

# **Programming with Objects (first draft)**

Teachers StaSemSoft Fontys ICT

2023-09-20

# Table of contents

<b>Preface</b>	<b>3</b>
<b>1 Introduction</b>	<b>4</b>
<b>2 Toolbox Verdiepend: Objects en Classes</b>	<b>5</b>
2.1 Algemene info . . . . .	5
2.2 Introductie met OOP . . . . .	6
2.3 Classes, objecten en constructors . . . . .	14
2.3.1 Dictaat C# classes . . . . .	14
2.3.2 1. Classes . . . . .	14
2.4 Properties en encapsulation . . . . .	20
2.5 Methods . . . . .	21
2.6 Casting en override ToString() . . . . .	21
2.7 Enum . . . . .	21
2.8 Gecombineerd . . . . .	21
2.9 Challenges . . . . .	21
2.10 Extra . . . . .	21
2.10.1 GUI vs Domain . . . . .	21
2.10.2 Git . . . . .	22
2.10.3 File Handling en Exception Handling . . . . .	22
2.10.4 Database . . . . .	22
2.10.5 Testing . . . . .	22
2.11 Best practices Voor de wanna-be Software Engineer . . . . .	22
2.12 Verder Verdiepend materiaal . . . . .	22
<b>3 Summary</b>	<b>23</b>
<b>References</b>	<b>24</b>

# Preface

This is a Quarto book.

To learn more about Quarto books visit <https://quarto.org/docs/books>.

# **1 Introduction**

This is a book created from markdown and executable code.

See Knuth (1984) for additional discussion of literate programming.

# 2 Toolbox Verdiepend: Objects en Classes

## 2.1 Algemene info

- Wat kan ik als ik de module ‘objects’ voldoende doorlopen heb? ### Wat kan ik als ik ‘objects’ beheers?

### 2.1.0.1 Toelichting Verdiepend niveau

Je past de volgende programmeer-concepten toe en hebt voorbeelden waaruit blijkt dat je de volgende concepten beheert:

Voor het verdiepend niveau geldt alles wat bij het oriënterende niveau gold met de volgende extra eisen:

Je past alle genoemde programmeerconcepten toe. De focus hierbij is op leesbare (b.v. naamgeving, indentation) en onderhoudbare software programma’s (dus alleen een programma dat werkt is niet goed genoeg).

Je maakt een ontwerp (CRC, class diagram) waarin de belangrijkste functionaliteit terug te vinden is.

Je vraagt feedback van een docent met software engineeringskennis en laat zien dat je deze feedback verwerkt hebt.

Concepten/Onderwerpen: (GEEN AFVINK-LIJST!)

1. objects / classes
2. constructors
3. encapsulation:
  - private fields
  - get/set- method en/of property
4. method/constructor overloading
5. *override ToString()*.
6. CRC / class diagram
7. Methods met (eigen) Classes als parameter of return value.
8. Classes in samenhang: *relations* (tussen classes)

- Multiplicity.

## 9. Scheiding GUI en Domain.

Ook heb je de volgende vaardigheden laten zien en/of er aan gewerkt:

- V1. Leesbaarheid / Onderhoudbaarheid
  - Coding Guidelines gevuld?
  - commentaar in code.  
Soms handig, maar ga geen onnodig commentaar toevoegen! Als een variable-, method- of class-name uitlegt nodig heeft, probeer dan eerst of je een betere naam kunt verzinnen.
- V2. Algoritmiek
- V3. Feedback gevraagd van docent, genoteerd, verwerkt.
- V4. Professioneel gecommuniceerd. Met name ben ik op tijd, meld ik me af bij docent als ik er een keer niet of te laat ben.

Een “soort” checklist (niet-compleet) die kan helpen: + Om concepten onder de knie te krijgen en ermee te oefenen heb je een aantal ‘trainingen’ gedaan. + In de ‘wedstrijd’ laat je zien wat je kunt. In een wedstrijd-app, een voor de ‘wedstrijd’ gemaakte app(licatie), laat je zien dat je weet hoe meerdere concepten binnen een onderhoudbare app worden gebruikt. De onderstaande regels gaan over je wedstrijd-apps! + Om jezelf en een docent te overtuigen dat je classes kunt maken heb je minstens 10 classes gemaakt, waarvan minstens 5 binnen 1 wedstrijd-app. + Deze classes hebben methods, fields, properties, constructors + In minstens 5 classes heb je een zinvolle ‘override ToString()’ toegevoegd.

---

## 2.2 Introductie met OOP

- Reference: Hoe programmeer je met objecten? (hero vs monsters) #### Hoe programmeer je met objecten?

