor dezelfde klasse zien.

Attributen



Kennisclip (let op: de demonstratie SchoolAdmin is verouderd en wordt anders aangepakt in 2022. De tekst heeft voorrang!)

Attributen, ook **velden** of **instantievariabelen** genoemd, zijn stukjes data die je bijhoudt in objecten. Ze stellen informatie voor die deel uitmaakt van een (object van een) klasse. Ze werken zoals de variabelen die je al kent, maar hun scope is een klasse of een object van een klasse, afhankelijk van de vraag of ze **static** zijn of niet. Door gebruik te maken van attributen, kunnen we stukjes data die samen horen ook samen houden op het niveau van de code. Alle data die samen hoort netjes groeperen en op een gestructureerd toegankelijk maken valt onder het begrip **encapsulatie** dat reeds eerder aan bod kwam.



Attributen behoren tot een algemenere categorie onderdelen van objecten genaamd **members**.

Basisvoorbeelden

Een typisch voorbeeld van een klasse is `Auto`. Er zijn verschillende stukjes data die deel kunnen uitmaken van één auto: de kilometerstand, het benzinepeil, de datum van het laatste onderhoud,...

Een reeds gekende manier om verwante informatie bij te houden is met behulp van "gesynchroniseerde" arrays, d.w.z. arrays die verwante data bijhouden op overeenkomstige posities. Onderstaand voorbeeld toont dit voor auto's:

```

class Program {
    public static void Main() {
        int aantalAutos = 3;
        int[] kilometers = new int[aantalAutos];
        double[] benzine = new double[aantalAutos];
        DateTime[] onderhoud = new DateTime[aantalAutos];
        for (int i = 0; i < aantalAutos; i++) {
            Console.WriteLine($"Kilometerstand van auto {i+1}?");
            kilometers[i] = Convert.ToInt32(Console.ReadLine());
            Console.WriteLine($"Benzinepeil van auto {i+1}?");
            benzine[i] = Convert.ToDouble(Console.ReadLine());
            Console.WriteLine($"Jaar recentste onderhoud auto {i+1}?");
            int jaar = Convert.ToInt32(Console.ReadLine());
            Console.WriteLine($"Maand recentste onderhoud auto {i+1}?");
            int maand = Convert.ToInt32(Console.ReadLine());
            Console.WriteLine($"Dag recentste onderhoud auto {i+1}?");
            int dag = Convert.ToInt32(Console.ReadLine());
            onderhoud[i] = new DateTime(jaar, maand, dag);
        }
        // later in de code
        for (int i = 0; i < aantalAutos; i++) {

            PrintOnderhoudsrapport(kilometers[i], benzine[i], onderhoud[i]);

        }
    }
}

```

Als we nu een methode willen uitvoeren die iets doet met alle informatie over een auto (bijvoorbeeld een onderhoudsrapport afprinten), moeten we drie waarden meegeven. Het is ordelijker deze zaken bij te houden in een object van klasse `Auto` als volgt:

```

class Auto {
    public int Kilometers;
    public double Benzine;
    public DateTime LaatsteOnderhoud;
}

```

Al deze velden zijn voorlopig `public`. Dat hoeft niet absoluut, maar het vergemakkelijkt de presentatie. Verdere opties volgen snel. Het is ook een afspraak om publieke velden met een hoofdletter te schrijven. Met deze voorstelling kunnen we dit nu doen:

```

class Program {
    public static void Main() {
        int aantalAutos = 3;
        Auto[] autos = new Auto[aantalAutos];
        for (int i = 0; i < aantalAutos; i++) {
            Auto nieuweAuto = new Auto();
            autos[i] = nieuweAuto;
            Console.WriteLine($"Kilometerstand van auto {i+1}?");
            nieuweAuto.Kilometers = Convert.ToInt32(Console.ReadLine());

            Console.WriteLine($"Benzinepeil van auto {i+1}?");
            nieuweAuto.Benzine = Convert.ToDouble(Console.ReadLine());
            Console.WriteLine($"Jaar recentste onderhoud auto {i+1}?");
            int jaar = Convert.ToInt32(Console.ReadLine());
            Console.WriteLine($"Maand recentste onderhoud auto {i+1}?");
            int maand = Convert.ToInt32(Console.ReadLine());
            Console.WriteLine($"Dag recentste onderhoud auto {i+1}?");
            int dag = Convert.ToInt32(Console.ReadLine());
            nieuweAuto.LaatsteOnderhoud = new DateTime(jaar, maand, dag);
        }
        // later in de code
        for (int i = 0; i < aantalAutos; i++) {
            PrintOnderhoudsrapport(autos[i]);
        }
    }
}

```

Je ziet hier al enkele voordelen van encapsulatie:

- je hoeft niet bij te houden hoe de informatie verspreid is over meerdere plaatsen
- als we meer informatie over auto's (bv. het oliepeil) in het onderhoudsrapport steken, hoeven we onze calls van `PrintOnderhoudsrapport` niet aan te passen
 - een kenmerk van goede code is dat wijzigingen typisch geen grote wijzigingen vereisen

Beginwaarden

Een veld krijgt normaal de defaultwaarde voor zijn type. [Defaultwaarden](#) hebben we reeds gezien. Het is mogelijk de beginwaarde aan te passen met de syntax voor een toekenning:

```

class Auto {
    public int Kilometers = 5; // in de fabriek vinden bv. een aantal testen plaats
    public double Benzine = 10; // nieuwe auto's moeten kunnen rijden
    public DateTime LaatsteOnderhoud = DateTime.Now;
}

