



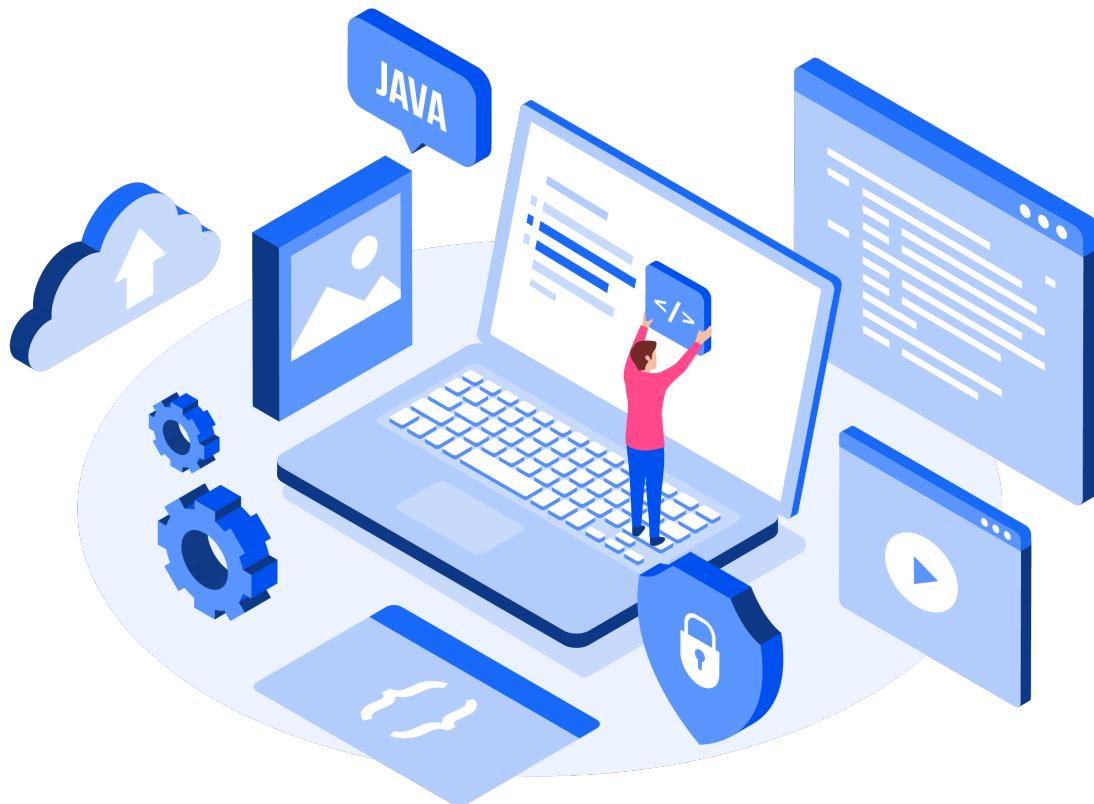
Programmeren met Java

In de cursus Programmeren met Java leer je structuren van object-georiënteerd programmeren (OOP). Met OOP-structuren leer je ook je code leesbaar, uitbreidbaar en overzichtelijk te houden. Je gaat interfaces en abstracte klassen gebruiken, daarnaast ga je unit-testen schrijven om de code automatisch te kunnen testen. Uiteindelijk leer je gebruik te maken van externe code middels Maven.

1. Inleiding

1.1 Backend development

Backend development gaat over het bouwen van software. Software komt origineel vanuit de wiskunde. Simpel gezegd wordt het gebruikt om instructies te beschrijven zodat een computer deze instructies kan uitvoeren.



Grapig genoeg was het concept van de computer al uitgevonden voordat er daadwerkelijk een machine bestond die deze instructies zou kunnen uitvoeren! Een computer was iets dat instructies kon *computen* (uitvoeren), vandaar de naam. Het automatisch uitvoeren van instructies bleek erg handig te zijn bij bijvoorbeeld het tellen van de bevolking. Dit werd gedaan door een van de eerste vormen van de computer: de tabulating machine. Het bedrijf dat deze machine heeft gebouwd, de Computing-Tabulating-Recording Company (CTR), werd later IBM: vandaag de dag één van de grootste IT bedrijven ter wereld.

Deze tabulating machines konden veel taken automatiseren, maar konden nog niet worden geprogrammeerd om *verschillende* taken te kunnen doen. Dit werd pas uitgevonden in de Tweede Wereldoorlog, toen er een computer nodig was om de encryptie van de Duitse Enigma machine te kunnen breken. (Hier is een hele goede film over gemaakt: [The Imitation Game](#).) Omdat de encryptie van Duitse berichten iedere dag veranderde, was er maar 24 uur beschikbaar om de code te kraken voordat deze nutteloos was. Onmogelijk voor een mens, maar prima te doen voor een computer.

Automatisering

Computers kunnen veel meer informatie verwerken en bewerken dan de mens ooit zou kunnen. Bovendien kunnen ze dat ook nog in een fractie van de tijd! Dit maakt ze erg geschikt om taken van ons over te nemen zodat wij veel efficiënter kunnen werken. Dit proces - het vervangen van menselijke arbeid door computers of computerprogramma's - noemen we **automatisering**. En hoewel dit ontzettend veel voordeelen biedt, kleven er ook nadelen aan.

Een computer doet precies wat je zegt, niet wat je wil.

Zo moet je de regels van het proces dat je wil automatiseren *heel specifiek vastleggen*. Een computer doet immers niets anders dan instructies uitvoeren en zal niets invullen op eigen initiatief. Heb je deze regels niet correct vastgelegd? Dan zal de computer het proces dus ook niet goed uitvoeren. Om deze reden zijn geautomatiseerde processen over het algemeen niet flexibel.

Hardware vs Software

De allereerste machines die werden gebouwd om processen te automatiseren bestonden volledig uit **hardware**: de tastbare onderdelen van een computer(systeem). Dit waren fysieke machines die volledig ontworpen waren voor het uitvoeren van één taak (denk aan een zaagmolen of een trein). Hierdoor waren deze machines niet in staat om een andere taak uit te voeren, totdat de machine opnieuw werd opgebouwd. Dit maakte de machines zeer inflexibel. Zo ontstond de vraag naar een manier om een machine te kunnen programmeren, zodat wat de machine deed dynamisch kon worden aangepast. De eerste vorm hiervan zagen we terug bij weefmachines waar door middel van gaten in een kaart de machine patronen kon weven.

Sindsdien is de behoefte aan flexibiliteit steeds verder toegenomen en werden er geschreven talen ontwikkeld om computers van instructies te kunnen voorzien. Deze talen noemen wij programmeertalen. Met deze programmeertalen schrijven wij **software**: verzamelingen van instructies die ingewikkelde acties kunnen uitvoeren.

SaaS

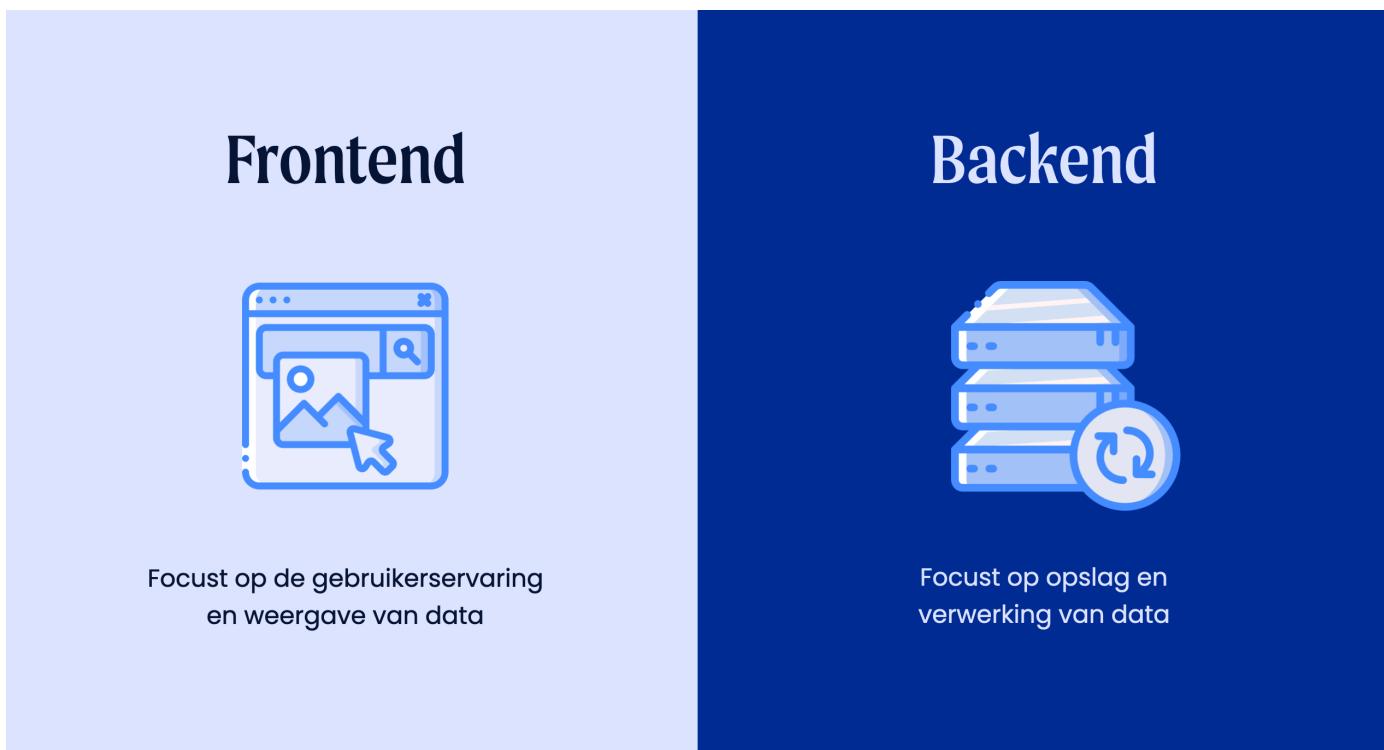
Software was iets wat je vroeger alleen op de computer kon installeren. Maar terwijl alle ontwikkelingen in de techniek ons leven veranderde, waren we dankzij de opmars van het internet ineens niet meer aan onze desktops gebonden. We wilden onze e-mail kunnen ontvangen op het werk én in de trein, onze bestanden op een plek opslaan zodat we er vanaf iedere willekeurige PC bij konden en documenten bewerken zonder daadwerkelijk een tekst-editor te hoeven installeren.

SaaS (Software as a Service) werd een baanbrekende nieuwe ontwikkeling in de geschiedenis van softwareontwikkeling. Het was een buzz-word waarmee je bij iedere nieuwe, hippe, start-up mee om je oren werd geslagen. Maar ook de grote bedrijven ontwikkelden zich snel om deel uit te maken van deze nieuwe beweging. Waar het web voorheen alleen uit websites bestond, was nu ook online software geboren. Dit heeft vele veranderingen in programmeertalen met zich meegbracht.

Wanneer je software installeert op je computer, is dit geschreven in één enkele programmeertaal. Die is zowel verantwoordelijk voor de dataverwerking, het uiterlijk (de UI) en de verbinding tussen die twee (de logica). Net als dat de applicatie een plekje heeft op het besturingssysteem, heeft de data dat ook.

Een webbrowser, daarentegen, is alleen in staat om aangeleverde webpagina's te geven. Wanneer de browser wordt afgeladen, zullen dus ook alle gemaakte aanpassingen verdwenen zijn. Zo is het principe van een **frontend** en een **backend** ontstaan. Backend ontwikkeling vindt plaats op de webserver en is voor de gebruiker onzichtbaar. Het bevat allerlei verborgen processen die ervoor zorgen dat data opgeslagen kan worden in de database, en later weer opgevraagd kan worden. Het zorgt ervoor dat je posts op Instagram altijd terug kunt zien op je profiel en terug kunt vinden welke recepten je had geliked op Allerhande.nl. Frontend

ontwikkeling gaat over het ontwikkelen van het deel van de applicatie dat gebruikers kunnen zien. Hierdoor associëren veel mensen frontend vaak alleen met styling en opmaak, maar vergis je niet: ook de frontend bevat een hoop logica om een applicatie te laten werken. Het zorgt ervoor dat de applicatie reageert op wat de gebruiker doet en vraagt.



Een backend developer heeft daarmee een onzettend belangrijke baan: je bent verantwoordelijk voor het creeren van de onderliggende structuur die de hele applicatie ondersteunt. Je bouwt logica om data op de juiste manier op te kunnen slaan of weer uit te lezen uit de database. Veranderd de gebruiker zijn of haar profielfoto? Dan zorg jij ervoor dat dit op de juiste manier afgehandeld wordt. Uiteindelijk bouw jij als backender robuuste en schambare systemen die grote hoeveelheden data en verkeer aankunnen. Zonder backend zouden webapplicaties niet bestaan!

Tip: Vraag je je wel eens af hoe programmeertalen zijn ontstaan? Hoe je kunt programmeren in een programma dat op zijn beurt ook geprogrammeerd is? [Dit filmpje legt het heel fijn uit!](#)

1.2 Software Levensloop

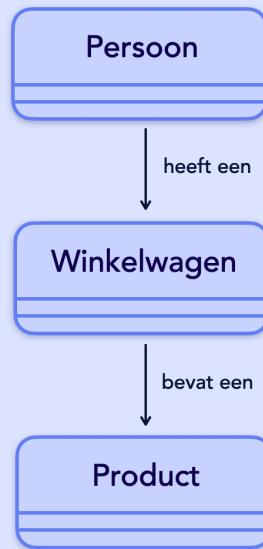
De ontwikkeling van een software programma kent een **levensloop**: het begint bij een idee en eindigt bij een draaiend programma. In dit hoofdstuk geven we een kort overzicht van de stappen die we doorlopen tijdens het ontwikkelen van een stukje software. Geen zorgen: gedurende deze cursus zullen we nog uitgebreid stil staan bij deze stappen.