### 2.2.0.1 Aan de hand van Hero tegen Monsters

Hoe groter software wordt, hoe tijdrovender het testen en onderhouden. Daarom wordt in de softwarewereld gezocht naar manieren om programma’s onderhoudbaar te maken. Een van de meer succesvolle manieren is het werken met *objecten*. Een team dat een Computer Game maakt over een *hero* (held) die tegen *monsters* vecht zal liefst op één plek willen programmeren

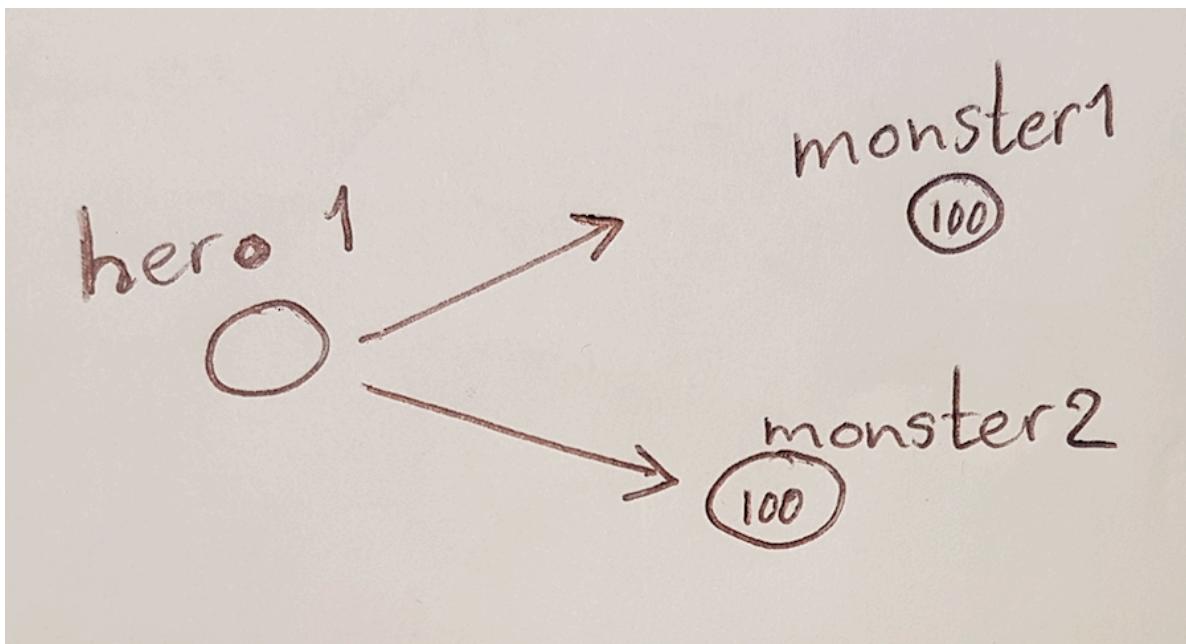


Figure 2.1: fig:MonsterGame

wat de eigenschappen van de hero zijn en ook wat een monster is en wat je aan het monster kunt vragen.

Als er 2 monsters zijn worden er 2 *objecten* gemaakt, als er 8 monsters zijn worden er 8 *objecten* gemaakt. Elk van deze objecten represeneert 1 monster. De eigenschappen en het gedrag van een monster wordt geprogrammeerd in een stuk van het programma dat we een **Class** (klasse) noemen en dat (in dit voorbeeld) de naam **Monster** krijgt. Voorbeeld in C# (hoe je dit in Visual Studio kunt doen komt een stukje verder):

```
class Monster {
    ...
}
```

waarbij op de plaats van de puntjes de code voor deze **class** komt. Voorbeeld in Java:

```
class Monster {
    ...
}
```

Voorbeeld in Swift:

```
class Monster {  
    ...  
}
```

Inderdaad, deze voorbeelden zijn hetzelfde. Je zult merken dat er echt wel verschillen zijn hoe je in de ene of de andere taal een `class` noteert, maar in welke taal je ook zit: *welke classes* je aanmaakt blijft hetzelfde!

In veel programmeertalen is afgesproken dat de naam van een `class` met een hoofdletter begint.

Software Engineers bedenken in het begin van een project welke `objecten` er nodig zijn en daaruit volgt welke `classes` er geprogrammeerd gaan worden. Dit kun je grotendeels bedenken zonder te weten in welke taal de software gebouwd gaat worden. Je kunt dat enigszins vergelijken met het bouwen van een huis: Waar de muren, ramen en deuren komen (de structuur) kun je tot zekere hoogte bedenken en tekenen zonder te weten of het huis met bakstenen, van beton of van hout gebouwd gaat worden. Een `object` kan bepaalde eigenschappen hebben. Zo zal elk Monster in eerste instantie helemaal gezond zijn. Als de *hero* hem aanvalt zal het *monster* moe worden of gewond raken en uiteindelijk wellicht bezwijken.

#### 2.2.0.1.1 Hoe maak ik een class aan in Visual Studio?

Klik met rechtermuisknop op het project en kies `Add Item`, kies daarna een `class`. Onderin het scherm kun je de gewenste `class name` angeven (`file name` is `class name` met extensie `.cs`) en dan op de *OK*-knop.

In veel programmeertalen is het gebruikelijk om elke `class` in een eigen file te programmeren.