```

Nu hebben nieuwe auto's standaard 5 km op de teller staan, enzovoort. Merk op: deze waarde wordt voor elk nieuw object opnieuw berekend. Als je dus twee auto's aanmaakt in je programma, zullen zij beide een **verschillende** datum van het laatste onderhoud bevatten.

static attributen

Iets dat `static` is, hoort niet bij de objecten, maar wel bij de hele klasse. Bijvoorbeeld: voor auto's worden de milieunormen na een paar jaar strenger. Er is een vaste set milieunormen en de te halen milieunorm wordt vastgelegd voor **alle** auto's. Daarom zouden we de milieunorm als volgt kunnen bijhouden:

```
enum MilieuNormen {  
    Euro1, Euro2, Euro3, Euro4, Euro5, Euro6  
}  
  
class Auto {  
    public static MilieuNormen HuidigeNorm;  
    // rest van de code voor Auto  
}
```

⚠ Herhaal: `static` **betekent niet "onveranderlijk" of "vast"**. Het betekent dat iets op niveau van de klasse werkt en niet op niveau van de objecten van die klasse.

Opdracht: SchoolAdmin

Deze opdracht maak je tijdens de les. Als je de les niet kan bijwonen, volg je de demonstratie in de kennisclip.

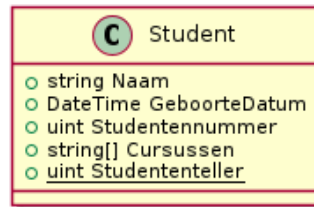
Doelstelling

We willen een programma maken dat ons helpt beheren wie is ingeschreven, welke cijfers behaald zijn, enzovoort. Hiervoor brengen we de concepten en functionaliteit die we gebruiken in kaart en stellen we ze voor in code.

We doen dit in een **apart project** `SchoolAdmin`. We gebruiken dus niet het project `IndividueleOefeningen`. Je volgt om dit project op te starten dezelfde stappen als eerder. Let hierbij goed op dat je dit project niet in een bestaande Git repository maakt. Je merkt direct dat je in een Git repository staat als je de nieuwe map `SchoolAdmin` opent in VSC en Git bash tussen haakjes `(main)` of `(master)` toont. Vraag in dat geval aan je lector om na te kijken waar je het project beter *wel* zet.

Klasse Student

We maken een klasse `Student`, met publieke attributen voor de naam, geboortedatum, het studentnummer en de gevolgde cursussen. Elke student kan vijf cursussen volgen. We houden ook bij *hoe veel* studenten er zijn via een attribuut.



In de `Main` methode maken we enkele studenten. Deze zijn als volgt:

- Said Aziz, geboren 1 juni 2000, volgt Programmeren en Databanken.
- Mieke Vermeulen, geboren 1 januari 1998, volgt Communicatie.

Delen via GitLab

Wanneer je de opdracht gevolgd hebt, deel je deze via Gitlab. Je maakt hiervoor een nieuw project aan op Gitlab, maar wel in dezelfde groep als eerder (die met jouw naam).

Negeer de mappen `bin` en `obj` in je versiebeheer.

Methoden

✓ Kennisclip

Instantiemethoden, ook **objectmethoden** genoemd, weerspiegelen staan toe om functionaliteit toe te voegen aan objecten van een bepaalde klasse. Soms wordt ook gezegd dat ze "**gedrag**" van de objecten voorzien. Ze verschillen van statische methoden omdat ze niet alleen gebruik kunnen maken van statische onderdelen van klassen, maar ook van het object waar ze zelf bij horen.

i Methoden behoren tot een algemenere categorie onderdelen van objecten genaamd **members**.

Basisvoorbeelden

We gaan verder met de klasse `Auto`. We willen bijvoorbeeld een applicatie voor de opvolging van deelauto's (Cambio, Poppy, etc.) schrijven. Er zijn verschillende soorten functionaliteit die je kan koppelen aan één auto:

- voltanken
- rijden
- op onderhoud gaan
- verkoopprijs bepalen

i Is het de auto die deze zaken doet, of is het een persoon? In werkelijkheid is het natuurlijk dat laatste. Maar de functionaliteit is wel veel sterker gelinkt aan auto's dan aan personen en misschien interesseert de persoon die de handeling uitvoert ons niet eens.

Je doet dit met objectmethoden. Deze lijken erg op `static` methoden, maar ze hebben toegang tot het object waarop ze zijn toegepast.

Een simpele implementatie van dit gedrag zie je hier:

```

class Auto {

    // objectvariabelen van eerder zijn er nog

    public void Voltanken()
    {
        Benzine = 50.0; // we veronderstellen even dat dat het maximum is
    }

    public void Rijden(int aantalKilometers)
    {
        Kilometers += aantalKilometers;
        Benzine -= 5.0 * (aantalKilometers/100.0);
    }

    public void Onderhouden()
    {
        LaatsteOnderhoud = DateTime.Now;
    }

    public double VerkoopsprijsBepalen()
    {
        return Math.Max(10000 * (1 - Kilometers / 200000.0), 1000);
    }
}

```

⚠ Bovenstaande code is kort om didactische redenen. Er wordt niet gecontroleerd dat je benzinepeil altijd minstens 0l is, er wordt verondersteld dat de capaciteit van je tank 50l is,...

ℹ Voor de duidelijkheid kan je het woordje `this` toevoegen om het huidige object expliciet voor te stellen. Het wordt sterk aangeraden dat je dit doet. Je code wordt er beter leesbaar door.