Ontwerp

Wanneer je software gaat bouwen, doe je dit vaak om een probleem op te lossen of om een taak te automatiseren. Zodra we als ontwikkelaars duidelijk hebben wat er gevraagd wordt, kan er begonnen worden aan het ontwerp van de software. Hiermee bedoelen we zowel een grafisch als een technisch ontwerp. Wanneer de applicatie ook een visueel aspect kent (een *User Interface*, zoals bijvoorbeeld een webpagina) kun je ongelijkveld begrijpen dat men daar een ontwerp voor maakt, waarbij er rekening gehouden wordt met zowel het uiterlijk als de gebruiksvriendelijkheid.

Een technisch ontwerp is echter net zo belangrijk. Dit is alleen wat abstracter: hiermee bepalen we hoe onze software in elkaar zit. Dit lijkt in eerste instantie misschien gek, maar het is erg goed mogelijk een softwaremodel uit te tekenen! In deze cursus gaan we dit doen door middel van **Unified Modeling Language (UML)**. Hiermee kunnen we vooraf beschrijven hoe onze software in elkaar zit. Stel, we maken een webshop, dan hebben we een **Persoon**, een **Winkelwagen** en een **Product** nodig. Deze zijn op de volgende manier aan elkaar verbonden:



Pas nadat we een goed beeld hebben van hoe we het probleem willen oplossen, kunnen we beginnen aan het bouwen van de software.

Bouwen

Is het ontwerp klaar? Dan kunnen we beginnen met bouwen. Wanneer we een brief of essay schrijven doen we dit meestal in een tekst editor zoals Microsoft Word. Bij programmeren gebruiken we ook editors om onze code te schrijven. Technisch gezien zou een tekst-editor zoals Notepad al voldoende zijn om te kunnen programmeren. Maar zag nou zelf, welk voorbeeld leest nou lekkerder?

```

communicatie - Main.java
12
13 >  public class Main {
14
15 >      public static void main(String[] args) {
16          System.out.println("Hello World!");
17      }
18  }
19
20
  
```

```

Untitled — Edited
Times Regular 32 B I U 1,1
public class Main {

    public static void main(String[] args) {
        System.out.println("Hello World!");
    }
}
  
```

Code editors noemen we **IDE's**. IDE staat voor Integrated Development Environment. Naast syntaxhighlighting, ondersteunt een IDE je in het gehele ontwikkelproces door woorden voor je af te maken, verbeteringen te suggereren en je te wijzen op fouten. Er bestaan IDE's in alle soorten en maten, maar tijdens deze leerlijn zul je gebruikmaken van IntelliJ. IntelliJ is een betaald product, in tegenstelling tot andere bekende gratis IDE's (zoals Visual Studio Code of Atom). Maar daar krijg je dan ook een hoop extra functionaliteit voor terug! Dit hoeft je uiteraard niet zelf te kosten. Tijdens je studielid bij NOVI krijg je de licentie van ons cadeau. De installatieinstructie voor IntelliJ vind je later in deze cursus.

Testen

Na het ontwerpen en het bouwen van onze software, is het tijd om te kijken of deze wel correct geprogrammeerd is. We willen dus de correcte werking van de instructies vaststellen, maar het zou kunnen zijn dat we met bepaalde edge-cases (uitzonderingssituaties) geen rekening hebben gehouden. Stel dat we een programma hebben gemaakt dat twee getallen door elkaar deelt. Dan willen we naast het controleren of het delen (10 gedeeld door 2 of 10 gedeeld door 5) werk zoals we verwachten, ook zorgen dat alles blijft werken als we 10 delen door 0.

Het testen van software kan op twee manieren: handmatig of automatisch. Bij handmatige testen worden testcases door een mens uitgevoerd. Denk bijvoorbeeld aan het controleren of het kopen van een product bij een webshop nog werkt. Handmatig testen kan erg handig zijn, omdat de tester erg flexibel is en diens intuïtie kan gebruiken. Het nadeel is echter dat het veel tijd kost en dat repetitief werk gevoelig is voor fouten: mensen kunnen nu eenmaal een mindere dag hebben. Laat een computer nou net goed zijn in uitvoerende repetitieve taken! Je raadt het al: ook dit klusje vraagt om automatisering.

Om een automatische test te doen, schrijven we code die verifieert of onze eerder geschreven code doet wat we verwachten. Dit doen we door de test een specifieke **invoer** te geven (bijvoorbeeld '2') en vervolgens te controleren of de **uitvoer** van het programma (10 gedeeld door 2) daadwerkelijk 5 als uitkomst geeft. Het voordeel van het automatiseren van dit soort testen, is dat het eenvoudig wordt om alle testen regelmatig te draaien. Zo weet je iedere keer als je een wijziging aanbrengt aan het programma of je niet per ongeluk iets kapot hebt gemaakt.

Verpakken

Nu we de geschreven software getest hebben, is het tijd om de code te verpakken tot iets wat door iemand anders *uit te voeren is*. Denk bijvoorbeeld aan transformeren van je geschreven code naar een .exe- of .dmg-bestand, zodat een ander dit met twee muisklikken kan draaien op diens computer. In het geval van Java maken we een zogenoemd .jar-bestand aan. Dit bestand bevat al je code, inclusief eventueel overige bestanden die je programma nodig heeft. Dit pakketje kan vervolgens op iedere computer gedraaid worden die Java heeft geïnstalleerd.

Zodra we onze software verpakken is het vaak ook een goed idee om het een versienummer te geven. Zo is het voorde dat gebruikers makkelijk te zien of er een update heeft plaatsgevonden in de software. Ook als een gebruiker een *bug* vindt is het belangrijk dat er kan worden vermeld in welke versie het fout is gegaan. Het kan bijvoorbeeld zo zijn dat een bepaald programma wel draait onder Windows versie 11, maar niet op Windows versie 10. Oftewel, versionering van software is erg belangrijk.

Draaien

Nu de software ontworpen, gebouwd, getest, verpakt en geversioneerd is, is het tijd om de software te draaien. Als de software een webapplicatie betreft (een applicatie die benaderbaar is vanaf het internet) moet deze niet geïnstalleerd worden op de computer van de gebruiker, maar op een **server**. Deze server is vervolgens verbonden met het internet en geeft de gebruikers op die manier toegang tot de gemaakte applicatie. Er zijn vele oplossingen om software op een server te installeren: van handmatige installaties op een eigen server tot volledig automatisch op een cloud-omgeving.

We hebben nu de hele lifecycle doorglopen die hoort bij het bouwen van software. Dit is niet iets wat eenmalig gebeurt, maar een continu proces dat bij iedere verandering aan de software weer van voor af aan begint. Erst ontwerpen we de nieuwe wijziging, dan bouwen en testen we deze, vervolgens verpakken we de software, geven het een hoger versienummer en installeren het. Daarna wordt het proces herhaald bij de volgende wijziging.

1.3 Java en het ecosysteem

Binnen deze cursus zullen we werken met de programmeertaal **Java**. Het is belangrijk om te weten dat alleen kennis van Java niet genoeg is om een programmeur te zijn. Er is een heel ecosysteem om de taal heen ontwikkeld, die je ook nodig hebt om een applicatie te kunnen schrijven. In deze paragraaf leer je hier meer over.

Java

Als eerste de programmeertaal Java: je herkent de taal aan het logo van het koffie kopje. Dit is was bedoelt als erkenning voor de Java-ontwikkelaars, omdat ze zo veel koffie dronken tijdens hun ontwikkelwerk.



De taal werd al in het begin van de jaren '90 uitgevonden. Het dus al best een oude taal, maar dit betekent niet dat er niet meer aan ontwikkeld wordt! Ieder half jaar wordt er een nieuwe versie met verbeteringen uitgebracht. Gelukkig wordt Java gekenmerkt door diens **backwards compatibility**. Dit houdt in dat een programma dat geschreven is in Java versie 1, nog steeds zou moeten kunnen draaien in de laatste versie. Om deze reden is Java voor bedrijven erg aantrekkelijk: het is zeer betrouwbaar. Het nadeel is echter dat er nog constructies in zitten die inmiddels verouderd zijn, maar vanwege backwards compatibility niet verwijderd kunnen worden.

Object georiënteerd programmeren

Java is een object georiënteerde (**Object Oriented**) taal. Dit is een programmeer-paradigma waarbij we programma's schrijven op basis van "objecten" die acties kunnen uitvoeren en relaties hebben tot elkaar. Denk bijvoorbeeld aan een kist met kinderspeelgoed: ieder stuk speelgoed heeft unieke eigenschappen en speeltjes die je ermee kunt spelen. Zo kan een speelgoedauto rijden, een speelgoedvliegtuig vliegen en een lego-poppetje kan lopen of vervoerd worden in de speelgoedauto- of vliegtuig. Elk stuk speelgoed is een "object" die ieder verschillende "methoden" (acties) heeft die je ermee kunt uitvoeren.

Andere typen programmeertalen werken anders. Zo heb je **procedurale** talen, die gericht zijn op het creëren van een reeks stapsgewijze instructies die de computer moet volgen. Deze talen zijn goed voor eenvoudige taken, maar het is niet mogelijk om onderlinge relaties te beschrijven. Zo heb je ook **functionele** programmeertalen. Deze talen zijn gebaseerd op het idee van wiskundige functies, waarbij dezelfde invoer altijd dezelfde uitvoer zal produceren. Deze talen zijn goed voor complexe problemen, maar hebben geen mogelijkheid om met objecten en klassen te werken.

Omdat Java object georiënteerd is, is de code gemakkelijk te onderhouden, hergebruiken en uit te breiden. Door de logica te bundelen in verschillende objecten - met ieder hun eigen dataset en gedrag - zijn deze bundeltjes gemakkelijk inwisselbaar of toepasbaar op meerdere plekken en hebben ze ook relaties met elkaar. Hierdoor zijn we in staat om complexe programma's te bouwen.

Libraries

Veel problemen en taken binnen programmeren zijn relatief standaard. Je kunt je voorstellen dat de code die tijden in 24-uurs notatie omzet naar 12-uurs notatie, door meerdere programma's gebruikt zou kunnen worden. Gelukkig hoeft je voor veel standaard problemen zelf geen code meer te schrijven. De makers van Java hebben dit al voor je gedaan en hun uitwerkingen beschikbaar gemaakt. Dit wordt gedaan in de vorm van bundels die we **libraries** (bibliotheeken) noemen. Zo is Java uitgerust met talloze *interne* libraries, zoals bijvoorbeeld Math, waarmee je allerlei wiskundige operaties kunt uitvoeren.

Oké zijn er *externe* libraries (geschreven door ontwikkelaars zelf) die je kunt gebruiken in jouw code. Bijvoorbeeld code om met een database te kunnen praten of code voor gezichtsherkenning. Indien je een standaard probleem hebt, is het dus zeer gebruikelijk dit op te lossen met een (externe) library. Dit zul je in de praktijk ook vaak tegenkomen. Daarom loont het om de vele libraries die het ecosysteem biedt, goed te leren kennen.

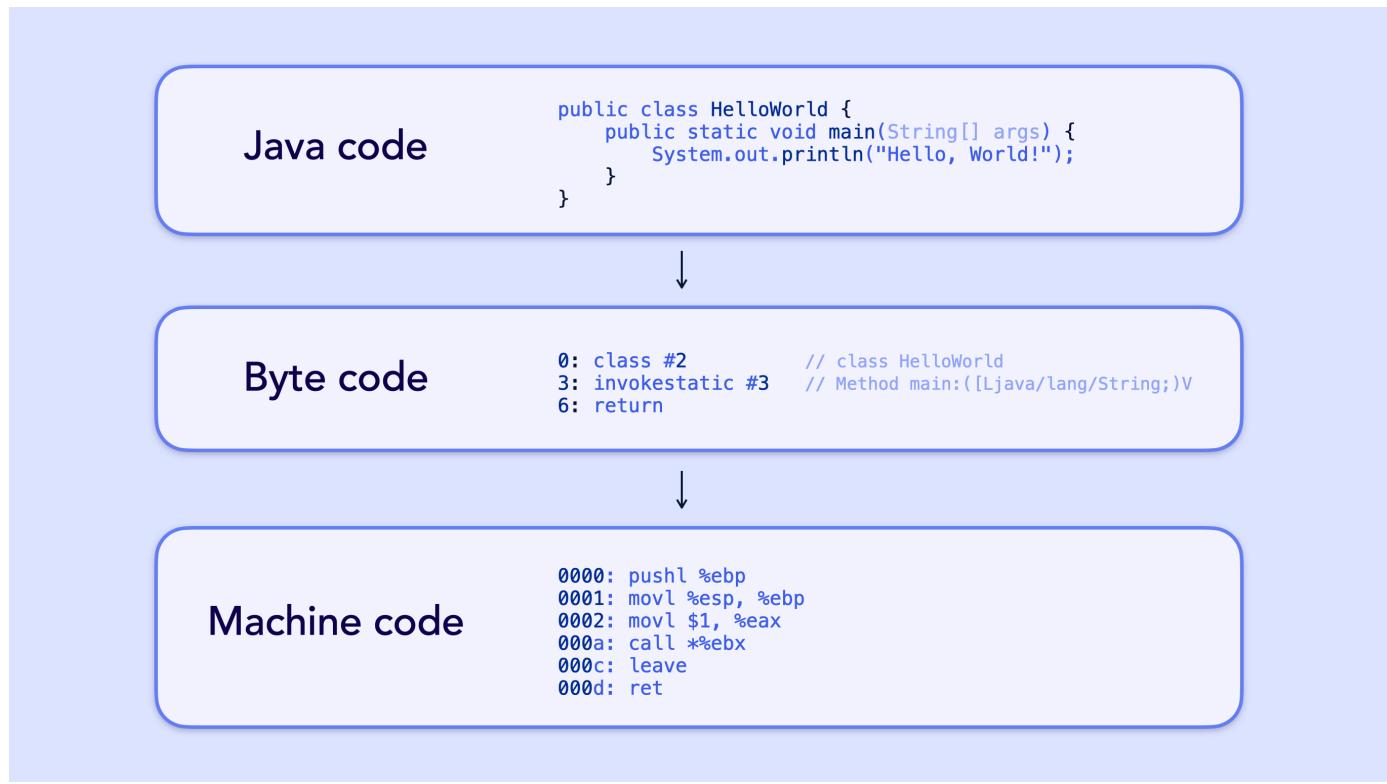
Frameworks

Naast libraries ga je ook frameworks tegenkomen. Een **framework** is een verzameling van meerdere (grote) libraries die dictieren hoe jij je software moet schrijven. Daarom noemen we frameworks ook wel *opinionated*: ze hebben een mening over hoe jouw software opgebouwd moet worden. Dit zorgt ervoor dat je niet volledig vrij bent in hoe je code schrijft, maar je krijgt er heel veel voor terug! Wanneer je een webapplicatie wil ontwikkelen met Java, is het Spring framework erg populair. Het zorgt ervoor dat we een hoop code niet meer zelf hoeven te schrijven. Je kunt je voorstellen dat het werken met frameworks in het bedrijfsleven daarom de norm is. In de cursus Spring Boot leer je hoe je applicaties bouwt met het Spring Framework.