#### 2.2.0.2 Gezondheid

In dit spel kunnen we dat realiseren door het monster `hitPoints` te geven: Voor een pas *aangemaakt* monster staat dit op 100, bij verwondingen wordt dit gehele getal steeds kleiner, bij 0 valt het monster verslagen neer. Een waarde als `hitPoints` die elk `object` van een bepaald `type` met zich meedraagt noemen we een `Field`. We maken hiertoe in `Monster` een `field levenspunten` aan.

```
class Monster {  
    int hitPoints = 100;  
}
```

Hiermee is bepaald dat **elk** monster `hitPoints` heeft. De waarde van dat getal kan per *monster object* verschillen: *Monster 1* kan nog op 100 staan terwijl *monster 2* misschien nog maar 13 over heeft.

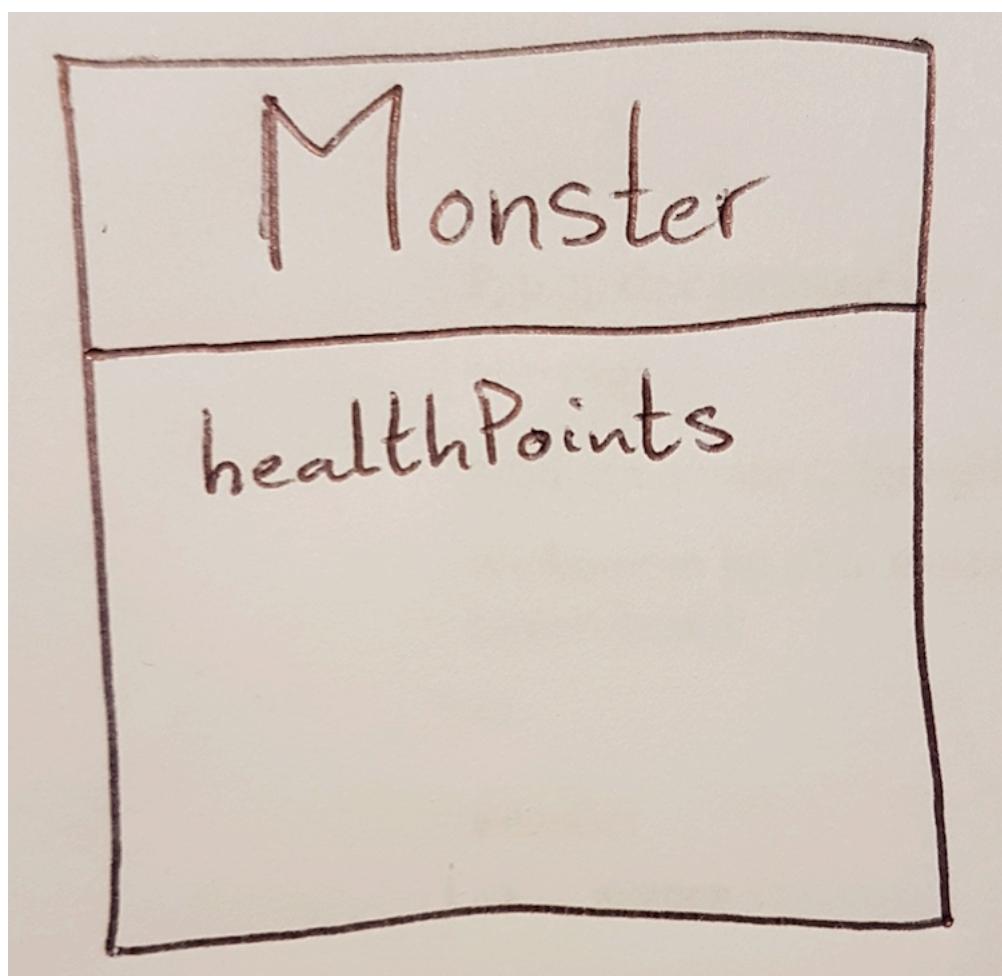
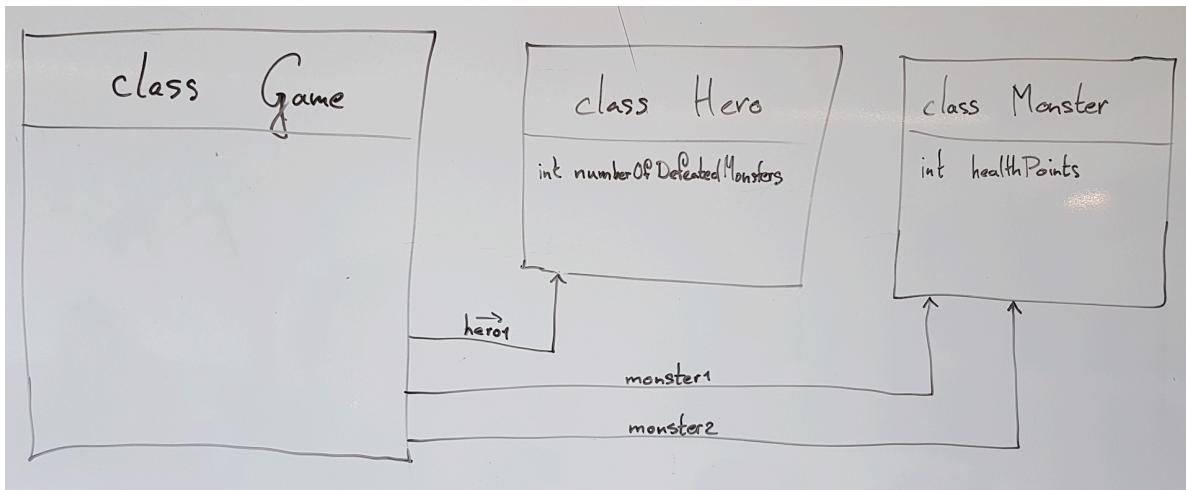
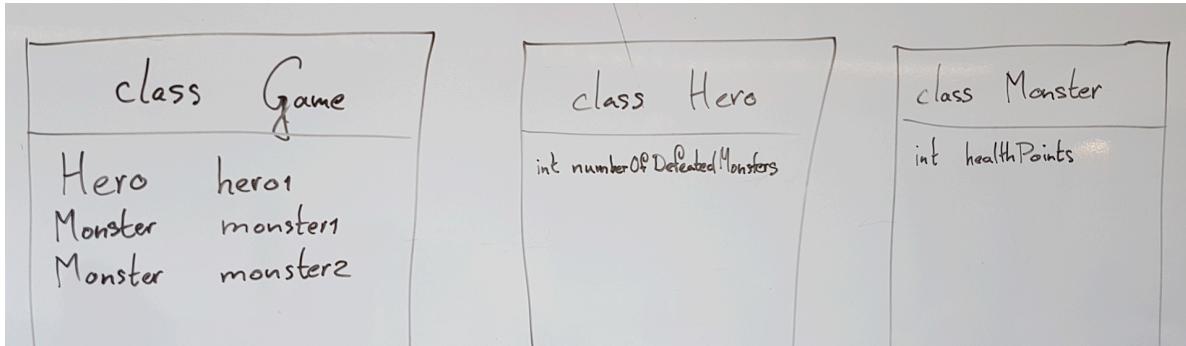