Gebruik

Om een objectmethode te gebruiken, hebben we een object nodig. We schrijven dan de naam van het object, gevolgd door een punt en een methodeoproep.

```
// in Program.cs
public static void DemonstreerAttributen() {
    Auto auto1 = new Auto();
    Auto auto2 = new Auto();
    auto1.Voltanken();
    auto1.Rijden(5);
    auto1.Rijden(10);
    auto1.Rijden(20);
    Console.WriteLine(auto1.Kilometers);
    Console.WriteLine(auto2.Kilometers);
}
```

Het gedrag van een object kan afhangen van de waarde van de instantievariabelen. Zo zal de verkoopswaarde van `auto1` iets lager liggen dan die van `auto2`. Dat komt omdat `this.Kilometers` deel uitmaakt van de berekening van de verkoopprijs. Ook dit valt onder het principe van **encapsulatie**: er vindt een berekening plaats "onder de motorkap". We hoeven niet te weten hoe de prijs berekend wordt, elk object weet van zichzelf hoe het de prijs berekent.

static methodes

Een statische methode is een methode die wel bij de klasse hoort, maar niet te maken heeft met een specifiek object van die klasse. We gebruiken terug de euronorm als voorbeeld:

```

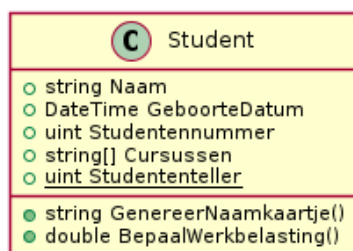
enum MilieuNormen {
    Euro1, Euro2, Euro3
}

class Auto {
    public static MilieuNormen HuidigeNorm;
    // rest van de code voor Auto
    public static void VerklaarNorm(MilieuNormen norm) {
        int jaartal;
        double CO;
        switch (norm) {
            case MilieuNormen.Euro1:
                jaartal = 1992;
                CO = 1.0;
                break;
            case MilieuNormen.Euro2:
                jaartal = 1996;
                CO = 1.0;
                break;
            case MilieuNormen.Euro3:
                jaartal = 2000;
                CO = 0.64;
                break;
            default:
                jaartal = -1;
                CO = -1;
                break;
        }
        Console.WriteLine($"Geïntroduceerd in {jaartal}");
        Console.WriteLine($"{CO} gram CO per km");
    }
}

```

SchoolAdmin project

We breiden onze klasse `Student` uit zodat ze overeenstemt met volgend diagram:



Toevoegen naamkaartje

Het naamkaartje is een stuk tekst, bestaande uit de naam van de student, gevolgd door `(STUDENT)`.

Toevoegen werkbelasting

Dit is 10u per week per cursus.

Push je vooruitgang naar Gitlab!

Access modifiers

✓ Kennisclip

Access modifiers bepalen welke code door welke andere code mag worden uitgevoerd of aangepast. We hebben al een aantal dingen `public` gemaakt bij wijze van demonstratie. Dit is handig om voorbeeldjes te tonen, maar in code van hoge kwaliteit, denk je grondig na voor je dit doet.

Public en private access modifiers

De **access modifier** geeft aan hoe zichtbaar een bepaald deel van de klasse is **voor code buiten de klasse zelf**. Wanneer je niet wilt dat "van buitenuit" (bv in `Main`, terwijl je een andere klasse dan `Program` schrijft) 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 `Auto` eens wat gebeurt wanneer je `public` verwijdt voor een van de methodes. Inderdaad, je krijgt een foutmelding. Lees deze. Ze zegt dat de methode die je wil oproepen wel bestaat, maar niet gebruikt mag worden. **Dat komt omdat je testcode in de klasse `Program` staat en je methode deel uitmaakt van een andere klasse (meerbepaald `Auto`).**

Als je duidelijk wil maken dat bepaalde code niet van buitenaf aangeroepen kan worden, schrijf dan `private` in plaats van `public`. Er zijn nog tussenliggende niveaus waar we later op ingaan en als je geen van beide modifiers schrijft, kan het zijn dat je code op zo'n tussenliggend niveau terecht komt. Als beginnende programmeur maak je er best een gewoonte van duidelijk te kiezen voor `public` of `private`.

Reden van private

Waarom zou je bepaalde zaken `private` maken? Het antwoord is opnieuw encapsulatie.

Neem als voorbeeld de kilometerstand. Het is wettelijk verboden een kilometerstand zomaar aan te passen. Hij mag alleen veranderen doordat er met een auto gereden wordt. Dan is het logisch dat dit ook alleen maar op deze manier kan voor onze auto-objecten. We kunnen dit realiseren door `kilometers` van `public` naar `private` te wijzigen. Dan kunnen we de kilometerstand nog wijzigen, maar enkel door de (publieke) methode `Rijden` te gebruiken. Als we hem dan nog willen kunnen bekijken (maar niet rechtstreeks wijzigen), kunnen we een extra (publieke!) methode `ToonKilometerstand` voorzien.

i Volgens de conventie maken we dan van de grote "K" in `kilometers` een kleine "k", omdat publieke members met Pascal case genoteerd worden en private members met camel case.

- i De richtlijnen rond naamgeving van Microsoft met betrekking tot attributen, methoden,... vind je [hier](#) terug.

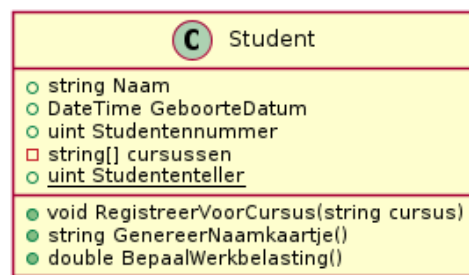
We kunnen ook methoden `private` maken. Dit gebeurt niet zo vaak als bij attributen, maar het kan wel. Dit doen we bijvoorbeeld als het gaat om een hulpmethode die binnen de klasse nuttig is, maar buiten de klasse fout gebruikt zou kunnen worden. Een typisch voorbeeld: een stukje code dat letterlijk herhaald wordt in twee of meer verschillende publieke methoden. In plaats van deze code te kopiëren, zonderen we ze af in een hulpmethode. Zo hoeven we eventuele bugfixes,... geen twee keer te doen.