Java Virtual Machine (JVM)

Java is een high-level programmeertaal die goed leesbaar is voor mensen. Maar hoewel programmeren misschien voelt alsof je computertaal spreekt, zijn talen zoals Java voor computers nog steeds niet te begrijpen. Computers zijn namelijk binair: ze kunnen alleen 0's en 1's interpreteren. Om dit nog iets lastiger te maken is het ook nog zo dat verschillende besturingssystemen (Windows/Mac/Linux) andere binaire code nodig hebben voor hetzelfde probleem.

Uiteindelijk moet de code die wij schrijven dus worden teruggevraagd naar binaire code die werkt op een specifiek besturingssysteem. Als we dit telkens handmatig zouden moeten doen zouden we niet eens meer aan programmeren toe komen... Gelukkig heeft Java hier een mooie oplossing voor! Door middel van de Java Virtual Machine (**JVM**) kunnen we onze code draaien op ieder besturingssysteem, zonder dat we daar zelf iets voor hoeven te doen. De Java compiler zet onze geschreven code om naar bytecode. Dit is het tussenformaat dat de JVM begrijpt. Deze bytecode wordt vervolgens door de JVM omgezet naar binaire code. En omdat er voor ieder besturingssysteem een JVM is geschreven, wordt de bytecode dus automatisch omgezet naar binaire code die geschikt is voor dat systeem. Dit zorgt ervoor dat wij ons hier tijdens het programmeren nooit mee bezig hoeven te houden.



Eerdere versies van Java waren relatief traag in verhouding tot andere talen, omdat hier dus nog een extra vertaalslag nodig was om bytecode om te zetten naar binair. Maar inmiddels bestaat dit probleem niet meer, omdat de JVM slim genoeg is om patronen binnen de code te ontdekken en deze langzaam te optimaliseren op basis van hoe het programma in de praktijk gebruikt wordt.

Een ander bijkomend voordeel van de JVM is dat het ook de deur opende voor het ontwikkelen van nieuwe programmeertalen. Het maakt de JVM bij het omzetten naar binaire code namelijk niet uit waar de bytecode vandaan komt, zolang het maar bytecode is. Dit zorgde ervoor dat er allerlei andere talen werden ontwikkeld, zoals Kotlin, Scala, Groovy en Clojure. En omdat ze allemaal op de JVM draaien, kan Java-code ook gebruikt worden in Kotlin of Scala en vice versa. Al deze talen delen dus hetzelfde ecosysteem waardoor ook de libraries uitwisselbaar zijn. Om deze reden is de JVM een enorm krachtig stukje software.

Compiler

We hebben het net gehad over het omzetten van Java-code naar bytecode. Dit proces noemen we **compilieren** en wordt uitgevoerd door iets dat we een **compiler** noemen. Dit betekent ook dat de compiler foutmeldingen zal geven als jouw code een fout bevat: de omzetting naar bytecode is dan namelijk niet mogelijk tot de fout wordt opgelost. Gelukkig zijn compilers er niet alleen om je te straffen als je iets fout doet: ze helpen je ook om fouten te voorkomen. Compilers lezen jouw volledige project uit, dus zijn ze in staat verbindingen te zien die voor ons als menselijke lezer moeilijk te herkennen zijn. Zo kan het gebeuren dat de schrijfwijze van de code (**syntax**) correct is, maar dat de compiler dit toch aanmerkt als fout. Die ziet namelijk dat wat er nu geschreven staat nooit zal werken wanneer het daadwerkelijk gedraaid wordt. Ten slotte worden er tijdens het compileren ook nog enkele optimalisaties doorgevoerd, waardoor je code veel sneller wordt.

Garbage collection

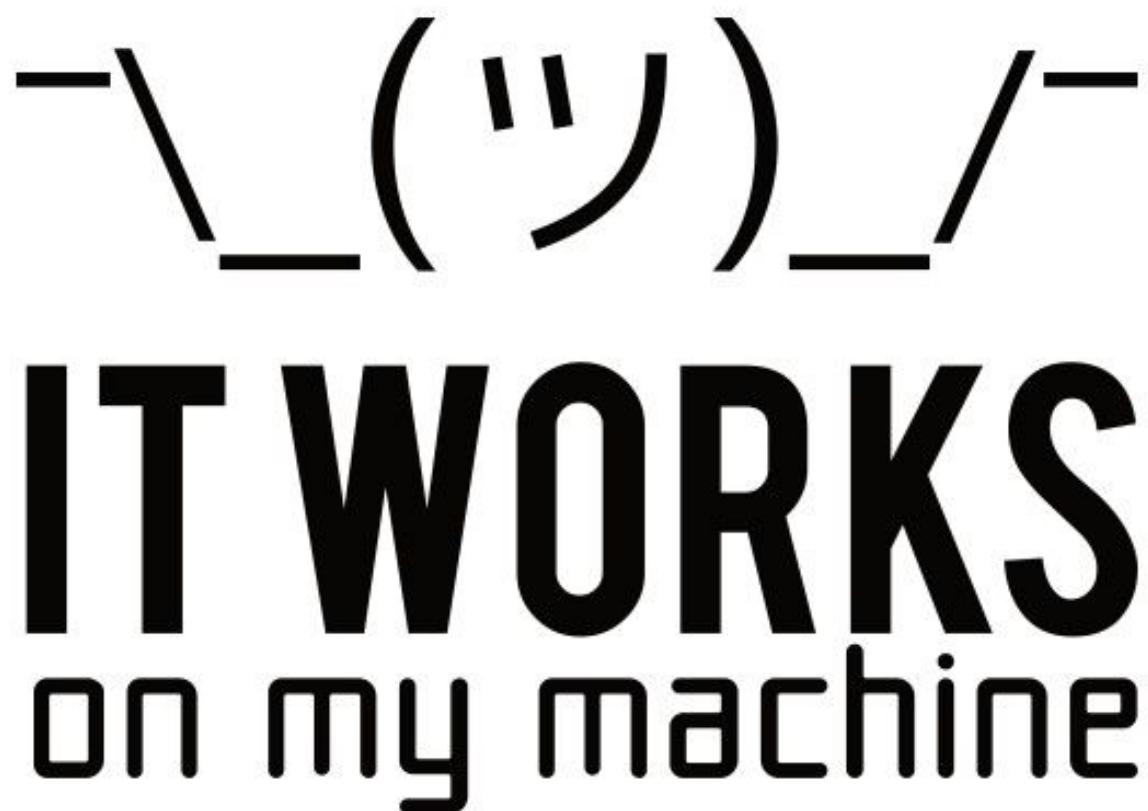
Een andere toepassing die de JVM levert, is het zogenaamde **garbage collection**: optimalisatie van geheugengebruik. Alle data die je binnen jouw project (tijdelijk) wil bewaren, moet ergens opgeslagen worden. Dit is iets dat bijna alle programma's continu doen. Het probleem is echter dat het interne werkgeheugen (**RAM**) van je computer beperkt is. Oftewel, op een gegeven moment is het geheugen vol en kan er geen nieuwe data worden opgeslagen, waardoor de applicatie niet meer werkt.

Om dit te voorkomen moet er geheugen worden vrijgegeven als data niet meer wordt gebruikt. Maar hoe doen we dat? In talen zoals C moet dit handmatig gedaan worden: de ontwikkelaar is hier zelf verantwoordelijk voor. Vergeet je een stukje geheugen vrij te geven? Dan spreken we van een **memory leak**. Hoe langer het programma draait, hoe meer geheugen er bezet wordt gehouden omdat een deel nooit meer wordt vrijgegeven. Tot uiteindelijk het geheugen vol zit en de applicatie crasht.

Gelukkig heeft Java een eigen **garbage collector**: er wordt continu gekeken of data nog gebruikt wordt. Zo niet, dan wordt dit automatisch vrijgegeven. Hier hebben wij als ontwikkelaars dus geen oogkijken meer naar: de JVM regelt alles.

Build tooling

Om het proces binnen de software levensloop te ondersteunen (bouwen, testen, verpakken, draaien) gebruiken we **build tooling**. Heel plat gezegd zijn build tools programma's die neventaken voor je kunnen draaien. Denk aan compileren, het beheren van externe libraries, testen, verpakken, versieren en het draaien van je software. Al deze taken zijn prima uit te voeren op je eigen computer, maar de kans is groot dat jouw computer niet de enige computer is waar de code gecompileerd en getest moet worden. Als je samenwerkst met iemand anders (of een heel team!) zal dit namelijk ook op een andere computer moeten gebeuren.



Dankzij build tooling is de werkwijze voor het uitvoeren van deze taken gestandaardiseerd. Hierdoor voorkom je de situatie dat het wel werkt op jou computer, maar niet op die van een ander. Binnen Java bestaan twee populaire build tools: Maven en Gradle. Tijdens deze leerlijn zullen we **Maven** gebruiken.

Versiebeheer van je code

Als ontwikkelaar werk je vaak in teamverband, waardoor je vaak in de situatie komt waarin je met meerdere mensen tegelijkertijd aan dezelfde code werkt. Maar ook als jij afwisselend vanaf je laptop en daarna weer vanaf je desktop aan hetzelfde project wil werken, is het gemakkelijk om wijzigingen kwijt te raken. Om ervoor te zorgen dat we elkaar wijzigingen niet overschrijven, hebben we ook hier een handige tool voor nodig. Dit noemen we **version control tooling**, oftewel: versiebeheer.

Ook hier voor bestaan meerdere oplossingen, waarvan wij gebruik zullen maken van Git. Git zorgt ervoor dat wijzigingen goed gesynchroniseerd worden tussen meerdere computers. Kun je gemaakte wijzigingen gemakkelijk ongedaan maken en terugkijken welke wijzigingen er in het verleden zijn gemaakt.

Databases

We hadden het al gehad over het opslaan van data in het werkgeheugen (RAM). Dit is echter tijdelijk: de data verdwijnt als het programma wordt afgesloten. Je kunt je voorstellen dat we sommige data zo willen opslaan dat het ook blijft bestaan als het programma wordt afgesloten, zodat het weer beschikbaar is als het programma opnieuw wordt opgestart. Dit noemen we data persisteren.

Een goede manier om data te persistenteren is door gebruik te maken van een database. Deze komen in vele soorten en maten, maar tijdens deze leerlijn maken we gebruik van een PostgreSQL database. Een database is niets anders dan een verzameling van tabellen waarin data opgeslagen staat. Als we bijvoorbeeld data van een *Person* willen opslaan, zouden we een *Person* tabel kunnen aanmaken. Deze kan er zo uitzien:

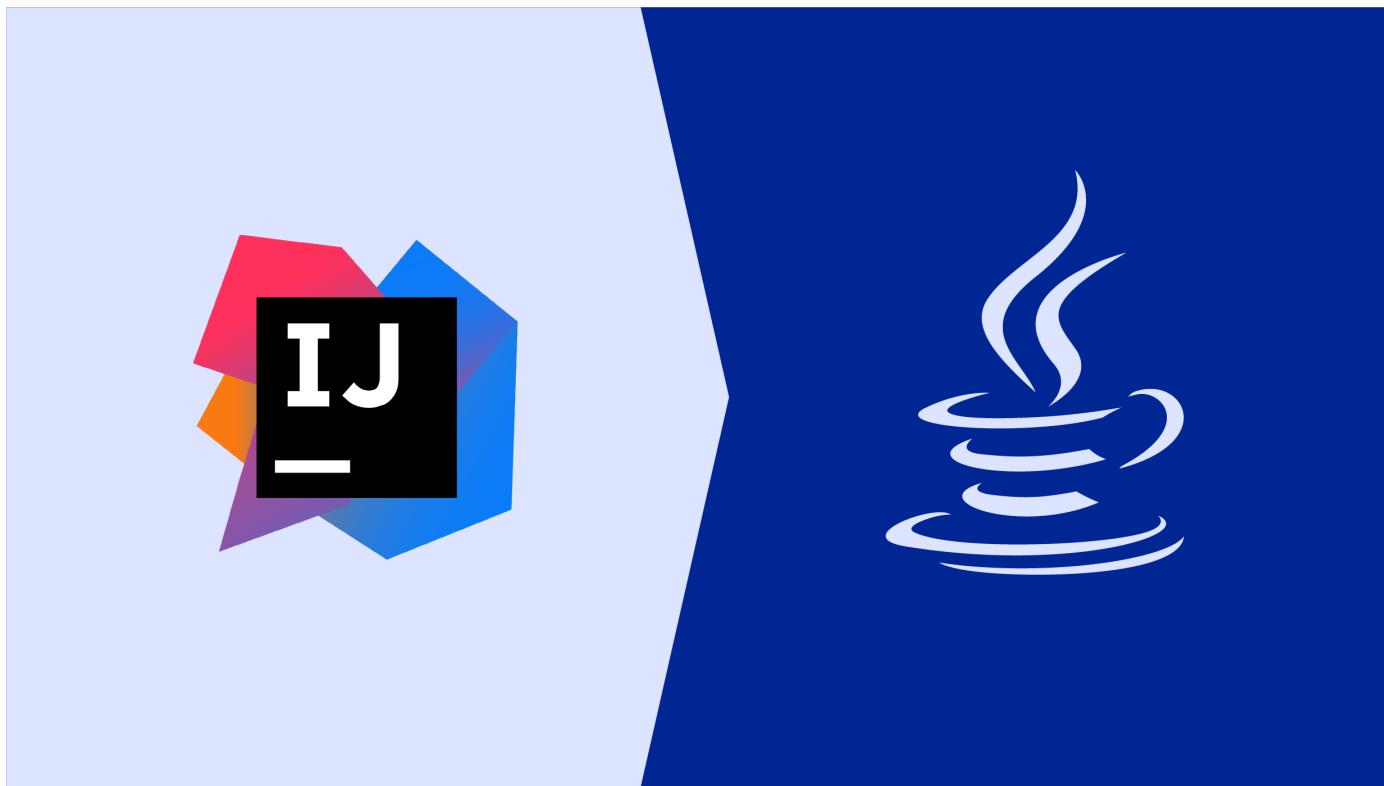
ID	Naam	Leeftijd
1	“Bob”	15
2	“Alice”	28
3	“Marieke”	21

Deze data kunnen we vervolgens uitlezen met behulp van een speciale taal: **SQL** (Structured Query language). Zo zouden we bijvoorbeeld alle personen kunnen opvragen uit de Persoon-tabel, of alleen iedereen die ouder is dan 20.