Figure 2.2: class Monster

De code die in een **class** staat wordt gedeeld met alle **objecten** van die **class** (meestal zeggen we: alle **objecten** van dat **type**, want een **class** is een manier om een **type** te definiëren). Om een **object** van **class Monster** aan te maken: C# of Java:

```
new Monster()
```

(we zeggen dan ook wel dat er gebruik gemaakt wordt van de **new operator**) Hiermee wordt ergens in het geheugen een **object** van **type Monster** aangemaakt, we hebben echter **geen** manier om naar dat **object** te *verwijzen* (*refereren*). Vergelijk het met een ballon met gas: zolang je het touwtje hebt (de referentie naar de balon) kun je bij de ballon, maar als je het touwtje loslaat kun je niet meer bij de ballon komen. Zo'n *referentie* kunnen we opslaan in een **Field** (ook wel een variabele genoemd) en dat **Field** moet ergens in een **class** zitten: We maken hiervoor een *Game-object* aan dat de referenties naar alle *heroes* en *monsters* bevat. De code van het *Game object* komt in de **class Game** te staan.



In **class Hero** is een **Field**

```
int numberDefeatedMonsters = 0;
```

aangemaakt. De held wil namelijk graag dat de hele wereld weet hoeveel monsters er door hem/haar verslagen zijn. De *Class Game* heeft referenties naar 1 *hero* en 2 monsters (*monster 1* en *monster 2*) en aan het ervoor vermelde *type* (dat zijn de *class* namen) kun je zien dat de *hero* zich gedraagt zoals in *class* ‘Hero’ geprogrammeerd is, terwijl de beide monsters zich gedragen volgens de code in *class* Monster.

```
public Game()
{
    Hero hero = new Hero();
    Monster monster1 = new Monster();
    Monster monster2 = new Monster();
}
```

### 2.2.0.3 Attack

Onze *hero* staat te popelen om een monster te gaan aanvallen. Hiervoor gaan we *gedrag* in de *class Hero* programmeren:

Dit wil zeggen dat je op een *object* van *type Hero* een *method* kunt aanroepen die *Attack* heet. Verder vertel je **welk** monster aangevallen wordt en hoeveel *schade* (*damage*) hierbij toegebracht wordt aan het monster (dus hoeveel er van de *hitPoints* punten van het monster af gaan). Als de method *Attack* op een *Held object* aangeroepen wordt wordt de code van die methode uitgevoerd.

