# Inheritance (overerving)

## Wat is Inheritance?

Inheritance is het Engelse woord voor ‘overerving’. Of, in gewone taal: het idee dat kinderen bepaalde eigenschappen van hun ouders meenemen. Dit is een heel belangrijk onderdeel in programming omdat het veel mogelijkheden geeft om meerdere programma’s dezelfde dingen te laten doen zonder dat je dit voor allemaal hoeft uit te programmeren.

Eerst maar een voorbeeldje dichter bij huis: Als je in Unity een object op een ander object sleept zal deze hieraan gekoppeld worden. Dit noemen we ‘childing’. Bepaalde eigenschappen die het parent-object heeft (zoals locatie en rotatie) worden ook door het child-object overgenomen. Dit is al een simpel voorbeeld van inheritance. Omdat de parent een bepaalde locatie heeft hoeven we de child niet ook deze locatie mee te geven.

We zullen verder deze les ingaan op Abstract classes en Virtual classes.

## Abstract classes

Abstract classes zijn classes die zelf geen functionaliteit hebben maar deze uitwerking overlaten aan de classes die van hun overerven. Je zou ze kunnen zien als een template. De abstracte class schrijft voor wat de child classes voor methodes moeten hebben. De child classes hebben allemaal een implementatie.

Je zou bijvoorbeeld een abstracte class ‘Enemy’ kunnen hebben. Deze heeft een aantal variabelen en een aantal methodes, namelijk Move() en Shoot(). De child classes van Enemy zijn de Sniper-class en de Melee class. Dit zou er als volgt uit kunnen zien:

*Abstract Class Enemy{*

*Public int hp;*

*Public int speed*

*Abstract public void Move();*

*Abstract public void Attack();*

*}*

*Class Sniper : Enemy{*

*Public override void Move(){*

*// Move code hier*

*}*

*Public override void Attack (){*

*// Shoot code hier*

*}*

*}*

*Class Melee : Enemy {*

*Public override void Move(){*

*// Move code hier*

*}*

*Public override void Attack(){*

*// Shoot code hier*

*}*

*}*

*Allereerst valt op dat bij de class declaration van de child classes het woord Enemy staat (na de dubbele punt, waar je vaak het woord ‘Monobehaviour’ ziet). Dit betekent dat de class een child is van Enemy (en het betekent dus ook dat classes die je schrijft met Monobehaviour een child zijn van Monobehaviour). Dit betekent dat hij alle variables en methods van Enemy kan gebruiken alsof het zijn eigen variables en methods zijn.*

Merk op dat de methods van Enemy zelf allemaal abstract zijn en dat de childen deze overnemen met het woord ‘override’ erbij. Abstract geeft dus alleen maar aan welke methods er moeten zijn, maar de child classes vullen deze zelf in. Elke abstract method moet ook worden overschreven in de child classes. (in de volgende paragraaf zien we een voorbeeld waarin dit niet hoeft)

Het grote voordeel van Abstract classes is dat je een List<Enemy> kan maken en je gegarandeerd de Move-functie en de Shoot-functie kan aanspreken zonder dat je weet wat voor type enemy het precies is.

## Virtual Classes

Virtual classes zijn een andere manier om inheritance te gebruiken. Waar abstract classes een template zijn, zijn virtual classes een basis die uitgebreid kan worden. Een virtual class is dus een gewone class, met gewone functies. De child class mag zelf bepalen welke functies hij wel en niet wil overschrijven.

Voorbeeld:

*Public Class Enemy {*

*Public int speed*

*Public void Move(){*

*// Move script hier.*

*}*

*Public virtual void Shoot(){*

*// Shoot script hier.*

*}*

*}*

*Public Class Sniper : Enemy {*

*Public override void Shoot(){*

*// Andere shoot hier*

*}*

*}*

Je ziet weer de dubbele punt om aan te geven dat Sniper van Enemy overerft. We overriden maar 1 class. Wat er nu gebeurt is dat de Sniper nog de Move-functie van de Enemy base-class gebruikt, maar zijn eigen Shoot heeft.

Je kan een virtual class dus gebruiken als elke versie van dit soort object ongeveer hetzelfde moet doen, maar er toch wat wijzigingen zijn. Je kan het script helemaal leeg laten en alleen de functies van de base-class gebruiken, of je kan elke functie overschrijven als dit logisch is.