1.4 Installatie IntelliJ

Tijdens deze leerlijn raden we je aan te werken met de IDE **IntelliJ**. IntelliJ is een betaald product, in tegenstelling tot andere bekende gratis IDE's (zoals Visual Studio Code, Atom of Eclipse), maar daar krijg je dan ook een hoop extra functionaliteit voor terug! Dit hoeft je uiteraard niet zelf te bekosten, tijdens je studietijd bij NOVI krijg je de licentie van ons cadeau.

Tijdens deze leerlijn zullen docenten tijdens hun uitleg altijd gebruik maken van deze IDE en we raden je daarom aan hetzelfde te doen. Als je liever een andere IDE wil gebruiken dan mag dat natuurlijk, maar mocht je tegen IDE-gerelateerde verschillen of problemen aanlopen, dan is dat op eigen risico.



Om gebruik te maken van de onderwijs-licentie van NOVI moet je je eerst bij JetBrains (de maker van o.a. IntelliJ) registeren als student. Zorg ervoor dat je dit vóór de eerste les gedaan hebt!

Licentie aanvragen

1. Register je met jouw @novi-education.nl e-mailadres via [dit formulier](#). Als het goed is opent het formulier op het tabje 'University email address'. Dit is het enige tabje dat ingevuld hoeft te worden. Op het formulier vul je het volgende in:
 - a. I'm a student;
 - b. Undergraduate;
 - c. Yes;
 - d. Jouw NOVI e-mailadres;

- e. Voor en achternaam;
f. Netherlands;
g. Vink het vakje aan 'I have read and I accept the JetBrains Account Agreement'
9. Er wordt een account voor je aangemaakt en een verificatie e-mail gestuurd naar jouw @nov-edu adres;
10. Open deze mail en activeer jouw account. Daarna kun je IntelliJ downloaden en installeren via de [account-pagina](#).

1 Student Pack License

[Buy new license](#)

 **JetBrains Product Pack for Students**

[Download ▾](#)

Licensed to: [REDACTED]

License restriction: For educational use only

Valid through: August 23, 2023

Following products included:

• AppCode	• CLion	• DataGrip	• DataSpell	• dotCover
• dotMemory	• dotTrace	• GoLand	• IntelliJ IDEA Ultimate	• PhpStorm
• PyCharm	• ReSharper	• ReSharper C++	• Rider	• RubyMine

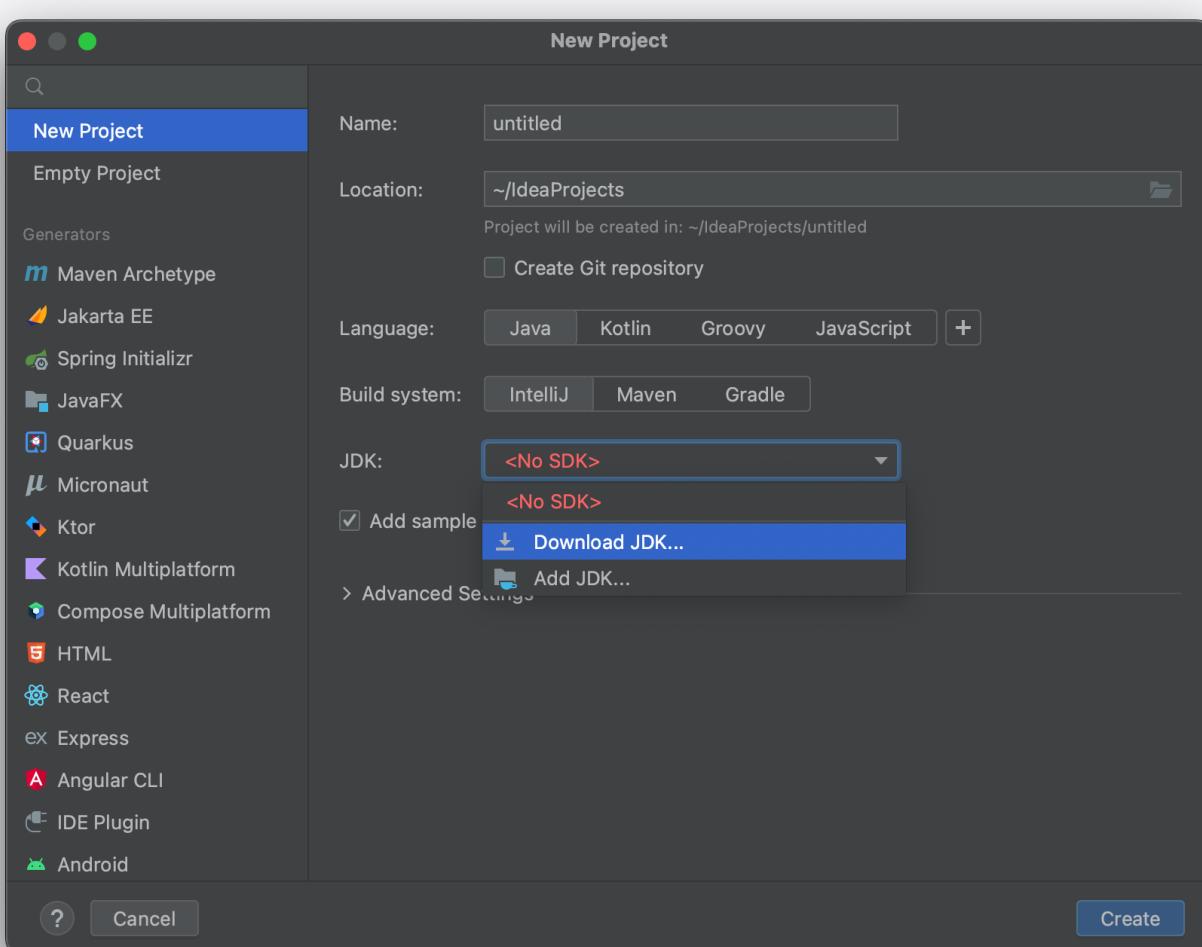
After downloading and installing the software, simply run it and follow the on-screen prompts to sign in with your JetBrains Account.

Zorg ervoor dat je IntelliJ daarna direct activeert door de applicatie na de installatie te [openen](#) en via de bovenste menu-balk op [Help > Register...](#) te klikken. Je kunt dan inloggen met jouw JetBrains account, waardoor de licentie wordt geactiveerd. In het volgende hoofdstuk leer je hoe je IntelliJ kunt gebruiken om een Java programma te draaien.

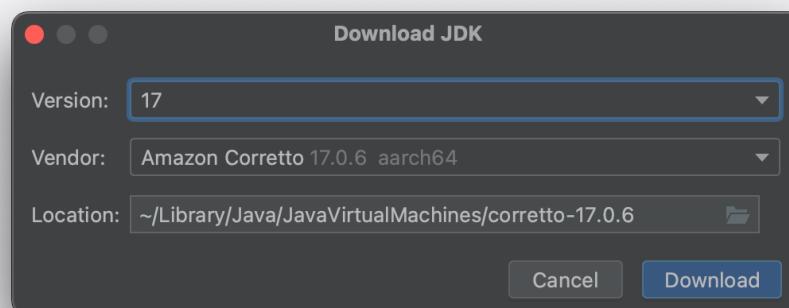
1.5 Installatie JDK

Wanneer je IntelliJ geïnstalleerd hebt is de eerste stap ook genomen in het inrichten van je ontwikkelomgeving. Om Java te gebruiken hebben we natuurlijk nog meer onderdelen nodig, maar IntelliJ helpt je hier graag en gemakkelijk mee op weg.

Om met Java aan de slag te gaan hebben we een Java SDK (Software Development Kit) nodig. Dit noemen we ook wel de JDK. **Java Development Kit**. Dit kunnen we gemakkelijk doen via IntelliJ. Wanneer je IntelliJ geopend hebt, klik je via de bovenste menu-balk op [File > New > Project...](#) te klikken.



1. Geef je project een naam. Bijvoorbeeld `HelloWorld`.
2. Controleer of Java is geselecteerd als taal bij het kopje `Language` en IntelliJ als `Build System`.
3. Bij de `JDK` kan je een `JDK` opgeven die al op je machine geïnstalleerd is, of je kiest voor de optie om IntelliJ deze voor je te laten downloaden en installeren. Selecteer daarvoor de optie `Download JDK...`.
4. Vervolgens opent zich een nieuwe pop-up. Selecteer hier `Version 17` van de Vendor `Amazon Corretto 17.x.x`.



Is je optie `Add sample code` geselecteerd laat staan, wordt er een project aangemaakt die direct klaar is voor gebruik. Zodra IntelliJ klaar is met het downloaden van de `JDK` kan je het gegenereerde programma uitvoeren door in de bovenste menubalk op `Run > Run Main` te drukken. Wat er allemaal gebeurt tijdens het runnen van zo'n programma zullen we uiteraard nog gaan behandelen, maar het resultaat moet het bericht "Hello World!" printen in de terminal.

2. De basis van Java

2.1 Inleiding

Java is een containerbegrip voor de **Java Virtual Machine** (JVM), de **Java programmeertaal** en de **Java API's** (de interne libraries die al voor je zijn geprogrammeerd). Zoals je hebt gelezen, behoort Java tot de gecompileerde talen. Dit betekent dat de code die je schrijft niet direct door de computer kan worden uitgevoerd: Java-code wordt door een compiler omgezet in bytecode, die vervolgens door de JVM uitgevoerd wordt op de computer.

We beginnen bij het begin: Java als programmeertaal. We zullen eerst kijken naar de basis syntax waaruit een Java applicatie is opgebouwd. Van hieruit werken we toe naar de grotere structuren en mechanismen die Java kent, om uiteindelijk grotere applicaties te kunnen bouwen. Let op: sommige van deze grotere structuren zijn noodzakelijk om een Java applicatie te maken, maar om een logische opbouw te garanderen zullen we deze niet direct uitleggen. Neem dus af en toe aan dat iets is zoals het is - de uitleg volgt in een later stadium. Beloofd.

2.2 Rondleiding door IntelliJ

Onderstaande video geeft je een eerste rondleiding door het gebruik van IntelliJ en het runnen van een Java-programma. Het werkt het best wanneer je zelf meedoet met de video, dan heb je direct een oefenproject waarin je de code-voorbeelden van de komende paragrafen kunt uitproberen.

Zie je geen video verschijnen? Bekijk de video [hier](#) in een nieuw tabblad.

2.3 Basis syntax

Hieronder zie je het kleinste mogelijke Java programma dat een instructie uitvoert. Traditioneel is het eerste programma wat je schrijft een programma dat de tekst "Hello world" in de terminal laat zien. Zo weet je gelijk of alles goed geïnstalleerd is en alles goed werkt!

```
// MyClass.java
public class MyClass {
    public static void main (String[] args) {
        System.out.println("Hello World!"); // Prints "Hello World!"
    }
}
```

Wil je deze code uitvoeren?

- Maak een nieuw bestand in de src-map en noem deze `MyClass.java`.
- Zet bovenstaande code in dit bestand.
- Voer `javac MyClass.java` uit;
- Voer `java MyClass` uit;

Java Class

Omdat Java een object georiënteerde taal is, moet alle code geschreven worden binnen een **class**. Wat een class precies is en hoe dit werkt, zullen we later uitgebreid toelichten. Voor nu geven we je alleen de syntax om een class te schrijven:

```
public class <ClassNaam> {  
}
```

Wat die access-modifier `public` daar precies doet, mag je voor nu negeren. Laten we eerst kijken naar de rest van de syntax: het keyword `class` vertelt Java dat er een nieuwe class wordt gedefinieerd. Direct hierna volgt de **naam** van de class. Hierbij is het conventie dat de naam van de class altijd begint met een hoofdletter. Bestaat de naam semantisch uit meerdere woorden, zoals "my first class" of "the best class ever made"? Dan mag je alle opvolgende woorden aan elkaar plakken en starten met een hoofdletter: `MyFirstClass` en `TheBestClassEverMade`. Deze schrijfwijze wordt **CamelCase** genoemd. De naam van een class kan geen spaties bevatten. Na de class naam volgen accolades; daartussen komt de inhoud van de class te staan.