De held roept dan van het tussen haakjes genoemde monster de method *LooseHealth* aan. Nu gaan we coderen hoe die method er uit kan zien: daartoe programmeren we de method *Attack* in de *class Hero*

```
void Attack(Monster monster, int damage)
{
    monster.LooseHealth(damage);
}
```

ofwel: als op een *object* van *type Hero* (want in die *class* staat deze code) de method *Attack* wordt aangeroepen (met als parameters tussen haakjes aangegeven **welk** monster en hoeveel *damage*) roept die de method *LooseHealth* aan van het aangegeven monster. De binnengekomen info over hoeveelheid *damage* wordt doorgegeven. In de method worden 2 zogenaamde parameters gebruikt, namelijk *monster* van het *type Monster* en *damage* van het *type int* (tussen haakjes te vinden na de methode naam). Het woord *void* wil zeggen dat er geen waarde wordt teruggegeven door de methode, Er kan ook in plaats van *void* een zogenaamd *return type* staan dat aangeeft wat voor soort waarde er terug gegeven wordt.

In *class Monster* moet vervolgens de method *LooseHealth* gecodeerd worden:

```

void LooseHealth(int damage)
{
    this.hitPoints = this.hitPoints - damage;
}

```

Uitleg: - Wederom begint het met *void* omdat de methode niks teruggeeft. - Dan de methodenaam *LooseHealth*. - Tussen de haakjes de ene parameter, genaamd *damage* en van type *int*. - Tussen de accolades of *curly brackets* ('{' en '}') staat een assignment: - Een assignment is te herkennen aan het *=*-teken (spreek uit als **wordt**). - Rechts van de *=* staat een *expressie*, zeg maar een berekening, die uitgerekend (geëvalueerd) wordt. In dit geval: *this.hitPoints - damage*. - Het woord *this* geeft aan dat er iets gedaan wordt met het *object* waar we nu 'in' zitten: het specifieke monster dus dat werd aangevallen en dat dus als parameter aan method *Attack* werd meegegeven. In methode *Attack* zie je dat van *DAT* specifieke monster de methode *LooseHealth* wordt aangeroepen. - *Evaluatie* (berekening) van *this.hitPoints* geeft het *hitPoints*-getal van dat monster. - De '*- damage*' zorgt dat de meegegeven waarde van de parameter hier vanaf getrokken wordt. - De waarde van *this.hitPoints* (links van de *=*) wordt de uitkomst van de berekening.

#### 2.2.0.3.1 Constructie van een object

Net zoals we bij een methode aanvullende informatie mee kunnen geven in de vorm van **parameters** kunnen we dat bij het aanmaken van een nieuw **object** ook. Hiertoe gebruiken we een **constructor**: Een **constructor** ziet er ongeveer uit als een methode:

```

Monster(int initialHealth)
{
    this.hitPoints = initialHealth;
}

```

Een **constructor** herken je als volgt: - De constructor lijkt heel erg op een normale methode, maar... - Er wordt geen **return-type** (of **void**) vermeld. - De naam (*Monster* in dit geval) is gelijk aan de naam van de **class**.

#### 2.2.0.3.2 Constructor in Visual Studio

Je kunt natuurlijk de code hierboven zelf intypen (tussen de accolades van de **class**) maar als je op die plek intypt *ctor* en dan 2x op *tab* drukt doet Visual Studio een deel van het werk voor je. Als we nu **new Monster(125)** aanroepen vanuit code wordt er een object van type *Monster* geconstrueerd en daarvoor staat na constructie de *hitPoints*-waarde op het meegegeven getal, 125 dus in dit geval. Bij een **constructor** kunnen (net als bij een *normale* methode) ook meerdere parameters meegegeven worden.

#### 2.2.0.4 Wat hebben we nu?

We hebben nu een basis neergezet voor een spel waarin een *hero* *monsters* kan aanvallen.

##### 2.2.0.4.1 Om het werkend te krijgen

Later wordt nog uitgelegd waarom, maar onthoudt vast dat we elk *Field private* maken. Methods en classes mogen public zijn.

##### 2.2.0.4.2 Code tot nu toe

Voor de volledigheid volgt nu de code van de classes zoals die tot hier beschreven is. Allereerst de class *Game*

```
namespace HereComeTheMonsters
{
    public class Game
    {
        public Game()
        {
            Hero hero = new Hero();
            Monster monster1 = new Monster(125);
            Monster monster2 = new Monster(100);
        }
    }
}
```

dan class *Hero*

```
namespace HereComeTheMonsters
{
    public class Hero
    {

        public Hero()
        {

        }

        public void Attack(Monster monster, int damage)
        {
            monster.LooseHealth(damage);
        }
    }
}
```

```
    }

}

}
```

en tot slot `class Monster`

```
namespace HereComeTheMonsters
{
    public class Monster
    {

        private int hitPoints = 100;

        public Monster(int initialHealth)
        {
            this.hitPoints = initialHealth;
        }

        public void LooseHealth(int damage)
        {
            this.hitPoints = this.hitPoints - damage;
        }
    }
}
```

## 2.3 Classes, objecten en constructors

- Reference: [Classes maken en gebruiken](#)

### 2.3.1 Dictaat C# classes

#### 2.3.2 1. Classes

Een `Class` is een soort *blauwdruk*. Zie het als een tekening van hoe iets er uit moet gaan zien en wat dat “ding” kan gaan doen zodra je het daadwerkelijk gaat maken. Van een boot

maak je eerst een **design**, een **ontwerp**: Wat voor soort hout heb je nodig? Waar liggen de verbindingen? Wat voor een motor komt er in te liggen? Et cetera. Deze tekening, de blauwdruk, kan niet varen. Je kunt er niet op dobberen. Pas wanneer de boot wordt gemaakt kun je er iets mee gaan doen. Je kunt er zelfs meerdere boten van maken! Zo is het met classes ook. Een class is de blauwdruk voor “iets”. Je kunt er pas mee aan de slag op het moment dat je de blauwdruk **instantieert**: een **object** van dat **type** maakt.