Mocht je de basis ook nodig hebben dan kan je Shoot.base() gebruiken

# UML

## Wat is UML?

UML staat voor Unified Modeling Language. Het is een verzameling van hulpmiddelen om een systeem te ontwerpen.   
We kennen hiervan vooral het zogenaamde Class Diagram maar UML kent ook een Use Case diagram, waarbij gekeken wordt naar hoe een applicatie wordt gebruikt. Deze wordt vooral gebruikt bij het ontwerpen van een UI om te kijken welke stappen een gebruiker moet doorlopen om een bepaald resultaat te krijgen (voorbeeld: In Hearthstone kost het erg veel klikken om een enkele kaart in het deck te wijzigen). Andere voorbeelden zijn een Sequence Diagram (in welke volgorde wordt je code uitgevoerd) of een State Machine Diagram (een overzicht van de connecties van een State Machine).

Als we het binnen development hebben over UML gaat het meestal over een Class Diagram. Een Class Diagram is een diagram waarin je weergeeft welke classes, functies en variables je programma heeft en hoe deze allemaal samenhangen om weer te geven hoe je programma werkt.

Class Diagrams zijn erg gangbaar in de wereld van Game Development omdat er vaak in teams wordt samengewerkt. In een team is het belangrijk dat iedereen weet waar ze aan toe zijn. Dit gebeurt door duidelijke afspraken te maken over wat er met alle delen van de code gebeurt. Als een nieuwe functie geschreven wordt moet meteen duidelijk zijn wat er als invoer ingaat en wat er als uitvoer uitkomt. Hierdoor kan iemand werken aan een deel van de code zonder dat hij of zij hoeft te weten wat het hele programma doet. Op deze manier wordt samenwerking dus veel makkelijker.   
Een ander voordeel ontstaat wanneer een nieuw teamlid aanschuift bij het team. Een teamlid kan snel zien hoe het project in elkaar zit en hoe alle onderdelen met elkaar samenwerken.

Kortom: Een Class Diagram wordt gebruikt voor overzicht en voor communicatie. Met een Class Diagram kan de Lead Dev makkelijker nieuwe functies inpassen in het bestaande ontwerp en kunnen andere devs snel zien hoe hun deel van de code samenwerkt met de rest van het project.

## Hoe ziet een Class Diagram eruit?

Een Class Diagram bestaat uit boxes met Class names, variabelen en functies.

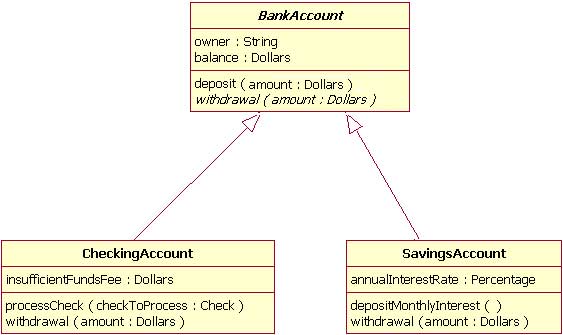


De bovenstaande box is van de class Flight. Deze class heeft drie variables: flightnumber, departureTime en flightDuration. Verder heeft de class twee functies: delayFlight en getArrivalTime. Hier is ook te zien wat de input en output van de functies zijn. delayFlight heeft als input de int numberOfMinutes en als output een Date. getArrivalTime heeft geen input en heeft als output een Date.

Er staat dus veel informatie in dit kleine blokje. Wat we nog niet zien is hoe de verschillende classes met elkaar verbonden zijn.

## Inheritance

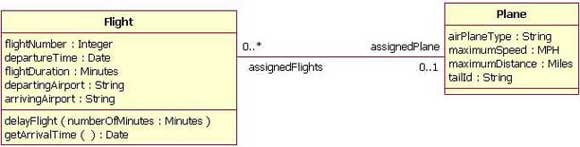
Verbinden van classes gaat met verschillende soorten pijlen. Zo is er een pijl voor Inheritance:



De twee classes CheckingAccount en SavingsAccount zijn allebei soorten BankAccounts. Dit wordt aangegeven door de open pijl. De twee erven dus alle functies en variables van BankAccount over.   
Merk verder op dat BankAccount een *italic* lettertype heeft. Dit betekent dat BankAccount een abstract class is. De functie withdrawal is hierbij ook italic, wat betekent dat elke class die overerft van BankAccount deze moet overschrijven.