- i Een studente vroeg in een van de afgelopen jaren: "Kunnen we niet gewoon afspreken dat we van sommige zaken afblijven?" In principe wel. Python doet het min of meer zo. Langs de andere kant: als wij meedelen dat de examenvragen op een publieke website staan en dat je er niet naartoe mag surfen, zou niemand dat dan doen? Private velden aanpassen kan soms een goed idee lijken op korte termijn, maar een project saboteren op langere termijn.

SchoolAdmin project

Privé maken (en hernoemen) `Cursussen`

Het is moeilijk afspraken te maken over hoe `Cursussen` gebruikt mag worden. Iemand zou zich altijd kunnen vergissen en een cursus invullen op een positie die niet vrij is, of op een andere vrije positie dan de eerste. Daarom leggen we de regels rond dit attribuut vast **in de klasse**. We doen dit door over te schakelen op volgende structuur:



`Cursussen` wordt `cursussen` met een kleine letter en wordt `private`. Dit wordt aangegeven door het rode vierkantje in plaats van een groen bolletje.


- i In C# schrijven we private members in camel case. Daarom wordt het attribuut hernoemd.


Hierbij gaat `RegistreerVoorCursus` zelf op zoek naar de eerste vrije positie. Als zo'n positie gevonden wordt, wordt de student ingeschreven voor de cursus.

Push je vooruitgang naar Gitlab!

Properties

Properties zijn een feature van C# om de leesbaarheid van code te verhogen. Ze zien er uit zoals attributen, maar werken zoals methoden.

 Properties behoren tot een algemenere categorie onderdelen van objecten genaamd **members**.

 [Kenniscлип voor deze inhoud](#). De camerabeelden zijn wat wazig, maar de schermopname is in orde.

Properties

In dit hoofdstuk bespreken we eerst waarom properties nuttig zijn. Vervolgens bespreken we de 2 soorten properties die er bestaan:

1. Full properties
2. Auto properties

In een wereld zonder properties

Stel dat we volgende klasse hebben:

```
public class Auto
{
    private int kilometers;
    private double benzine;
}
```

Stel nu dat we het benzinepeil van een auto als volgt proberen aanpassen:

```
Auto auto = new Auto();
auto.benzine += 10; //DIT ZAL DUS NIET WERKEN, daar benzine private is.
```

Misschien is de eerdere methode `TankVol()` te beperkt en willen we wel een willekeurige hoeveelheid benzine kunnen toevoegen of verwijderen, zo lang we niet minder dan 0l of meer dan 50l in de tank doen.

Een eerste mogelijkheid is om hier methodes voor te schrijven:

```

public class Auto
{
    private int kilometers;
    private double benzine;
    public double GetBenzine() {
        return this.benzine;
    }
    public void SetBenzine(double waarde) {
        if(waarde >= 0 && waarde <= 50) {
            this.benzine = waarde;
        }
    }
}

```

Dit gaat. De methodes zijn public, dus we kunnen ze overal oproepen. Bovendien verhindert de `SetBenzine` -methode ongeldige waarden. Het nadeel is dat de syntax om deze methodes te gebruiken zwaar is. Met publieke attributen konden we dit doen:

```

Auto auto = new Auto();
auto.Benzine += 10;

```

Met de zogenaamde **getter** en **setter** moeten we dit doen:

```

Auto auto = new Auto();
auto.SetBenzine(auto.GetBenzine() + 10);

```

Het lijkt niet zo veel, maar code stapelt zich op doorheen de tijd. Properties lossen dit probleem op. Ze zorgen ervoor dat we kunnen "doen alsof" we publieke velden hebben, maar dezelfde hoeveelheid controle kunnen uitoefenen als met getters en setters.

Full properties

Een **full property** ziet er als volgt uit:

```

class Auto
{
    private int kilometers;
    private double benzine;

    public double Benzine
    {
        get
        {
            return benzine;
        }
        set
        {
            benzine = value;
        }
    }
}

```

Dankzij deze code kunnen we nu elders dit doen:

```


Auto auto = new Auto();
auto.Benzine = 20; //set
Console.WriteLine($"Het benzinepeil is {auto.Benzine}"); //get


```

Vergelijk dit met de vorige alinea waar we dit met Get en Set methoden moesten doen. Deze property syntax is veel eenvoudiger in het gebruik.

We zullen de property nu stuk per stuk analyseren:

- `public double Benzine` : Merk op dat we `Benzine` met een hoofdletter schrijven. Vaak wordt gekozen voor dezelfde naam als de variabele die we afschermen (in dit geval `benzine` met een kleine "b"), maar dan in Pascal case. Dat is niet strikt noodzakelijk. Je zou ook `Naft` kunnen schrijven. `public` en `double` staan er om dezelfde reden als in de oplossing met methodes `GetBenzine()` en `SetBenzine()` : we willen deze property buiten de klasse kunnen gebruiken en als we hem opvragen, krijgen we een `double` .
- `{}`: 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 benzine` maar dit mag even goed bijvoorbeeld `return 4` of een hele reeks berekeningen zijn. Eigenlijk werkt dit net als de body van een methode en kan je hierin doen wat je in een methode kan doen.
 - We kunnen nu van buitenaf toch de waarde van `benzine` onrechtstreeks uitlezen via de property en het `get` -gedeelte: `Console.WriteLine(auto.Benzine);`
- `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 instantievariabele aan te passen. De waarde die we van buitenuit krijgen (eigenlijk is dit een parameter van een methode) zal **altijd** in een lokale variabele `value` worden bewaard. Deze zal van het type van de property zijn. In dit geval dus `double` , want het type bij `Benzine` is `double` . Vervolgens kunnen we `value` toewijzen aan de interne variabele indien gewenst: `benzine=value` .