Een andere vergelijking is dat van een **lopende band**: Van een lopende band kunnen meerdere dingen (objecten) van een bepaald type afrollen: Elke keer dat ik **new** roep tegen de class komt er een object van dat type van de lopende band afrollen. Hoe de lopende band is ‘geprogrammeerd’ bepaalt wat een object van dat type uiteindelijk kan.

We gaan eerst naar wat voorbeelden kijken die je al kent. Daarna gaan we zelf een **class** maken.

### 2.3.2.1 2.1. Classes gebruiken

Je hebt al eerder de **class Random** gezien, waarschijnlijk zonder te weten dat het een class was. Een voorbeeld:

```
Random dobbelsteen = new Random();
int getal = dobbelsteen.Next();
```

Eerst wordt een variabele aangemaakt van het type Random met de naam ‘dobbelsteen’. (ja, een **class** is een **type**, hoewel in C# niet alle types ook een class zijn). De **Random** is hier de class, de lopende band: door er **new** tegen te roepen (volgende regel) wordt een nieuw object van dit type aangemaakt en de variabele ‘dobbelsteen’ verwijst naar dit object.

We zeggen dat dobbelsteen een **instance** (instantie) of **object** is van de Random class. Van daar de term object-georiënteerd: **Object Oriented Programming**, kortweg O.O.P.).

In de derde regel zie je dat de dobbelsteen gebruikt wordt om een willekeurig getal op te vragen. Daar gebruik je dus functionaliteit die beschreven staat in de Random class. We zeggen dan dat je de **method** ‘Next’ aanroeft (**calls**) op het object ‘dobbelsteen’.

Tip: Je krijgt een **NullPointerException** wanneer je een instantie van een **class** gebruikt zonder dat deze is aangemaakt met **new**.

Je **form** is ook een class. In elke class kun je **fields** (variabelen, maar dan buiten een method) en **methods** programmeren. Bekijk het volgende voorbeeldje (laat **partial** en de toevoeging : **Form** even voor wat het is).

```
public partial class FormDemo : Form
{
```

```

private Random dobbelsteen = new Random();

public int GooiDobbelsteen();
{
    return dobbelsteen.Next(1, 7);
}

private void knopDoeIets_Click(object ...)
{
    int getal = GooiDobbelsteen();
}

```

In het voorbeeld is een **field** aan form toegevoegd van het type Random en als naam dobbelsteen. Deze wordt direct, op dezelfde regel nog, geïnitialiseerd. In de rest van het programma kun je de dobbelsteen veilig gebruiken.

Er is ook een **method** aan het form toegevoegd. Deze kan, net als de variabele, overal in deze FormDemo class gebruikt worden. Dit wordt ook gedaan bij de click-event **handler** van de knop. Wanneer de gebruiker op de knop drukt zal er een *next* worp met de dobbelsteen worden gegooid. Het willekeurige getal wordt door de GooiDobbelsteen methode teruggegeven (**return value**). De output van deze methode wordt opgevangen in een integer met de naam ‘getal’.

#### 2.3.2.1.1 Voorbeeld bestaande class: **StringBuilder**

Hieronder zie je een ander voorbeeld van het gebruik van een class.

```

StringBuilder welkom = new StringBuilder();
welkom.Append("Welkom ");
welkom.Append("bij programmeren");
MessageBox.Show(welkom.ToString());

```

Hier wordt gebruik gemaakt van de **StringBuilder** class. Probeer het bovenstaande stukje code ook even zelf uit. Op de eerste regel code wordt een variabele van het type **StringBuilder** aangemaakt. De variabele heet **welkom** en wordt meteen geïnitialiseerd. Via de ‘Append’ method kunnen er stukken tekst aan worden toegevoegd. Deze wordt in zijn geheel, door de ‘**ToString()**’ methode aan te roepen, aan de gebruiker laten zien.

Probeer zelf ook wat **fields** en **methodes** toe te voegen aan je form. Kijk eens wat je er allemaal mee kunt doen.