## Association

Het is ook mogelijk aan te geven dat classes iets met elkaar te maken hebben. Zie het volgende voorbeeld:



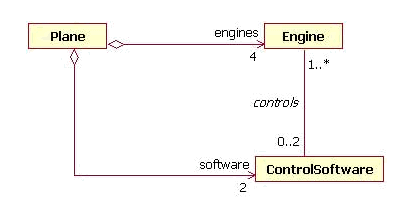
In dit voorbeeld hebben Flight en Plane met elkaar te maken. Een Flight heeft 0 of 1 Plane en een Plane heeft een willekeurig aantal Flights. De Classes staan los van elkaar maar gebruiken elkaar wel.   
Het zou prima mogelijk zijn om een ander soort relatie te hebben waarbij een Flight gemaakt wordt door een Plane en alleen de Plane informatie heeft over de Flight en niet andersom.

Een probleem hiermee zou kunnen zijn dat een Plane misschien wel kapot gaat en er voor de Flight een andere Plane gevonden moet worden. Als de Flight alleen bekend is bij één Plane dan kan deze niet vervangen worden maar moet vanuit een nieuwe Plane ook een nieuwe Flight gemaakt worden (en misschien alle tickets wel omgeboekt).

Hopelijk is aan dit voorbeeld te zien dat het belangrijk is van tevoren goed na te denken over hoe de classes met elkaar samenhangen. Een keus die onbelangrijk lijkt aan het begin van een project kan later grote problemen opleveren.

## Composition

Composition betekent dat de functionaliteit van een class opgebouwd is uit de functionaliteit van de onderdelen ervan. Bekijk het volgende voorbeeld:



Een Plane bestaat niet zomaar op zichzelf. De motor en software die ingebouwd zitten nemen ook een deel van de functionaliteit voor hun rekening. Om dit aan te geven hebben we de diamantvormige pijlen.  
Een mooi voorbeeld van composition is de Tree waar we naar hebben gekeken. Een Tree bestaat uit Nodes. Hier is dus ook sprake van een composition.

## Meer informatie

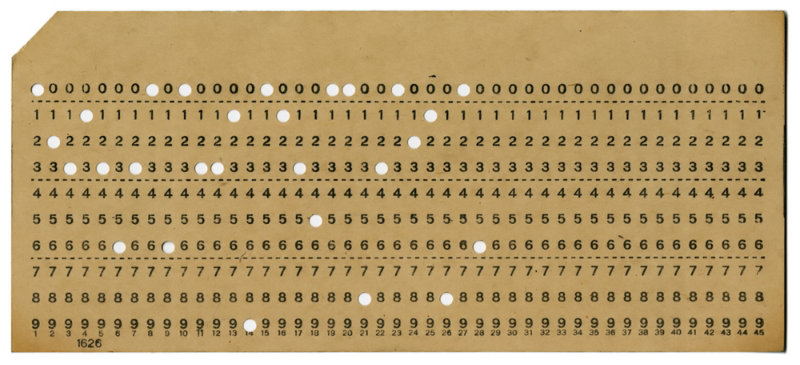
Voor meer informatie kijk op <http://www.ibm.com/developerworks/rational/library/content/RationalEdge/sep04/bell/>

Zelf UML’s maken kan gelukkig ook met gratis software. NClass is een mooie optie, maar er zijn zeker andere opties. Experimenteer zelf en kijk welke software goed voor jou werkt.

# Object-Oriented Programming

## Geschiedenis van programmeren

Toen de eerste computers werden ontworpen waren er nog geen classes en objecten. Er was zelfs nog geen programmeertaal. De eerste programmeur was Ada Lovelace (ja, de eerste programmeur was een vrouw), die een serie van wiskundige formules schreef die iets uitrekenden als je ze op de juiste volgorde uitvoerde.   
Deze computers hadden nog geen intern geheugen dus alles werd gemaakt met ponskaarten (engels: Punch Cards). Dit waren stukken karton waarin je gaatjes kon maken om aan te geven welke waardes je getallen kregen bij elke stap van je algoritme.