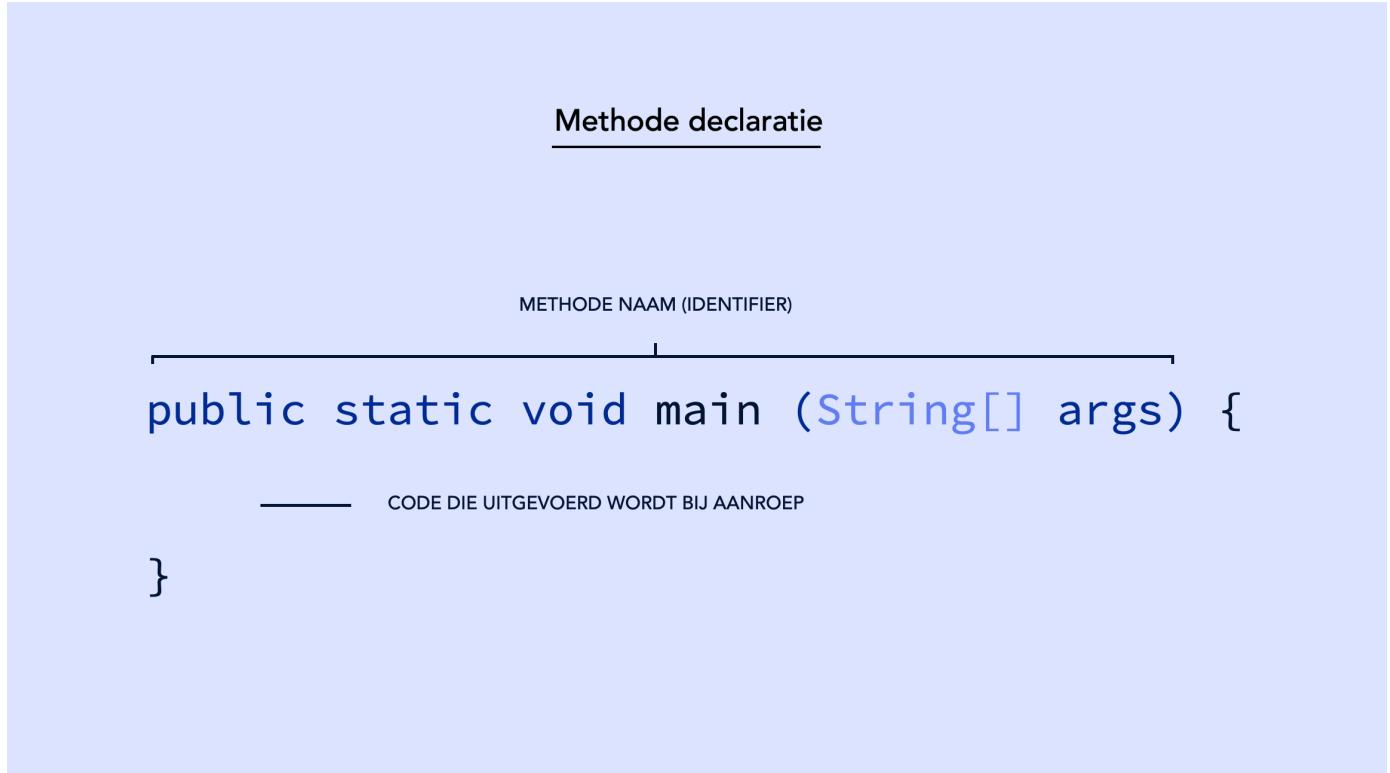
Sidenote: Het maakt voor Java niet uit hoe je de class noemt. `My_Class`, `My_class`, `My_cLaSs` en `myClass` zouden allemaal valide namen zijn voor de JVM. Voor jezelf en je medeontwikkelaars is het echter handig om conventies aan te houden.

Entrypoint

Een applicatie kan uiteindelijk heel veel code code bevatten. Daarom moet het voor Java duidelijk zijn wat het begin, ook wel het **entrypoint** genoemd, van jouw code is. Oftewel: welke instructies moeten als eerst worden uitgevoerd? In Java is dat de zogenaamde `main`-methode. Als in Java een programma start, zoekt het automatisch naar een `main`-methode in het project en zal de code worden uitgevoerd die daar tussen de accolades (...) staat. We noemen zo'n blok code een **method declaration**. Voor nu kun je de eerste regel beschouwen als de naam van een methode.

De syntax om de `main` methode te declareren is als volgt:

```
public static void main (String[] args) {  
}
```



De keywords `public`, `static`, en `void` mag je voor nu even negeren. Geen zorgen: je zult snel genoeg gaan begrijpen wat al deze definities inhouden. Voor nu is het voldoende om deze methode over te nemen en te onthouden dat een Java programma altijd zal starten met het uitvoeren van deze methode. De naam van deze methode is `main`. Hé, daar is dan weer één hoofdletter gebruikt! En dat is de bedoeling: het is conventie om de naam van methoden altijd te beginnen met een kleine letter. Verder gelden dezelfde regels als voor classes.

Wanneer je een Java programma draait, zal alles (en wat "alles" is wordt in de komende hoofdstukken behandeld) tussen de accolades worden uitvoerd. Als het programma bij de sluitende accolade van de `main`-methode komt, is het programma klaar.

Commentaar

Niet alle tekst die we schrijven is bedoeld om als code uit te voeren. Soms willen we bijvoorbeeld een beschrijving toevoegen aan een regel code zodat deze makkelijker te begrijpen is: dit noemen we **commentaar**. We kunnen commentaar toevoegen door een regel met `//` te beginnen. Alles wat hierna komt zal dus niet als code worden uitgevoerd. Hier zien we bijvoorbeeld dat de tekst `Prints "Hello World!"` geen code is, maar commentaar omdat er `//` voor staat.

```
// MyClass.java
public class MyClass {
    public static void main (String[] args) {
        System.out.println("Hello World!"); // Prints "Hello World!"
    }
}
```

Het is ook mogelijk om meerdere regels uit-commentariëren door er `/* */` omheen te zetten:

```
// MyClass.java
public class MyClass {
    /* Dit is een belangrijke methode, onder andere omdat dit
     * het startpunt is van iedere Java applicatie! */
    public static void main (String[] args) {
        System.out.println("Hello World!");
    }
}
```

Het is echter zonde van de tijd om al deze combinaties uit je hoofd te leren. Je kunt regels code gemakkelijk in- en uit commentaar plaatsen door de regel te selecteren en daarna op `CTRL + /` (Windows) of `CMD + /` (Mac) te drukken. Dat gaat een stuk sneller!

2.4 Variabelen

Laten we eens gaan kijken naar de basis van iedere programmeertaal: het opslaan van stukjes data. Binnen Java kan je werken met zowel **statements** als **expressies**. Een **statement** is een regel die iets moet doen of uitvoeren, zoals bijvoorbeeld het tonen van een tekst in de terminal:

```
System.out.println("Java is te gek!");
```

Een **expressie** is een regel die iets berekent en teruggeeft, zoals bijvoorbeeld het optellen van twee getallen:

```
1 + 1;
```

Wat hierbij belangrijk is, is dat de expressie `1 + 1` pas 2 wordt *nadat* deze regel code uitgevoerd is. Het uitvoeren van een expressie noemten we **evalueren**. Het evalueren van de expressie `1 + 1` geeft als uitkomst 2. En zo kan de expressie `5 + 10 - 8` geëvalueerd worden naar 7. Expressies kunnen dus heel groot worden, maar dat hoeft niet. De meest simpele expressie is namelijk gewoon een getal! Het getal 8 is ook een expressie die evalueert naar... 8. Naast getallen zijn er nog veel meer vormen van expressies, zoals bijvoorbeeld tekst. Herken je de `main`-methode uit het vorige hoofdstuk nog?

```
// MyClass.java
public class MyClass {
    public static void main (String[] args) {
        System.out.println("Hello World!");
    }
}
```

Hier zie je dat de statement `System.out.println` dicteert dat er een tekst op het scherm moet verschijnen. De tekst zelf, "Hello World!" is dan weer een expressie.

Wanneer we expressies schrijven, is het belangrijk dat we de uitkomsten daarvan ergens opslaan. Doen we dat niet? Dan wordt het lastig om een werkend stukje software te schrijven... De evaluaties van onze expressies "verdwijnen" dan gewoon. Als we niet expliciet vermelden dat deze uitkomsten onthouden moeten worden, kan het programma ze daarna niet meer gebruiken. We slaan deze waarden daarom op in variabelen: een soort containerretjes om het resultaat in het werkgeheugen (RAM) van de computer op te slaan.

Variabelen als denkbeeldige containers

totalPrice

34

productName

"Voetbal"

hasPaid

false

Je hebt verschillende typen variabelen binnen Java. In dit hoofdstuk behandelen en gebruiken we vooral primitieve types:

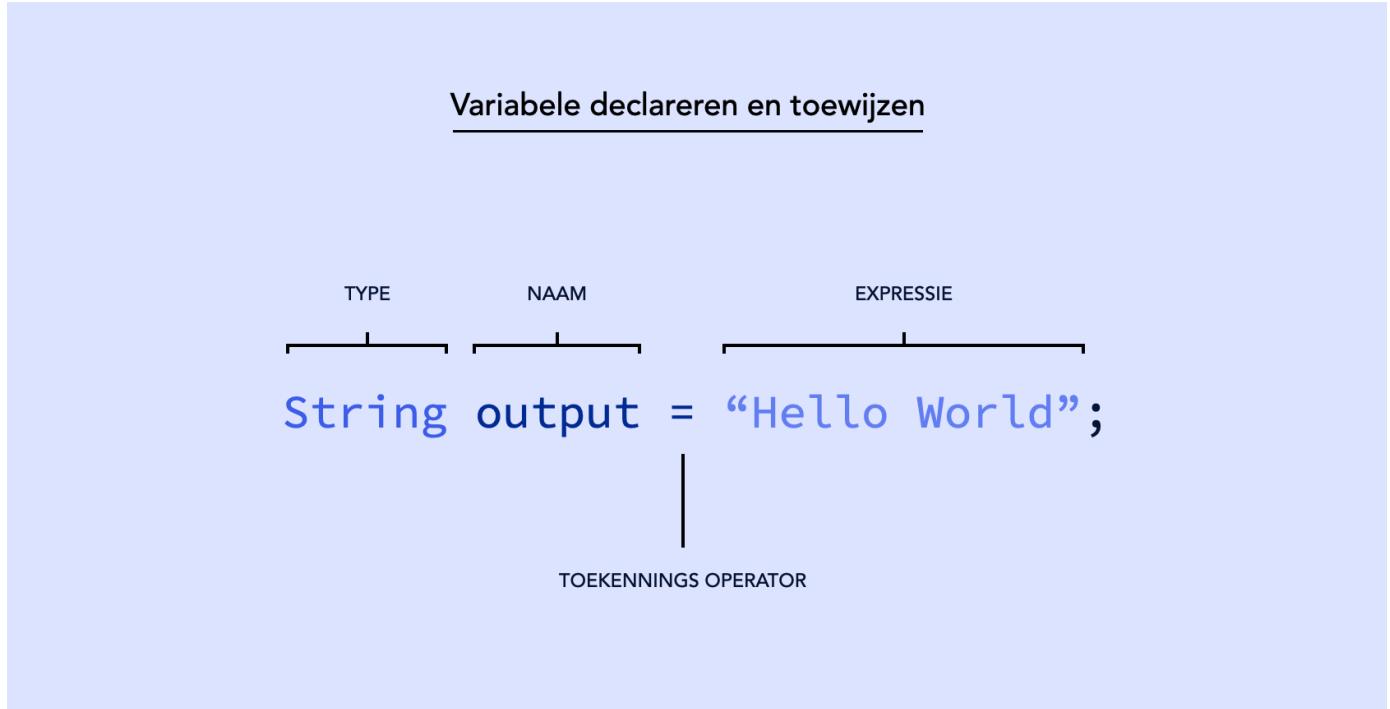
- int
- float
- double
- byte
- short
- boolean

Daarnaast heb je in Java ook object types, zoals `String`. Hoe je die gebruikt, vertellen we later. Voor nu mag je aannemen dat een `String` enkele regels tekst kan bewaren.

Declareren en toewijzen

Wanneer we een variabele aanmaken, noemen we dit **declareren**. We vertellen Java dat we van plan zijn om een `String` op te slaan en hoe we de "container" willen noemen. Wanneer we daar ook daadwerkelijk een waarde (resultaat van een expressie) in plaatsen, noemen we dit **toewijzen**. Het toewijzen van een waarde kan vaker dan één keer gebeuren: we mogen de inhoud van de container zo vaak veranderen als we willen. Laten we ons programma eens omschrijven naar een situatie waarin we een variabele declareren en toewijzen:

```
// MyClass.java
public class MyClass {
    public static void main (String[] args) {
        String output;
        output = "Hello World!";
        System.out.println(output);
    }
}
```



Zie je geen video verschijnen? Bekijk de video [hier](#) in een nieuw tabblad.

2.5 Variabel types

Naast `Strings` - waar we woorden en zinnen in kunnen opslaan - gebruiken we binnen Java ook `booleans` en `getallen`: ronde getallen, komma-getallen, kleine getallen en hele grote getallen. Hoe je deze types gebruikt leggen we je in deze paragraaf uit.

boolean

Een `boolean` is een variabele die slechts twee mogelijke waarden kent, namelijk `true` (waar) en `false` (niet waar). Je kunt dit vergelijken met aan of uit en 1 of 0. Alhoewel de `boolean` op zichzelf niet heel nuttig lijkt, is dit datatype heel belangrijk voor het toepassen van logica in Java. Een `boolean` declarer je als volgt:

```
boolean isComplete = false;
```

In dit geval maken we een container waar een `boolean`-waarde in past met de naam `isComplete` en wijzen de waarde `"false"` toe. Het is conveniente om variabelen die een `boolean` bevatten een naam te geven die begint met `"is"` of `"should"`. Ook is het gebruikelijk de benaming altijd volgens het format `is <Positive>` te beschrijven, om te voorkomen dat we dubbele ontkenningen gebruiken.

Een variabele genaamd `isNotComplete` is dus niet zo handig. Probeer `isNotComplete = true;` zelf maar eens te interpreteren, is de taak nou compleet of niet? Je merkt het al: deze expressie is veel moeizamer te begrijpen dan `isComplete = false;`

Sidenote: Sommige talen behandelen daadwerkelijk getallen als booleans. Vaak is `'0'` dan `false` en andere getallen zijn `true`. Java kent echter alleen de keywords `true` en `false` als geldige waarden voor een boolean.