 Let goed op dat je in je setter schrijft `benzine = value` en niet `Benzine = value` . Dat eerste past de verborgen instantievariabele aan. Dat tweede roept de setter opnieuw op. En opnieuw. En opnieuw. Probeer gerust eens een breakpoint te plaatsen voor de toekenning en dan de debugger te starten als je niet ziet waarom dit een probleem is.

 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 `Benzine` heeft nog steeds het probleem dat we negatieve waarden kunnen toewijzen (via de `set`) die dan vervolgens zal toegewezen worden aan `benzine` .

We kunnen in de `set` code extra controles inbouwen. Als volgt:

```

public double Benzine
{
    get
    {
        return benzine;
    }

    set
    {
        if(value >= 0 and value <= 50) {
            benzine = value;
        }
    }
}

```

Deze code zal het benzinepeil enkel aanpassen als het geldig is en anders stilletjes niets doen. Wat je vaak tegenkomt is `throw new ArgumentException($"{value} is geen geldig benzinepeil")`. Dit doet je programma crashen, maar legt ook uit waarom. We kunnen de code binnen `set` (en `get`) zo complex maken als we zelf willen.

⚠ Je kan dus extra controles toevoegen, maar deze hebben alleen zin als je de variabele **via de property** aanpast. Als je in een methode van de klasse `auto benzine` met kleine "b" aanpast en niet voorzichtig bent, kan je nog steeds een negatief peil instellen. Daarom wordt aangeraden **ook binnen de klasse** gebruik te maken van de property, dus zo veel mogelijk `Benzine` in plaats van `benzine` te gebruiken.

Property variaties

We zijn niet verplicht om zowel de `get` en de `set` code van een property te schrijven.

Write-only property

```

public double Benzine
{
    set
    {
        if(value >= 0) {
            benzine = value;
        }
    }
}

```

We kunnen dus enkel `benzine` een waarde geven, maar niet van buitenuit uitlezen.

Read-only property

```
public double Benzine
{
    get
    {
        return benzine;
    }
}
```

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 definiëren we een read-only property met een private setter:

```
public double Benzine
{
    get
    {
        return benzine;
    }
    private set
    {
        if(value >= 0) {
            benzine = value;
        }
    }
}
```

Van buitenuit zal enkel code werken die de `get` -van deze property aanroept:

`Console.WriteLine(auto.Benzine);`. Code die de `set` van buitenuit nodig heeft zal een fout geven zoals: `auto.Benzine=40`.

Read-only Get-omvormers

Veel properties bieden toegang tot een achterliggende variabele van hetzelfde type, maar dat is niet verplicht. Je kan ook iets berekenen en dat teruggeven via een getter.

Als we verder gaan met de klasse `Auto` :

```

public class Auto
{
    private int kilometers;
    private double benzine;
    // stelt het aantal blokjes benzine voor op je display
    // bij 50l heb je 5 blokjes
    // bij tussen 40 en 50l heb je 4 blokjes
    // ...
    // bij minder dan 10l heb je 0 blokjes
    public int Blokjes {
        get {
            return Math.Floor(this.benzine / 10);
        }
    }
}

```

Je hebt iets gelijkaardigs gezien bij `DateTime`. Daar kan je allerlei stukjes informatie uit opvragen, maar eigenlijk wordt er maar één ding bijgehouden: het aantal "ticks". Dat zijn fracties van seconden sinds een bepaalde startdatum. Als je het uur of het aantal minuten of de maand of de weekdag of nog iets anders opvraagt via een `DateTime`, wordt deze ter plekke berekend uit het aantal ticks. Zo moet maar één stukje informatie worden bijgehouden, wat leidt tot minder fouten.

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. Ze zorgen dat je al snel properties hebt, maar staan je niet toe complexe berekeningen te doen of waarden te controleren zoals full properties dat wel doen.

Voor `Auto` kan je bijvoorbeeld schrijven:

```

public class Auto
{
    public double Benzine
    { get; set; }
}

```

Dit maakt achter de schermen een privé-attribuut aan zoals `benzine` met kleine "b", maar met een een verborgen naam. We kunnen dus niet rechtstreeks aan dat attribuut.



Wij zullen geen gebruik maken van autoproperties, omdat het verschil met publieke attributen pas in meer geavanceerde scenario's zichtbaar wordt. We geven ze hier mee zodat je ze kan herkennen als je ze tegenkomt.

Oefeningen

Richtlijnen

Structuur oefeningen

Vanaf hier veronderstellen we dat je in één groot project werkt dat één klasse `Program` heeft. Deze klasse heeft een `Main` methode die een keuzemenu opstart. Oefeningen rond eenzelfde topic worden (statische) methodes van één klasse met een methode `ToonSubmenu`, die je een menu toont van alle oefeningen over dat topic en die je toestaat een oefening naar keuze te testen. Dit wordt uitgelegd in de eerste oefening.

Oefening: H10-voorbereiding

Leerdoelen

- een ordelijke menustructuur voor je code voorzien

Functionele analyse

We willen dat we alle oefeningen die we in dit vak maken op een ordelijke manier kunnen opstarten. We doen dit door een keuzemenu met twee niveaus te voorzien: de gebruiker kan elke reeds geschreven oefening uitvoeren door eerst het algemene onderwerp aan te geven en vervolgens de specifieke oefening.