Deze computers konden alleen nog maar wiskundige berekeningen uitvoeren dus eigenlijk waren ze vooral te vergelijken met de huidige rekenmachines. (maar dan ongeveer zo groot als de gemiddelde studentenkamer).

De opvolger hiervan waren computers met intern geheugen waarin de programmeertaal Assembly heet. Hierin wordt direct de hardware aangeroepen en kunnen interne geheugenadressen bijvoorbeeld aangeroepen worden.   
Omdat het niet te doen is hierin een groot programma te maken (geloof me, het is echt niet leuk) werden er al snel ‘high-level’ programmeertalen uitgevonden. In deze talen kon je algemenere code schrijven die door een compiler werden omgezet naar Assembly (en dit gebeurt bij onze eigen code nog steeds op de achtergrond).

Hierdoor ontstonden meerdere programmeertalen die allemaal hun eigen standaarden hanteerden. Veel hiervan waren procedurele talen. Het idee hiervan is dat de code wordt uitgevoerd in de volgorde waarin de regels in het programma staan. Dit maakte de code natuurlijk niet erg flexibel. Object oriented programming had hiervoor een oplossing.

## Object-Oriented Programming

**To do: polymorphism**

Naast de procedurele talen ontstond ook een taal genaamd Smalltalk. Het idee van deze taal was dat alles in het programma een eigen object was (denk aan classes en functies). Dit betekende hier vooral dat elke Class zijn eigen variabelen beheerde en dat andere Classes hier niet aan mochten komen. De Classes konden wel met elkaar ‘praten’ (vandaar de naam Smalltalk) en konden dus vragen aan een andere class wat de waarde van een bepaalde variabele was. De variabelen waren dan ook allemaal standaard private (dus niet public).

Men stapte ook af van het idee van 1 centrale class die de baas was over alle andere classes. De classes lossen allemaal deelproblemen op in plaats van dat 1 enkele class alles moet doen.

Hetzelfde principe geldt voor alle Object-Georiënteerde Programmeertalen (zoals C#). Het grote verschil in Unity is dat we wat makkelijker omgaan met variabelen. De reden hiervoor is dat we in de inspector makkelijk waarden van variabelen willen kunnen veranderen en het dus onhandig zou zijn om deze allemaal private te maken.

Wat nog wel relevant is is de manier van omgaan met classes en functies. Er gelden wat regels:

1: Een class doet altijd maar 1 ding en is zo kort mogelijk. Bijvoorbeeld: een character controller is een prima class, maar als je een score wil bijhouden maak je hiervoor een nieuwe class.

2: Classes gebruiken inheritance om de verhoudingen tussen de classes aan te geven. (we komen nog op specifieke voorbeelden van inheritance zoals abstract en virtual classes)

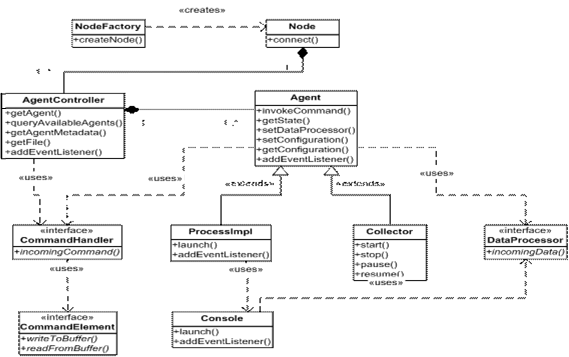
3: Functies zijn zo kort mogelijk en doen maar 1 ding. Als je meerdere stappen in je functie hebt wil je liever van elke stap een aparte functie maken.

De reden voor deze regels is dat het de code overzichtelijker maakt en makkelijker te onderhouden en uit te breiden.

Als voorbeeld kan je denken aan een functie die een lijst sorteert en daarna alle oneven getallen teruggeeft. Dit kan prima in 2 for-loops die na elkaar worden uitgevoerd.   
Een mooiere uitwerking is dat deze functie verwijst naar SorteerLijst() en GeefOnevenGetallen(), die deze taken voor je uitvoeren. Het grote voordeel hier is dat als je een snellere manier hebt gevonden om je lijst te sorteren, je alleen maar een andere functie hoeft aan te roepen en niet alles hoeft te gaan herschrijven.