Getallen

Het klinkt misschien gek, maar het opslaan van getallen is complexer - en veel belangrijker! - dan je op het eerste gezicht kunt vermoeden. Je zult zien dat je naarmate je grotere applicaties gaat schrijven, getallen daar een hele grote rol in spelen. De manier waarop je jouw getallen opslaat heeft dus een groot effect op hoe efficiënt jouw applicatie met het geheugen omgaat!

Alleen al voor het opslaan van **integers** (hele getallen) heb je verschillende dingen om rekening mee te houden. Java kent voor het opslaan van getallen alleen **signed types**: getallen voorzien van een `+`- of `-`-teken. Alle getallen die we gebruiken kunnen dus zowel negatief als positief zijn. Ook moet je nadenken over de maximale en minimale waarden die je erin wil opslaan. Java kent diverse types om getallen in op te slaan - elk met een ander aantal bits! - die iets zeggen over hoe groot de inhoud maximaal mag zijn.

Datatype	Number of bits	min-value	max-value
byte	8 bits	-128	127
short	16 bits	-32768	32.767
int	32 bits	-2147483648	2.147.483.647
long	64 bits	-9223372036854775808	9.223.372.036.854.775.807
float	32 bits	1.40239846e-45	3.40282347e+38
double	64 bits	4.94065645841246544e-324	1.79769313486231570e+308

Twee datatypes die wel in dit lijstje thuis horen, maar niet bedoeld zijn om getallen in op te slaan, zijn boolean en char.

Datatype	Number of bits	min-value	max-value
boolean	1 bit	0	1
char	16 bits	0	65535

Deze datatypes zijn **unsigned**; je kunt hier geen negatieve getallen in opslaan, omdat je er geen + of - teken voor mag zetten. Waar char voor gebruikt wordt, zullen we later behandelen. Voor nu houden we het even bij onze wiskundige integers.

Grapig genoeg zie je bij boolean 0 en 1 staan, terwijl we zojuist hebben uitgelegd dat dit in Java alleen true of false mag zijn. Als dit stukje je nieuwsgierigheid wekt, kun je onderstaande bonusopslag over opslag doornemen. De **TLDR** (Too Long Didn't Read) versie van het antwoord is het volgende: bij het schrijven van Java code, dient Java dat een boolean-variabele de waarden true en false mag hebben. In het geheugen wordt dit echter binair opgeslagen. Voor een boolean is maar 1 bit nodig die uit of aan kan staan. Dit wordt over het algemeen weergegeven met 0 of 1. De tabel hierboven gaat over de opslag in het geheugen. Eigenlijk is de minimale waarde van een char 0000000000000000 en de maximale waarde 1111111111111111. In de tabel zijn de getal-waarden van de bijbehorende binaire representatie weergegeven.

*Bonus: Uitleg over opslag

Stel je voor dat we een getallenreeks hebben die start bij 1 en telkens verdubbelt wordt:

1,2,4,8,16,32....

Nu draaien we de volgorde van deze getallen draaien om:32,16,8,4,2,1

We noemen elk van deze getallen een bit en we gaan uit van 5 bits per reeks. Stel nu dat we ieder getal aan kunnen zetten met een 1 of uit kunnen zetten met een 0.

16,8,4,2,1

0,0,0,0,0

0+0+0+0+0 = 0 is het kleinste getal dat we kunnen maken.

16,8,4,2,1

1,1,1,1,1

1+2+4+8+16 = 31 is het grootste getal dat we kunnen maken.

Zo kunnen we elke getal tussen de 0 en 31 uitdrukken in vijf nullen en éénen.

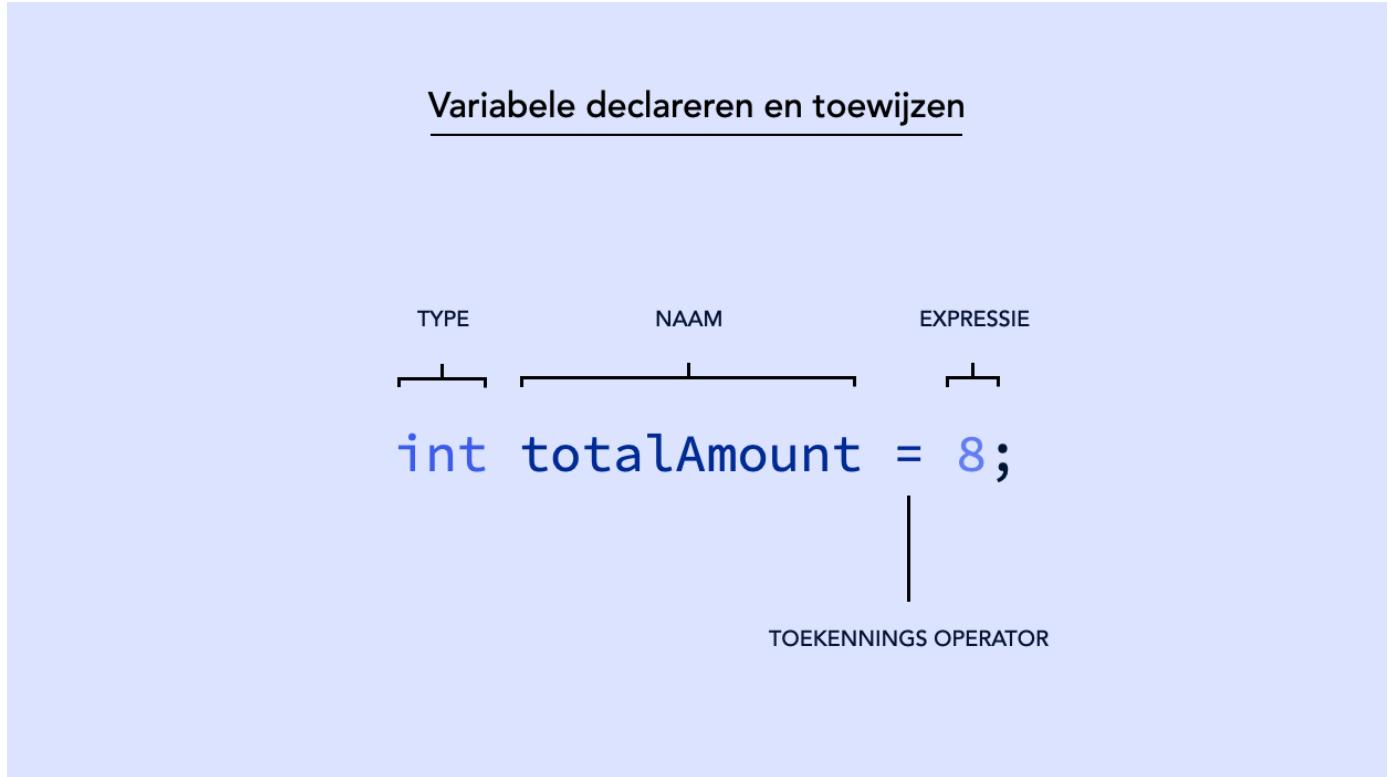
Als we ook negatieve getallen willen opslaan hebben we 1 "bit" nodig om op te slaan of het getal "negatief" of "positief" is. In 5 bits kunnen we dan -16 t/m 15 opslaan. 1 bit voor de sign en 4 bits voor de getallen 0 tot 15.

Aangezien het hebben van 0 en -0 niet zo veel zin heeft, worden de negatieve getallen nog iets opgeschoven naar -1 tot -16.

Als je wilt weten hoe negatieve getallen echt opgeslagen worden, kun je je verdiepen in een principe dat we "2's compliment" noemen. Dit valt echter buiten de scope van deze leerlijn.

int

Voor het opslaan van hele getallen wordt int veruit het meest gebruikt. Types als byte en short zie je minder vaak. De long wordt veel gebruikt voor getallen die groter kunnen zijn - of worden! - dan de maximale range van een int. De declaratie en toewijzing van gehele getallen heeft de volgende syntax:



Het kiezen van het juiste type is belangrijk. Gelukkig loop je niet zo snel tegen issues aan wanneer je `int` en `long` gebruikt. Het is echter wel handig om te weten wat het gedrag van Java is als het niet meer past. We nemen het volgende voorbeeld:

```
public class MyClass {
    public static void main(String[] args) {
        int myNumber = 2147483647;
        myNumber = myNumber + 1;
        System.out.println(myNumber); // prints -2147483648
    }
}
```

We declareren een `int`-variabele en noemen deze `myNumber`. We wijzen ook direct een waarde toe, namelijk de maximale waarde van een `int`. Daarna kennen we weer een nieuwe waarde toe, namelijk de oude waarde + 1. De code print nu, in plaats van de daadwerkelijke uitkomst, de minimale waarde van een `int`. Dit wordt ook wel `Integer overflow` genoemd.

Bonus uitleg: Hoe werkt een Integer overflow?
Stel we hebben een 5 bits signed getal, waarbij 1111 de maximale (positieve) waarde is en 00000 de maximale negatieve waarde. We wijzen nu de maximale waarde toe en tellen er 1 bij op:

1111
00001+
00000

We hebben nu 6 bits met helemaal links een 1, maar omdat het getal maar 5 bits kan bevatten, valt die 1 er af. Daardoor blijft 00000 over, het minimale getal.

Kommagetallen

De `float` en `double` worden gebruikt voor kommagettallen. Houd er rekening mee dat beide types niet 100% accuraat zijn. Je kunt er grofweg van uitgaan dat een `float` tot zeven getallen achter de komma precies is en een `double` tot 14 getallen achter de komma. Als je een kommagetal wil gebruiken, is dit standaard een `double`:

```
public class MyClass {
    public static void main(String[] args) {
        double grade = 2.0;
    }
}
```

Om een `float` te definiëren moet je een `f` achter het getal zetten. Dit noem je een `postfix`:

```
public class MyClass {
    public static void main(String[] args) {
        float exactGradeWrong = 2.0; // dit gaat fout
        float exactGradeRight = 2.0f; // dit gaat goed
    }
}
```

Ook de `long` en de `double` hebben een postfix. Deze is vrijwel nooit nodig, maar mag je er altijd achter zetten.

```
public class MyClass {
    public static void main(String[] args) {
        short valueA = 1;
        int valueB = 1;
        long valueC = 1L; // de postfix voor een long is "L"
        float valueD = 2.0f;
        double valueE = 2.0d; // de postfix voor een double is "d"
    }
}
```

Java heeft dus, net als veel andere programmeertalen, verschillende typen variabelen. Op deze manier kunnen we correct omgaan met de verschillende typen data die we in een programma nodig hebben. Daarnaast biedt het veiligheid: door zeer specifiek te zijn in de types die je gebruikt heb je minder kans om per ongeluk bugs in je code te implementeren. Door het juiste type variabele te gebruiken, kun je dus optimaal profiteren van alle functionaliteit die Java te bieden heeft!

2.6 Opdracht: variabelen maken

De ontwikkelaar van dit Java-programma is halverwege het programmeren een broodje pindakaas gaan smeren en kwam daarna nooit meer terug. Wanneer je de code probeert te runnen, zal de compiler een lijst aan foutmeldingen laten zien. Daarom hebben we jouw hulp nodig om de code af te maken!

In de Main-class staan een hoop System.out.println-statementen, maar alle variabelen die geprint moeten worden zijn rood gekleurd. Dat komt omdat deze variabelen nog niet zijn gedeclareerd. Gelukkig heeft de ontwikkelaar wat hints achtergelaten en een aantal testen geschreven voor dit programma die je helpen de verloren variabelen te declareren.

Je kunt deze opdracht maken door het project te clonen of te downloaden naar jouw eigen computer via [deze GitHub repository](#). Hier vind je ook de gedetailleerde instructies van de opdracht zelf. De uitwerkingen staan op de branch `uitwerkingen`.

Hoe open ik dit project?

Als je nog niet weet hoe je gebruik maakt van Git, kun je het project downloaden als `.zip` bestand door op de grote groene knop te klikken:

Code Internal
generated from [hogeschoolnovi/backend-leerlijn-template](#)

Code Issues Pull requests Actions Projects Wiki Security Insights Settings

main ▾ 3 branches 0 tags Go to file Add file ▾ **Code**

Your main branch isn't protected

Protect this branch from force pushing or deletion, or require status checks before merging.

Clone Local Codespaces (New)

HTTPS SSH GitHub CLI

git@github.com:hogeschoolnovi/backend-jav

Use a password-protected SSH key.

Open with GitHub Desktop

Download ZIP

5 days ago

5 days ago

41 minutes ago

File	Description	Last Commit
<code>.mvn/wrapper</code>	Initial commit	5 days ago
<code>src</code>	Variabele	5 days ago
<code>.gitignore</code>	Initial commit	5 days ago
<code>mvnw</code>	Initial commit	5 days ago
<code>mvnw.cmd</code>	Initial commit	5 days ago
<code>pom.xml</code>	Variabele	5 days ago
<code>readme.md</code>	Update readme.md	41 minutes ago

Wanneer het project is gedownload, verschijnt deze in jouw lokale map `Downloads`. Pak de gecomprimeerde map eerst uit. Daarna kun je het project in IntelliJ openen via `File > Open...`, waarna je de gehele projectmap selecteert.

2.7 Operatoren

Nu we waardes kunnen opslaan, willen we er natuurlijk ook dingen mee doen. Hiervoor kent Java **operatoren**. Eigenlijk heb je daar al mee gewerkt, want bij het toewijzen van waarden aan variabelen heb je immers de toekennings operator = gebruikt!

```
int totalAmount = 8;
```

Operatoren zijn te verdelen in een aantal categorieën:

- De wiskundige operatoren;
- Toekennings operatoren;
- Relatievele operatoren;
- Logica operatoren;
- Bitwise operatoren*

*De bitwise operatoren zul je in de praktijk weinig tegenkomen en hebben alleen voor performance grote toegevoegde waarde. Daarom zullen we deze niet verder behandelen.

Onderstaande video neemt je mee in het gebruik van een aantal standaard operatoren. Daarna zul je je in deze paragraaf verder verdiepen in de grote verscheidenheid aan operatoren die Java biedt.