### 2.3.2.2 2.2. Zelf classes maken

Stel je een boot voor met een snelheid, een naam, een gewicht en een aantal bemanningsleden. De boot kan varen en kan het anker uitgooien. Hoe zou je dit maken in de software? Allerlei variabelen maken en losse methoden? Wat nou als er meerdere boten nodig zijn? Voor dit soort *complex types* kunnen we gelukkig ook onze eigen classes maken. Bekijk het volgende voorbeeld.

```
class Boot
{
    private int Snelheid;
    private string Naam;
    private int Gewicht;
    private int AantalBemanningsleden;

    // Hieronder staan de methodes.
    public int GetSnelheid() {
        return Snelheid;
    }

    public void SetSnelheid(int snelheid)
    {
        Snelheid = snelheid;
    }

    public void Varen(int snelheid) { ... }

    public bool AnkerUitgooien() { ... }
}
```

In dit voorbeeld zie je als het goed is alle aspecten terug die beschreven stonden in het stukje tekst hierboven. Zo maak je een class! Dit kan heel gemakkelijk in Visual Studio door met je rechter-muis-knop te klikken op je C# project in de Solution Explorer en vervolgens ‘Add’ > ‘Class’ te kiezen. Nu kun je een naam voor je class ingeven en klaar ben je. Nu kun je fields en methodes toe gaan voegen. Net zoals bij je form.

```
Boot boot = new Boot();
boot.Varen(100);
MessageBox.Show("De snelheid is " + boot.GetSnelheid());
```

In bovenstaand voorbeeld zie je hoe je een eigen gemaakte class kunt gebruiken. Eigen net zoals een variable van het type Random. Je maakt een variabele aan van het juiste type (in dit geval Boot) en initialiseert deze met de **constructor** (dat is een soort van methode met dezelfde naam als de class). Dit gebeurt allemaal op de eerste regel van bovenstaand stukje code. Nu kun je de variabele gebruiken! Je kunt er methodes van aanroepen, zoals ‘Varen’ en ‘GetSnelheid’.

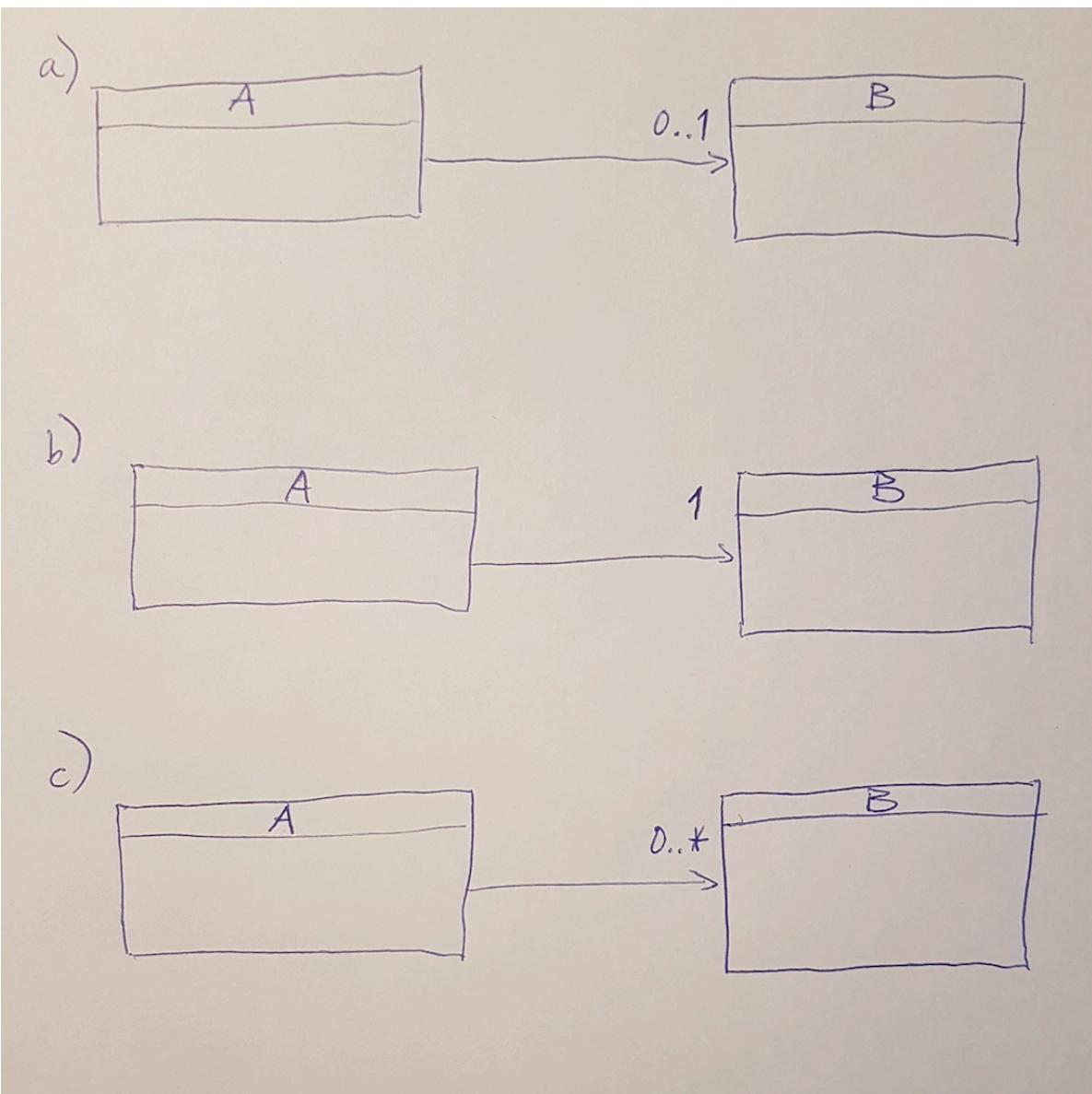
```
Boot flDutch = new Boot();
Boot titanic = new Boot();
flDutch.GetSnelheid();
titanic.GetSnelheid();
```