## Class Diagrams (UML)

Als de regels voor OOP gevolgd worden kan de code makkelijker te lezen en onderhouden zijn, maar er zullen wel veel classes en functies geschreven worden. Om het overzicht hierover te bewaren zijn er class diagrams uitgevonden. Het idee is dat in een diagram aangegeven wordt welke classes, functies en variabelen er gebruikt gaan worden in het programma. Hiermee is het mogelijk van tevoren te zien of alles op een logische manier samenwerkt. Ook belangrijk is dat nieuwe developers zich snel in kunnen lezen in hoe alle code met elkaar samenwerkt en later de code ook netjes onderhouden kan worden (nadat je spel al gereleased is bijvoorbeeld).



Hierboven zie je een voorbeeld van een UML-diagram. Je ziet hier de blokken, die classes voorstellen. In de blokken staan ook de functies die de classes hebben. Even belangrijk zijn de verschillende soorten pijlen die de relaties tussen de classes aangeven. Hiermee kan je in een oogopslag zien welke classes informatie van andere classes gebruiken en hoe de inheritance tussen de verschillende classes werkt.

UML is dus een ontwerpmiddel maar ook een communicatiemiddel. Hoe groter je project wordt (en een spel wordt al gauw een groot project), hoe belangrijker het wordt om een class diagram te maken en te onderhouden.

## Opdracht

Zoek op internet op hoe class diagrams in elkaar zitten (er zijn meerdere manieren, maar probeer het in eerste instantie zo eenvoudig mogelijk te houden). Pak een project van vorig jaar en werk het class diagram van dit project eens uit. Hoeveel classes heb je? Hoe lang zijn je functies?

Als extra oefening: maak een nieuw ontwerp voor je project waarbij je de regels voor OOP aanhoudt. Zorg dat je classes en functies maar 1 ding doen en zo kort mogelijk worden.

# Constructor

## Wat is een constructor?

Allereerst het goede nieuws: je gebruikt al constructors! Je hebt je het alleen nooit gerealiseerd. Elke keer dat je het woord ‘new’ in je code aanroept wordt de constructor van die functie aangeroepen. Bijvoorbeeld: je hebt een script geschreven dat er als volgt uitziet:

*Public class MyClass{*

*Void Start(){*

*}*

*}*

Als je nu een nieuwe instantie van deze class aanmaakt met:

*MyClass myClass =* *new MyClass();*

Gaat Unity op zoek naar een functie in MyClass die er als volgt uitziet:

*Public MyClass(){*

*}*

Zoals je ziet heeft deze functie dezelfde naam als onze class en geen return type. Dit is een constructor. Elke class heeft in principe een default lege constructor staan, al wordt deze niet afgebeeld tenzij je hem zelf typt.

Als Unity dus naar onze MyClass gaat en ziet dat er geen constructor uitgeschreven staat, wordt de default lege constructor uitgevoerd.

## Waarvoor worden constructors gebruikt?

Het voordeel van een constructor is dat deze aangeroepen wordt als onderdeel van het maken van een nieuwe instantie van de class. Je kan deze dus op dezelfde manier gebruiken als de Start()-functie. Het nadeel van Start() is alleen dat deze werkt met Monobehaviour en je niet in al je classes Monobehaviour will inbouwen (het brengt namelijk ook beperkingen met zich mee).

Kortom: je kan de constructor gebruiken voor alles wat je wil doen als onderdeel van het maken van een nieuwe instantie van je class. Hierbij kan je denken aan het setten van variables of het aanroepen van complexere functies.

Bijvoorbeeld:

*Public class MyClass{*

*Public int vijf;*

*Public MyClass{*

*Vijf =5;*

*}*

*}*

## Meerdere constructors

Constructors worden pas echt interessant als je bedenkt dat je meerdere constructors kan hebben, zolang ze verschillende overloads hebben. Bijvoorbeeld:

*Public class MyClass{*

*Public int vijf;*

*Public MyClass{*

*Vijf =5;*

*}*

*Public MyClass(int getal){*

*Vijf = getal;*

*}*

*}*

Een functie kan een willekeurig aantal constructors hebben, afhankelijk van wat voor opstartfuncties de class zoal moet hebben.