Zie je geen video verschijnen? Bekijk de video [hier](#) in een nieuw tabblad.

Wiskundige operatoren

Java kent vijf wiskundige operatoren die je waarschijnlijk bekend voorkomen. We gebruiken ze voor optellen, aftrekken, delen en vermenigvuldigen. De operatoren opereren steeds op twee getallen die we de **operands** noemen, waarbij de operator tussen de getallen staat. De operands vormen samen met de operator een **expressie**:

Met de + operator kun je twee getallen optellen. $2 + 8 //$ geeft 10
 Met de - operator kun je een getal van een ander getal aftrekken. $1 - 3 //$ geeft -2
 Met de / operator kun je een getal door een ander getal delen. $4 / 2 //$ geeft 2
 Met de * operator kun je 2 getallen met elkaar vermenigvuldigen. $2 * 3 //$ geeft 6
 Met de % operator kun je de modulo van een getal met een ander getal bepalen. $8 \% 3 //$ geeft 2

Een operatator die je waarschijnlijk nog niet kende, is **modulo**. Deze operatator berekent wat er **overblijft** als je getallen door elkaar probeert te delen. Dat is bijvoorbeeld handig wanneer je wil checken of een getal **even**, of **oneven** is. Wanneer je een even getal deelt door 2, blijft er namelijk niets over. Kijk maar:

```
20 % 10 // geeft 0, want:  

// tien kunnen we 2x uit twintig halen, en vervolgens blijft er niets over.  

23 % 10 // geeft 3, want:  

// tien kunnen we nog steeds 2x uit twintig halen, maar er blijft dan nog 3 over.
```

Je kunt er bij wiskundige operatoren vanuit gaan dat hier dezelfde regels gelden als in de normale wiskunde:

1. Vermenigvuldigen gaat altijd voor op optellen of aftrekken.
2. Om specifieke delen van de expressie voorrang te geven, kun je ronde haken gebruiken om sub-expressies te maken.

```
int x = 1 + 2 * 3; // geeft 7
int y = (1 + 2) * 3; // geeft 9
```

In het eerste voorbeeld gelden de normale regels. De vermenigvuldiging $2 * 3$ wordt eerst geëvalueerd omdat de `*` operator voorrang heeft op de `+` operator. Vervolgens wordt er 1 bij opgeteld. In het tweede voorbeeld wordt eerst de sub-expressie $1 + 2$, geëvalueerd. Ten slotte blijft de expressie $3 * 3$ over.

Ten slotte is het ook belangrijk om goed op het datatype van deze operanden (`byte`, `int`, etc.) te letten, omdat dit invloed heeft op de uitkomst. Zolang we met hele getallen werken, gelden de volgende regels:

1. Het resultaat van een expressie is een `int`, als beide operanden `int`'s of kleiner zijn;
2. Als minimaal één van de operanden een `long` is, dan is het resultaat van de expressie ook een `long`.

Datatypes optellen

Als beide operanden een `int` (of kleiner) zijn is het resultaat een `int`

<code>byte + byte = int</code>	<code>short + short = int</code>
<code>byte + short = int</code>	<code>short + int = int</code>
<code>byte + int = int</code>	<code>int + int = int</code>

Bij de `/` operator zijn er door deze regels extra bijzonderheden om rekening mee te houden. Stel dat we twee `int`'s door elkaar delen in de expressie $8 / 3$. Wat denk je dat dan het resultaat zal zijn?

Oké, wiskundig gezien is de uitkomst $2,666667$. Maar binnen Java resulteert dit in de `int`?! Om het kommagetal te kunnen berekenen, zullen we minimaal van één van de operanden een `float` of een `double` moeten maken. De expressies $8 / 3f$, $8f / 3$ en $8f / 3f$ resulteren wel allemaal in $2,6666667$. Indien expressies zowel een `float` als een `double` bevatten, resulteren ze in een `double`.

Oké kent de `/` operator een foutscenario die de overige wiskundige operatoren niet kennen wanneer je probeert te delen door 0. Dat komt omdat je wiskundig gezien niet kunt delen door 0. Nul is immers niets. Wanneer je dit toch probeert te doen, zal dit resulteren in een **Exception**: een foutconstructie in Java om om te gaan met ongeldige zaken, zoals hier:

```
public class MyClass {
    public static void main(String[] args) {
        int x = 6;
        int y = 0;
        int z = x / y; // delen door 0
    }
}
```

Hoe Exceptions werken, behandelen we later in deze cursus.

Toekenningsoperatoren

Het toekennen van waardes aan een variabele is op zichzelf al een operator, namelijk `=`. Deze operator hebben we natuurlijk al eerder gezien. De `=` operator opereert op zowel een variabele als op een expressie:

```
int x = 4;
int x = 4 + 6;
```

Naast de standaard toekenningsoperator `=` zijn er nog enkele bijzondere varianten die het opschrijven van wiskundig operatoren kunnen verkorten. Zo is de `+=` operator een verkorte schrijfwijze voor toewijzen en optellen. Kijk maar eens naar het volgende voorbeeld, waar twee getallen worden opgeteld:

```
public class MyClass {
    public static void main(String[] args) {
        int amount = 6; // Declareer de variabele amount en wijs de waarde 6 toe
        amount = amount + 3; // Wijs het resultaat van de expressie amount + 3 toe aan variabele amount (die evalueert naar 6 + 3 die evalueert naar 9)
        System.out.println(amount); // print 9
    }
}
```

De regel `amount = amount + 3` kunnen we in Java verkort opschrijven als `amount += 3`. Onderstaand voorbeeld doet dus precies hetzelfde als het voorbeeld hierboven:

```
public class MyClass {
```

```
public static void main(String[] args) {
    int amount = 6;
    amount += 3;
    System.out.println(amount);
}
```

Dezelfde regels gelden ook voor `-=`, `*=`, `/=` en `%=`.

Twee andere operaties die vaak voorkomen zijn het optellen met 1 (ook wel **incrementeren** genoemd) en het aftrekken van 1 (ook wel **decrementeren** genoemd).

Hier voor kent Java de operatoren `++` en `--`. Deze operatoren pas je toe op numerieke variabelen, om er 1 bij op te tellen of 1 vanaf te trekken.
Je kunt de operator voor- of achter de variable zetten, en dit heeft een belangrijk subtel verschil in gedrag. Dit gedrag is het beste uit te drukken in zinnen:

- `x++` betekent "Haal eerst de waarde van `x` op, en doe **daarna** `x = x + 1`".
- `x--` betekent "Haal eerst de waarde van `x` op, en doe **daarna** `x = x - 1`".
- `++x` betekent "Doe eerst `x = x + 1`, en haal **daarna** de waarde van `x` op".
- `--x` betekent "Doe eerst `x = x - 1`, en haal **daarna** de waarde van `x` op".

In code heeft dit de volgende implicatie:

```
public class MyClass {
    public static void main(String[] args) {
        int x = 0;
        int y = x++; // x = 1, y = 0

        int q = 0;
        int r = ++q; // q = 1, r = 1
    }
}
```

Relationale Operatoren

We gebruiken relationele operatoren om - je raadt het niet! - de relatie tussen twee data-items te bepalen. Dit klinkt misschien wat cryptisch, maar je kunt een expressie met een relationele operator lezen als een vraag met ja of nee als mogelijk antwoord. Of, om in programmeertermen te blijven, een true of false antwoord. Het resultaat van een relationele operator is daarom altijd een boolean.

De relationele operatoren zijn `==`, `!=`, `<`, `>`, `<=`, `>=`. Wat deze operatoren precies vragen, leggen we uit aan de hand van een aantal voorbeelden.
Laten we beginnen bij de eerste. `==` test op equality (gelijkwaardigheid). Als twee waarden gelijk zijn is het resultaat true, in andere gevallen is het false:

```
public class MyClass {
    public static void main(String[] args) {
        int x = 4;
        boolean b1 = 2 == 2; //true
        boolean b2 = x == 4; //true
        boolean b3 = x - 2 == 2; //true
        boolean b4 = x == 8; //false
    }
}
```

`De operator != test juist op inequality (ongelijkwaardigheid). Als twee waarden ongelijk zijn, is het resultaat true, anders is het resultaat false:`

```
public class MyClass {
    public static void main(String[] args) {
        int x = 4;
        boolean b1 = 2 != 2; //false
        boolean b2 = x != 4; //false
        boolean b3 = x - 2 != 2; //false
        boolean b4 = x; != 8 //true
    }
}
```

- `<` (kleiner dan) test of de linker waarde kleiner is dan de rechter waarde.
- `>` (groter dan) test of de linker waarde groter is dan de rechter waarde.
- `<=` (kleiner of gelijk aan) test of de linker waarde **kleiner** is dan, of **gelijk is** aan de rechter waarde.
- `>=` (groter of gelijk aan) test of de linker waarde **groter** is dan of **gelijk is** aan de rechter waarde.

Laten we dit ook eens in een voorbeeld bekijken:

```
public class MyClass {
    public static void main(String[] args) {
        int x = 4;
        boolean b1 = x < 2; //false
        boolean b2 = x < 4; //false
        boolean b3 = x < 8; //true

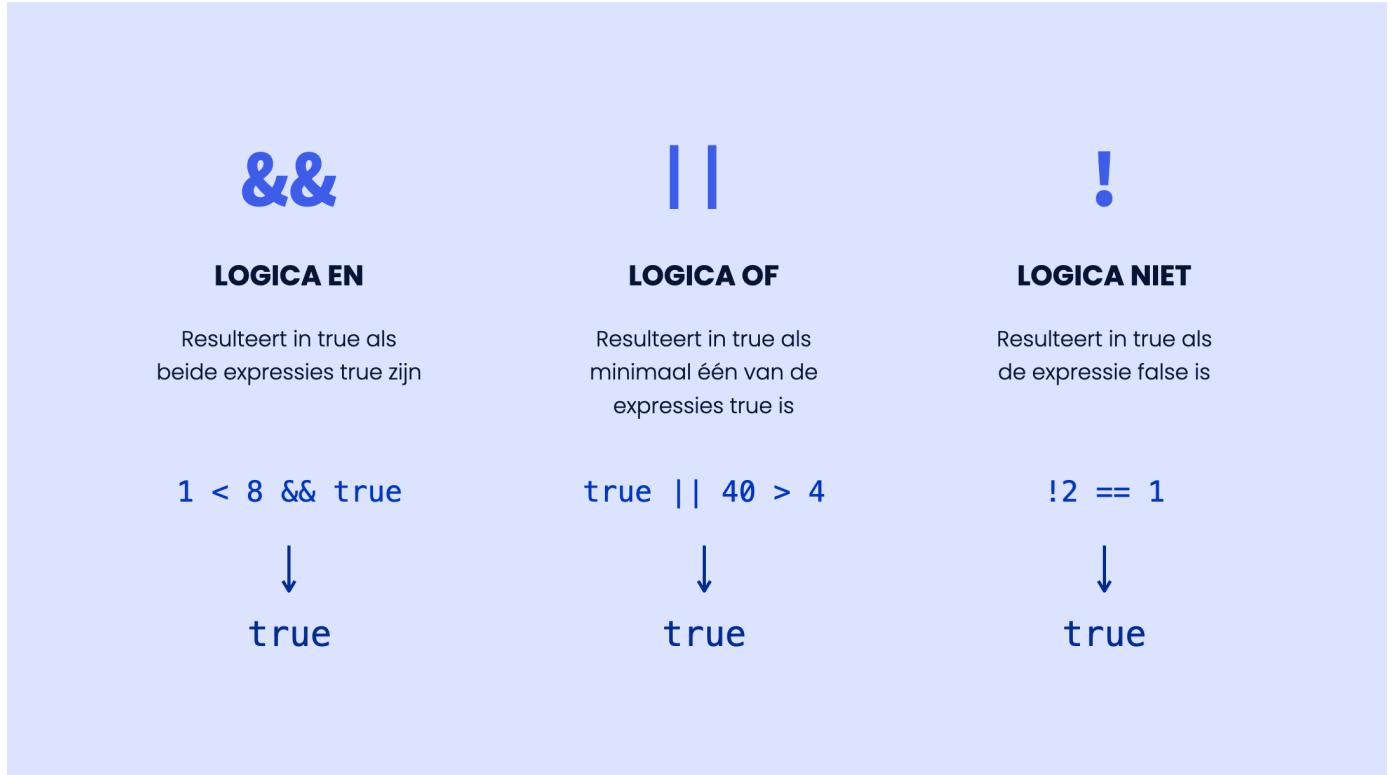
        boolean b4 = x > 2; //true
        boolean b5 = x > 4; //false
        boolean b6 = x > 8; //false

        boolean b7 = x <= 2; //false
        boolean b8 = x <= 4; //true
        boolean b9 = x <= 8; //true

        boolean b10 = x >= 2; //true
        boolean b11 = x >= 4; //true
        boolean b12 = x >= 8; //false
    }
}
```

Logica operatoren

Met logica operatoren kun je meerdere boolean expressies combineren. Het resultaat van een expressie met een logica operator is zelf opnieuw een boolean.



Met de EN-operator (`&&`) kun je aangeven dat *zowel* de expressie links als de expressie rechts waar moet zijn.

```
boolean b1 = true && true; //true
boolean b2 = true && false; //false
boolean b3 = false && true; //false
boolean b4 = false && false; //false
```

Er is iets belangrijks dat je moet weten over logica operatoren... Ze zijn erg lui. Ze evalueren van links naar rechts en stoppen er ook zo snel mogelijk mee (short-circuit). Neem bovenstaand voorbeeld. Bij het gebruik van de `&&` operator moeten *beide* condities waar zijn. Als de operator bij `b3` bij het linker deel van vergelijking al een false ontdekt, heeft het geen zin om de volgende vergelijking nog te checken. De operator houdt het voor gezien.