Technische analyse

- Laat in je `Main` methode een lijst van alle topics zien waarover oefeningen gemaakt zijn. In het begin is dit enkel `DateTime`. De gebruiker kan een topic aanduiden door een nummer in te geven, want voor elk topic staat ook een nummer.
- Gebruik een switch op de gebruikersinput om te bepalen van welk topic de `ToonSubmenu` methode moet worden opgeroepen. Deze methode heeft return type `void` en geen parameters.
- Voorzie een eerste klasse, `DateTimeOefeningen`, met deze methode `ToonSubmenu`. Totdat je oefeningen hebt om te demonstreren, toont `ToonSubmenu` gewoonweg de tekst "Er zijn nog geen oefeningen over dit topic".
- Indien er wel oefeningen zijn (deze oefening moet je dus updaten naarmate je vordert), wordt elke reeds geprogrammeerde oefening genummerd en getoond en kan de gebruiker kiezen om deze uit te voeren.
- Nadat een oefening getest is, kan je opnieuw een topic en een oefening kiezen. Het programma eindigt nooit.

⚠ Dit is maar een voorbeeld! De getoonde topics en oefeningen gaan afhangen van wat je al gedaan hebt.

```
Welkom bij de demo Objectgeoriënteerd Programmeren!
Topic van de uit te voeren oefening?
1. DateTime
2. Properties en access modifiers
> 1
Uit te voeren oefening?
1. H10-dag-van-de-week
2. H10-ticks-sinds-2000
3. H10-schrikkelteller
> 2
Sinds 1 januari 2000 zijn er (...) ticks voorbijgegaan.
Topic van de uit te voeren oefening?
(...)
```

Oefening: H10-dag-van-de-week

Leerdoelen

- aanmaken van `DateTime` objecten
- formatteren van `DateTime` objecten

Functionele analyse

We willen voor een willekeurige datum kunnen bepalen welke dag van de week het is.

Technische analyse

- je moet eerst de dag, maand en jaar opvragen en een `DateTime` aanmaken
- daarna moet je laten zien over welke dag van de week het gaat
 - gebruik hiervoor formattering van een `DateTime`
 - laat ook de datum zelf zien in een formaat dat leesbaar is voor de gebruiker
 - als je computer niet volledig ingesteld is op Belgisch Nederlands, kan het resultaat er wat anders uitzien.
- maak deze methode toegankelijk via `ToonSubmenu` van de klasse `DateTimeOefeningen`
- noem de methode waarin je dit schrijft `WeekdagProgramma`

Voorbeeldinteractie

```
Welke dag?  
> 14  
Welke maand?  
> 2  
Welk jaar?  
> 2020  
14 februari 2020 is een vrijdag.
```

Oefening: H10-ticks-sinds-2000

Leerdoelen

- aanmaken van `DateTime` objecten

Functionele analyse

We willen weten hoe veel fracties van een seconde al verlopen zijn sinds het begin van de jaren 2000.

Technische analyse

- .NET stelt deze fracties (1 / 10000 milliseconden) voor als "ticks"
- We willen weten hoe veel ticks er voorbijgegaan zijn sinds het absolute begin van het jaar 2000
- maak deze methode toegankelijk via `ToonSubmenu` van de klasse `DateTimeOefeningen`
- Noem de methode waarin je dit schrijft `Ticks2000Programma`

Voorbeeldinteractie

```
Sinds 1 januari 2000 zijn er (hier wordt het aantal getoond) ticks voorbijgegaan.
```

Oefening: H10-schrikkelteller

Leerdoelen

- gebruik van een statische methode

Functionele analyse

We willen bepalen hoe veel schrikkeljaren er zijn tussen 1799 en 2021.

Technische analyse

- implementeer zelf geen logica voor schrikkeljaren, maar laat dit over aan de klassen `DateTime`
- maak gebruik van een statische methode van deze klasse
- maak deze methode toegankelijk via `ToonSubmenu` van de klasse `DateTimeOefeningen`
- noem je methode `SchrikkeljaarProgramma`

Voorbeeldinteractie

Er zijn (hier wordt het aantal getoond) schrikkeljaren tussen 1799 en 2021.

Oefening: H10-simpele-timing

Leerdoelen

- eenvoudig code leren timen
- gebruiken van `DateTime`
- herhaling arrays

Functionele analyse

We zijn benieuwd hoe lang het duurt een array van 1 miljoen `int`s te maken en op te vullen met de waarden 1,2,...

Technische analyse

- Bepaal het tijdstip voor en na aanmaken van de array.
- Vul de array in met een `for`-lus.
- maak deze methode toegankelijk via `ToonSubmenu` van de klasse `DateTimeOefeningen`
- Noem de methode waarin je dit schrijft `ArrayTimerProgramma`

Voorbeeldinteractie

Het duurt (hier wordt het aantal getoond) milliseconden om een array van een miljoen element

Oefening: H10-verjaardag-v2

Leerdoelen

- leren werken met objecten
- gebruik maken van properties en methodes

Functionele analyse

We zullen het programma uit om het aantal dagen tot een verjaardag te bepalen (uit de theorie) aanpassen zodat het aantal dagen tot de volgende verjaardag wordt getoond. Dit betekent dat er nooit 0 dagen tot een verjaardag zijn, maar in extreme gevallen duizenden dagen kunnen over gaan (bijvoorbeeld: van 29 februari 1996 tot 29 februari 2004).