# Coroutines

## Wat is een coroutine?

Als een functie wordt uitgevoerd, wordt deze uitgevoerd in 1 enkele frame. Een coroutine wordt met 1 stap per frame uitgevoerd. Dit geeft hele interessante mogelijkheden.

Het idee van een coroutine is dat het een functie is die wordt uitgevoerd tot een bepaald punt en dan gepauzeerd wordt tot het volgende frame. Op deze manier kan je allerlei effecten genereren. Een mogelijkheid is een object laten bewegen, langzaam van kleur laten veranderen of een zwaar script over langere tijd laten werken zodat het hele programma niet wordt opgehouden.

Er zijn twee soorten coroutines. De eerste soort doet elke frame iets zodat er een doorlopende lijn is. De andere soort voert iets uit en wacht daarna een vaste tijd totdat de volgende actie wordt uitgevoerd. Beide soorten hebben veel gebruiksmogelijkheden.

## Hoe programmeer ik een coroutine?

Een coroutine ziet er in de basis altijd hetzelfde uit. De structuur is als volgt:

*Public IEnumerator MyCoroutine(){*

*// Do something*

*Yield return null;*

*}*

De bijzondere onderdelen van deze functie zijn het return type: IEnumerator en de yield actie.

IEnumerator is een interface die dingen op een rijtje zet en ze een voor een kan afhandelen, waarbij steeds een waarde teruggegeven wordt. De for-functionaliteit gebruikt zelf ook een IEnumerator om bij te houden waar in de lus hij op dit moment bezig is.   
Je gebruikt IEnumerator hier dus om iets op een rij te zetten en stok voor stuk af te handelen. Houd dit in gedachten als we naar de volgende stappen kijken.

Yield is een keyword actie die niets anders zegt dan dat de huidige actie gepauzeerd wordt totdat hij bericht krijgt dat hij weer verder moet. De regel yield return null zegt nu dat de actie gepauzeerd wordt en dat de waarde null teruggegeven wordt aan de functie die de coroutine heeft aangeroepen. Volgende frame gaat de coroutine weer verder waar hij gebleven was.

Een voorbeeld om dit duidelijk te maken:

*Public IEnumerator SimpleCoroutine(){*

*// Do something*

*Yield return null*

*// Do something else.*

*}*

Wat deze coroutine doet is stap voor stap:

1: Er wordt een IEnumerator aangemaakt die berekent welke stappen er nodig zijn

2: De functie wordt uitgevoerd tot aan de eerste yield

3: De yield zet de uitvoering op pauze en onthoudt waar hij is gebleven bij de uitvoering

4: Het volgende frame wordt de coroutine weer opgestart en gaat deze verder vanaf de yield

5: De rest van de coroutine wordt uitgevoerd.

Dit is een redelijk makkelijk voorbeeld maar bekijk nu het volgende voorbeeld:

*Public IEnumerator ForCoroutine(){*

*For ( int i = 0; i < 10 ; i++){*

*// Do something*

*Yield return null;*

*}*

*}*

Deze coroutine voert 10 frames lang elke frame het stuk code uit. Dit geeft veel mogelijkheden als je elke frame iets uit wil voeren. Maar soms wil je een tijdje wachten tot je de volgende stap uit wil voeren. Bekijk hiervoor het volgende stuk code uit de Unity documentatie:

*Public IEnumerator Fade() {*

*for (float f = 1f; f >= 0; f -= 0.1f) {*

*Color c = renderer.material.color;*

*c.a = f;*

*renderer.material.color = c;*

*yield return new WaitForSeconds(.1f);*

*}*

*}*

Het verschil ten opzichte van het vorige voorbeeld is dat de return veranderd is naar “return new WaitForSeconds(.1f)”. WaitForSeconds is een functie die alleen werkt met coroutines en de IEnumerator vraagt een tijdje te wachten voordat hij doorgaat met het uitvoeren van de coroutine.

Met deze gereedschappen kan je functies over langere tijd uitvoeren. Er zijn geen vaste regels voor wanneer je wel of geen coroutine zou moeten gebruiken maar met wat creativiteit liggen overal mogelijkheden.

## Opdracht:

Maak een loading bar voor een scene die een Coroutine gebruikt om te laten zien hoe ver de scene is met laden. (het is waarschijnlijk beter om het zonder coroutine te doen, maar het is alsnog een prima oefening).