Tot slot kun je hierboven een van de meest krachtige aspecten van OOP zien. Elke klasse kunt u gebruiken om meerdere instanties van te maken. Nu heb ik twee boten in mijn code! Elke boot met zijn eigen invulling. Zo kan ik heel gemakkelijk boten toevoegen in de code.

Probeer eens je eigen klasse te maken en te gebruiken! Wat kunt u verzinnen en wat ga je er mee doen?

- Reference: Van Class Diagram naar code ##### Class Diagram

Al eerder heb je **class diagrams** gezien: *plaatjes waarin classes als rechthoeken worden getoond:*



In figuur

zie je steeds twee **classes**, *A* en *B*. De relatie tussen *A* en *B* is steeds anders: De pijl geeft een relatie aan tussen de **classes**, inclusief richting (**class A kent class B**). We gaan nu kijken hoe die relaties naar C#-code vertaald kunnen worden:

Mogelijk bij (a) behorende C#-code

```
public class A {
```

```
// Fields  
private B b;  
  
}
```

Hierbij *kent class A class B*. Het **Field** krijgt vaak de naam van de **class** maar dan beginnend met kleine letter. De waarde van *b* kan **null** zijn of een object van type *B*.

In situatie (b)

```
public class A {  
  
    // Fields  
    private B b = new B();  
  
}
```

Hierbij *kent class A class B*. De waarde van *b* wordt direct ingevuld, deze zal dus niet **null** zijn.

Bij (c) behorende C#-code:

```
public class A {  
  
    // Fields  
    private List<B> bs = new List<B>();  
  
}
```

Een **object** van type A kent 0 of meer (vanwege de `0..*`) objecten van type B. Voor de naam van het **Field** (hier *bs*) wordt doorgaans het meervoud gekozen van *b*. Stel dus bijvoorbeeld dat **class B** niet *B* zou heten maar *BattleRager*, dan zou het Field *b* in plaats daarvan *battleRager* heten en het Field *bs* zou *battleRagers* worden.

Merk op dat in plaats van een **List** ook een **Array** gebruikt kan worden. + [Reference: Constructor](#)

## 2.4 Properties en encapsulation

- [Reference: private en public](#)
- [Reference: Property](#)

## 2.5 Methods

- Reference: methods in classes

## 2.6 Casting en override ToString()

- Reference: Casting en ‘as’
- Reference: Jouw object als string
- Reference: override ToString

## 2.7 Enum

- Reference: Enum

## 2.8 Gecombineerd

- Reference: Voorbeeld class Aapje
- Reference: Voorbeeld class Dierenverzorgen
- Reference: Voorbeeld class Speler
- Reference: Voorbeelden OOP: pinautomaat
- Training: ObjectOriented Invuloeufening
- Training: Oplossing bij Training ObjectOriented Invuloeufening
- Training: Traffic Light
- Training: De Marimba en de Bas

## 2.9 Challenges

- Challenge: Verzameling

## 2.10 Extra

### 2.10.1 GUI vs Domain

- Reference: GUI vs Domain

## 2.10.2 Git

- Reference: Wat is Git?
- Reference: Getting Started
- Reference: Commit, push en pull
- Reference: git
- Unit tests

## 2.10.3 File Handling en Exception Handling

- Reference: File Handling
- Training: FileHandling
- Training: Exception Handling, zie ook zip met startmateriaal.

## 2.10.4 Database

- What is a database?
- Local Database connection in C#
- Learn SQL in 60 minutes

## 2.10.5 Testing

- Automated testing
- Create and run unit tests for managed code

## 2.11 Best practices Voor de wanna-be Software Engineer

Mensen die Software Engineer willen worden zullen hier zeker van smullen: + Theorie So you wanna be a Software Engineer. Als je de volgende video van Tim Corey nog niet bekeken had is dat nu zeker een aanrader! + video Tim Corey: Top 10 C# Best Practices + Software Craftsmanship Manifesto + Clean Code - Uncle Bob / Lesson 1

## 2.12 Verder Verdiepend materiaal

- external resources
- C# Basics - video 1 t/m 39
- howto: Als ik vasthang? Over oplossen problemen

## **3 Summary**

In summary, this book has no content whatsoever.

## References

- Knuth, Donald E. 1984. “Literate Programming.” *Comput. J.* 27 (2): 97–111. <https://doi.org/10.1093/comjnl/27.2.97>.