Technische analyse

Je moet één geval toevoegen, namelijk het geval waarin het oude programma 0 dagen tot de volgende verjaardag zou geven.

Noem je methode `VerjaardagProgramma`. Maak deze methode toegankelijk via `ToonSubMenu` van de klasse `DateTimeOefeningen`.

Oefening: H10-Getallencombinatie

Leerdoelen

- werken met klassen en objecten
- instantieattributen
- instantiemethoden

Functionele analyse

Dit programma geeft op basis van de input van twee getallen de som van beide getallen, het verschil, het product en de deling. In het laatste geval en indien er een deling door nul zou worden uitgevoerd, wordt dit woordelijk weergegeven.

Technische analyse

Voorzie voor volgende oefening eerst een nieuwe submenuklasse met als naam `EigenObjectOefeningen`.

Maak een eigen klasse `GetallenCombinatie` in een eigen file, `GetallenCombinatie.cs`. Deze klasse bevat 2 getallen (type `int`). Er zijn 4 methoden, die allemaal een `double` teruggeven:

- `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 weer. Print "Fout" naar de console indien je zou moeten delen door 0 en voer dan de deling uit. Wat er dan gebeurt, is niet belangrijk.

Gebruik `public` attributen `Getal1` en `Getal2`. Plaats onderstaande code in een publieke statische methode van `EigenObjectOefeningen` met naam `DemonstreerOperaties` met return type `void`:

```
GetallenCombinatie paar1 = new GetallenCombinatie();
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());
```

Zorg dat je `DemonstreerOperaties` kan oproepen via het submenu van `EigenObjectOefeningen`.

Voorbeeldinteractie(s)

```
Paar: 12, 34
Som = 46
Verschil = -22
Product = 408
Quotient = 0,352941176470588
```

Oefening: H10-StudentKlasse

⚠ Deze oefening veronderstelt dat je de theoriefilmpjes hebt gevolgd en dat je daar de klasse `Student` al hebt aangemaakt in een `SchoolAdmin` project.

Leerdoelen

- werken met klassen en objecten
- opstart van het project

Functionele analyse

Dit programma vraagt om de naam en leeftijd van een student. Vervolgens worden de punten voor 3 vakken gevraagd, waarna het gemiddelde wordt teruggegeven.

Technische analyse

Breid je klasse `Student` uit met een tweede array `CursusResultaten`. Voorzie ook een methode `Kwoteer` om een cijfer aan een cursus met een bepaalde index toe te kennen. Signaleer ongeldige waarden met een `Console.WriteLine("Ongeldig cijfer!")`. Je kan ook nooit een cijfer boven 20 behalen.

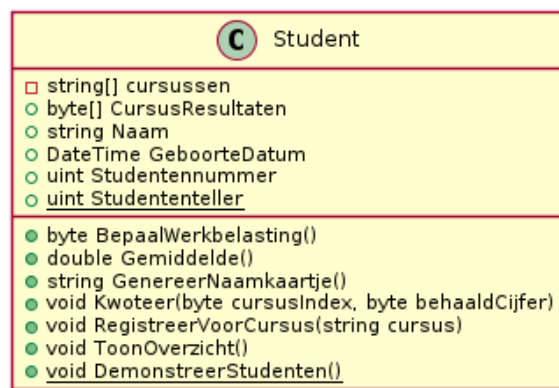
Voeg aan de klasse een methode `Gemiddelde()` toe. Deze berekent het gemiddelde van de niet-`null` cursussen als `double`.

Maak het return type van `BepaalWerkbelasting` ook een `byte`.

Voeg ook een methode `ToonOverzicht` toe, die de persoonsgegevens en behaalde cijfers voor elke student toont. Kijk naar de voorbeeldinteractie voor de juiste voorstellingswijze.

Test je programma door een statische methode (in de klasse `Student`), `DemonstreerStudenten` te voorzien, die twee studenten aanmaakt via variabelen `student1`, `student2`. Elke student is ingeschreven voor minstens drie vakken die je zelf kiest en krijgt een geldig cijfer (naar keuze) voor elk vak, een naam en een geldige leeftijd. Vervolgens wordt van elke student de `ToonOverzicht`-methode opgeroepen. In je `Main`-methode voorzie je een (niet-genest) keuzemenu dat vraagt wat je wil doen en op dit moment is de enige optie `DemonstreerStudenten` uitvoeren.

Een overzicht van de klasse na al deze wijzigingen:



Voorbeeldcode om de eerste student aan te maken:

```
Student student1= new Student();
student1.Geboortedatum = new DateTime(2001,1,3);
student1.Naam = "Said Aziz";
student1.RegistreerVoorCursus("Communicatie");
student1.CursusResultaten[0] = 12;
student1.RegistreerVoorCursus("Programmeren");
student1.CursusResultaten[1] = 15;
student1.RegistreerVoorCursus("Webtechnologie");
student1.CursusResultaten[2] = 13;
student1.ToonOverzicht();
```

Voorbeeldinteractie(s)

Wat wil je doen?

1. DemonstreerStudenten uitvoeren

> 1

Said Aziz, 20 jaar

Cijferrapport:

Communicatie:	12
Programmeren:	15
Webtechnologie:	13
Gemiddelde:	13.3

Mieke Vermeulen, 21 jaar

Cijferrapport:

Communicatie:	13
Programmeren:	16
Databanken:	14
Gemiddelde:	14.3

⚠ Commit je aanpassingen na deze oefening!

Oefening: H10-Cursus

⚠ Deze klasse hoort bij het SchoolAdmin project.

Leerdoelen

- werken met klassen en objecten
- opstart van het project
- arrays van objecten maken

Functionele analyse

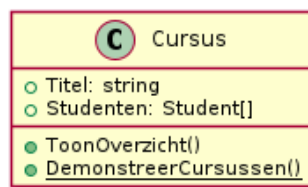
We zullen studenten groeperen in cursussen. Bij elke cursus horen op dit moment exact twee studenten.

Technische analyse

Werk verder in het SchoolAdmin project. Maak in dit nieuw project een nieuwe klasse `Cursus` in een file `Cursus.cs`. Deze klasse heeft twee attributen: `Studenten` en `Titel`. `Studenten` is een array van `Student`-objecten. De initiële waarde voor dit attribuut is een array met een capaciteit van 2 studenten. `Titel` is gewoonweg een `string`. `Cursus` heeft ook een methode `ToonOverzicht` die de titel van de cursus toont, gevolgd door de namen van alle studenten die de cursus volgen.

Test je programma door een statische methode (in de klasse `Cursus`), `DemonstreerCursussen` te voorzien, die vier cursussen ("Communicatie", "Databanken", "Programmeren" en "Webtechnologie") aanmaakt via variabelen `communicatie`, `programmeren`, `webtechnologie` en `databanken`. Maak ook twee studenten aan (dezelfde als in `DemonstreerStudenten`) en maak hen lid van de cursussen waarvoor ze een cijfer hebben (zie voorbeeldinteractie van de vorige oefening). Toon tenslotte voor elke cursus het overzicht via `ToonOverzicht`. De methode `DemonstreerCursussen` kan ook opgeroepen worden via het keuzemenu in `Main`.

Je klasse `Cursus` ziet er uiteindelijk zo uit:



UML-klassendiagram voor Cursus

Voorbeeldinteractie(s)

Wat wil je doen?

1. DemonstreerStudenten uitvoeren

2. DemonstreerCursussen uitvoeren

> 2

Communicatie

Said Aziz

Mieke Vermeulen

Databanken

Mieke Vermeulen

Programmeren

Said Aziz

Mieke Vermeulen

Webtechnologie

Said Aziz



Commit je aanpassingen!

Oefening: H10-CursusResultaat

Leerdoelen

- wegwerken gesynchroniseerde arrays
- encapsulatie
- access modifiers


Functionele analyse

De eerdere oefening H10-StudentKlasse gebruikte gesynchroniseerde arrays. We willen deze wegwerken.

Technische analyse

Voorzie een klasse CursusResultaat met twee velden: `Naam` en `Resultaat`. Het eerste stelt de naam van de cursus voor, het tweede het behaalde resultaat.

Vervang vervolgens de arrays `cursussen` en `CursusResultaten` door één `private` array van objecten van deze nieuwe klasse met naam `cursusResultaten`. Vervang `RegistreerVoorCursus` door `RegistreerCursusResultaat` om dit mogelijk te maken (met een parameter voor de naam en een parameter voor het cijfer). `DemonstreerStudenten` moet identiek dezelfde uitvoer blijven produceren als tevoren.

-  Deze oefening vraagt om veel aanpassingen, maar ze zijn niet zo groot. Hou vooral je hoofd erbij en denk goed na over hoe je elk fout gemarkeerd stukje code kan herschrijven wanneer je de oude arrays hebt verwijderd.

-  Commit je aanpassingen!

Oefening: H10-Figuren

Leerdoelen

- werken met klassen en objecten
- gebruik maken van properties om geldige waarden af te dwingen


Functionele analyse

Dit programma maakt enkele rechthoeken en driehoeken met gegeven afmetingen (in meter) aan, berekent hun oppervlakte en toont deze info aan de gebruiker. De rechthoeken en driehoeken die worden aangemaakt, zijn al gecodeerd in het programma. De gebruiker hoeft dus niets anders te doen dan het programma te starten.

Technische analyse

Er is een klasse `Rechthoek` met **full properties** `Breedte` en `Hoogte` en een klasse `Driehoek` met `Basis` en `Hoogte`. Je programma maakt de figuren die hierboven beschreven worden aan met beginwaarde `1.0` voor elke afmeting en stelt daarna hun afmetingen in via de setters voor deze properties. De oppervlakte wordt bepaald in een read-only property (dus met alleen een getter en geen setter). Deze heet `Oppervlakte` en is van het type `double`.

Indien om het even welk van deze properties wordt ingesteld op `0` of minder, signaleer je dit via de code `Console.WriteLine($"Het is verboden een (afmeting) van (waarde) in te stellen!")` (zie voorbeeldcode).

-  De wiskundige formule voor de oppervlakte van een driehoek is $\text{basis} * \text{hoogte} / 2$.

Schrijf de voorbeelden uit in een `static` methode `DemonstreerFiguren` van de klasse `EigenObjectOefeningen`.

Voorbeeldinteractie(s)

(Er worden twee rechthoeken en twee driehoeken aangemaakt. De afmetingen van de eerste rechthoek worden eerst op `-1` en `0` ingesteld. Daarna krijgen ze de waarden die je ziet in het bericht hieronder. Formateer ook met 1 cijfer na de komma.)

Het is verboden een breedte van `-1` in te stellen!

Het is verboden een breedte van `0` in te stellen!

Een rechthoek met een breedte van `2,2m` en een hoogte van `1,5m` heeft een oppervlakte van `3,3m`

Een rechthoek met een breedte van `3m` en een hoogte van `1m` heeft een oppervlakte van `3m2`.

Een driehoek met een basis van `3m` en een hoogte van `1m` heeft een oppervlakte van `1,5m2`.

Een driehoek met een basis van `2m` en een hoogte van `2m` heeft een oppervlakte van `2m2`.