Met de OF operator (`||`) kun je aangeven dat de expressie links of de expressie rechts waar moet zijn.

```
boolean b1 = true || true; //true
boolean b2 = true || false; //true
boolean b3 = false || true; //true
boolean b4 = false || false; //false
```

En inderdaad, ook deze operatoren zijn lui. Minimaal één van de condities moet bij het gebruik van `||` waar zijn. Zodra er één `true` gevonden wordt, stopt ook deze operator met evalueren. Het maakt immers niet meer uit wat de uitkomst van de tweede expressie is! Het minimum is één en de operator is dan tevreden. Waarom dit belangrijk is? De manier van evalueren heeft uiteindelijk invloed op de uitkomst! Onderstaand voorbeeld is wellicht wat abstract, maar laten we deze samen eens evalueren:

```
public class MyClass {
    public static void main(String[] args) {
        int x = 0;
        boolean b = false && x++ == 0;
        System.out.println(b);
        System.out.println(x);
    }
}
```

Tip: probeer eerst zelf eens te bereceneren wat de uitkomst is van `x` en `b` voordat je kijkt naar onderstaande uitleg!

De uitkomst van `x`

Laten we eens op een rijtje zetten wat er gebeurt:

1. In de main methode van `myClass` wordt als eerste de `int` variabele `x` gedefinieerd. Daar wordt direct de waarde 0 aan toegewezen.
2. We declareren de `boolean` variabele `b` en wijzen daar het resultaat van de expressie `false && x++ == 0` aan toe.
3. De expressie wordt geëvalueerd van links naar rechts. Eerst komen we een `false` tegen, en daarna een `&&`.
4. De `&&` operator kan alleen resulteren in `true` indien *beide* waarden waar zijn. Echter, na de eerste `false` is het resultaat al bekend. De rechter expressie (`x++ == 0`) wordt niet uitgevoerd.
5. `System.out.println(b)` print `false`.
6. `System.out.println(x)` print 0.

Laten we nu kijken wat er gebeurt als de linkerkant van de `&&` operator `true` zou zijn geweest. Wat denk jij dat hier de uitkomst van `x` en `b` zijn?

```
public class MyClass {
    public static void main(String[] args) {
        int x = 0;
        boolean b = true && x++ == 0;
        System.out.println(b);
        System.out.println(x);
    }
}
```

1. In de main methode van `myClass` wordt als eerste de `int` variabele `x` gedefinieerd. Daar wordt direct de waarde 0 aan toegewezen.
2. We declareren de `boolean` variabele `b` en wijzen daar het resultaat van de expressie `false && x++ == 0` aan toe.
3. De expressie wordt geëvalueerd van links naar rechts. Eerst komen we een `true` tegen, en daarna een `&&`.

4. De `&&` operator kan alleen resulteren in `true` indien *beide* waarden waar zijn. Daarom zullen we de tweede expressie (`x++ == 0`) ook moeten evalueren.
 5. `x++` betekent "Haal eerst de waarde van `x` op, doe daarna `x = x + 1`". De expressie betekent dus eigenlijk: "`x == 0` en daarna `x = x + 1`". Aangezien `x` bij het uitvoeren van de expressie nog `0` is, evalueert de rechter expressie naar `true`. `System.out.println(b)` print `true`. `System.out.println(x)` print `1`.

Laat bovenstaande voorbeelden goed op je inwerken. Het is belangrijk om deze theorie goed te begrijpen. Als je deze voorbeelden goed kunt volgen, heb je een aardig beeld van het gebruik van operatoren!

2.8 Opdracht: operatoren

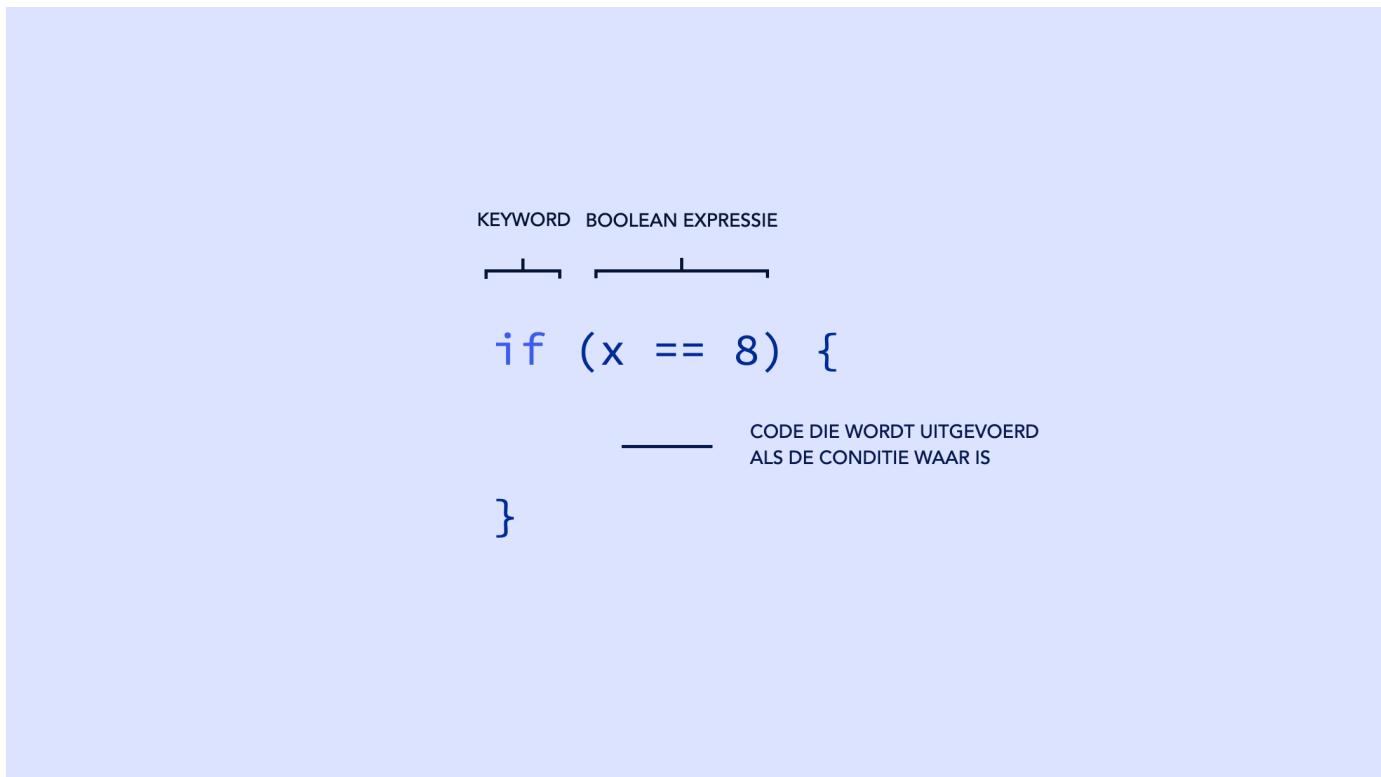
Je gaat operators gebruiken om waarden te muteren! Je zult waarden gaan optellen, aftrekken, door elkaar delen en met elkaar vergelijken. In de `Main`-class zul je een hoop declaraties en toewijzingen vinden. Welke operators kunnen we gebruiken om alles te laten kloppen?

Je kunt deze opdracht maken door het project te clonen of te downloaden naar jouw eigen computer via [deze GitHub repository](#). Hier vind je ook de gedetailleerde instructies van de opdracht zelf. De uitwerkingen staan op de branch `uitwerkingen`.

3. Controlflow

3.1 If-statements

Alle code die we tot nu toe hebben geschreven was vrij... rechtlijnig. De statements werden netjes in volgorde van boven naar beneden uitgevoerd. Er zijn echter een hoop statements die de flow van het programma kunnen aanpassen. Deze statements kun je grofweg onderverdelen in drie groepen: **beslissingsstatements**, **loopstatements** en **jumpstatements**. Beslissingsstatements kun je ook weer onderverdelen in twee groepen: **if-statements** en **switch-statements**. Switch-statements zullen we aan het eind van dit hoofdstuk behandelen. Eerst gaan we in op het gebruik van if-statements:



Zoals je ziet bestaat een if-statement niet uit één regel code, maar uit meerdere regels code. We hoeven deze statement daarom niet te beëindigen met een `;`. Het einde van het code-block wordt gekenmerkt door de sluitende `}`.

De code tussen de eerste `()` van de if-statement wordt alleen uitgevoerd indien de boolean expressie die tussen de `()` staat waar is. Is dit niet zo? Dan wordt het gehele code-block door de compiler overgeslagen. Laten we dit eens in een werkend voorbeeld bekijken:

```
public class MyClass {
    public static void main(String[] args) {
        int x = 0;

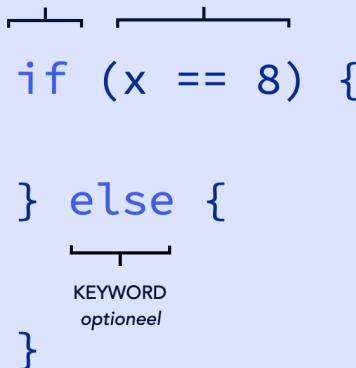
        if (x != 0) { // false
            x = x + 10; // wordt niet uitgevoerd
        }
        System.out.println(x); // 0

        if (x == 0) { // true
            x = x + 10; // wordt wel uitgevoerd
        }
        System.out.println(x); // 10
    }
}
```

Wanneer we alternatieven willen toevoegen, kunnen we een `else`-blok toevoegen. Is er meer dan één alternatief? Dan gebruik je `else if`-blokken tot je bij de laatste optie komt.

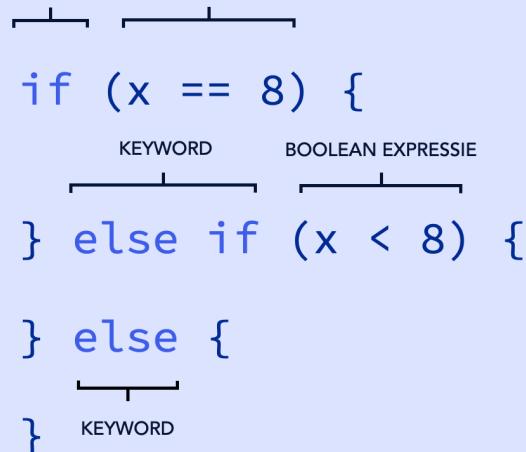
If-statement met één alternatief

KEYWORD BOOLEAN EXPRESSIE



If-statement met meerdere alternatieven

KEYWORD BOOLEAN EXPRESSIE



Wanneer je een else-if statement toevoegt, voeg je daarmee ook een nieuwe conditie toe die geëvalueerd wordt. Wanneer geen enkele conditie waar is, valt de compiler terug op het else-block. Dit kun je in het volgende voorbeeld in actie zien. We noemen de verschillende else-if en else blokken ook wel branches. Je kunt zoveel else-if branches toevoegen als je zelf wilt. Als er echter een matchende conditie wordt gevonden, wordt het gehele if, else-if, else blok direct afgesloten. Er is dus altijd maar één branch waar.

```

public class MyClass {
    public static void main(String[] args) {
        int test1 = 0;

        if (test1 == 0) { // --> MATCH!
            System.out.println("0"); // --> print 0;
            // Jump naar PUNT 1, na een matchend blok worden de andere alternatieven niet gecheckt of uitgevoerd.
        } else if (test1 == 5) {
            System.out.println("5");
        } else if (test1 == 10) {
            System.out.println("10");
        } else {
            System.out.println("no match");
        }

        // -- PUNT 1 --
        int test2 = 5;

        if (test2 == 0) { // --> NO MATCH, jump naar punt 2.
            System.out.println("0");
        } else if (test2 == 5) { // -- PUNT 2 --> MATCH!
            System.out.println("5"); // --> print 5
            // Jump naar punt 3, na een matchend blok worden de andere alternatieven niet gecheckt of uitgevoerd.
        } else if (test2 == 10) {
            System.out.println("10");
        } else {
            System.out.println("no match");
        }

        // -- PUNT 3 --
        int test3 = 8;

        if (test3 == 0) { // --> NO MATCH, jump naar punt 4.
            System.out.println("0");
        } else if (test3 == 5) { // -- PUNT 4 --> NO MATCH, jump naar punt 5.
            System.out.println("5");
        } else if (test3 == 10) { // -- PUNT 5 --> NO MATCH, jump naar punt 6.
            System.out.println("10");
        } else { // -- PUNT 6 --
            System.out.println("no match"); // --> print "no match"
            // Jump naar punt 7.
        }
        // -- PUNT 7 --
    }
}

```

Uiteraard kun je de code ook zelf in IntelliJ runnen om te zien wat er precies geprint wordt! Hieronder zie je nog een voorbeeld van hoe if statements gebruikt kunnen worden om beslissingen te nemen in de code.

```

public class MyClass {
    public static void main(String[] args) {
        int grade = 7;

        if (grade <= 5.5) { // Deze branch wordt overgeslagen
            System.out.println("Onvoldoende, herkansen");
        } else if (grade < 6.5) { // Deze branch wordt overgeslagen
            System.out.println("Hakken over de sloot");
        } else if (test1 < 8) { // Deze branch wordt uitgevoerd
            System.out.println("Netjes!");
        } else { //Deze branch wordt niet meer uitgevoerd
            System.out.println("Cum Laude");
        }
    }
}

```

```

    }
}

```

3.2 Loop statements

Zoals je inmiddels gemerkt hebt, zijn if-statements ontzettend handig om stukjes code conditioneel uit te voeren. Maar we gaan ook heel veel situaties tegenkomen waarin we code moeten herhalen totdat een bepaalde conditie is bereikt.

Denk bijvoorbeeld aan operaties die je wilt toepassen op elk element in een lijst. Denk maar aan de lijst met behaalde tentamencijfers: het kan zomaar gebeuren dat elke cijfer met 1.5 punt moet worden verhoogd, omdat het tentamen erg moeilijk bleek. Of wanneer je bijvoorbeeld een lijst met productprijzen wil sorteren van laag naar hoog. Lijsten worden altijd opgeslagen als **arrays** of **ArrayLists**, die we later zullen behandelen. We benoemen ze wel alvast, omdat ze heel vaak gebruikt worden in combinatie met loops.

Als we drie keer hetzelfde willen doen, kunnen we het stukje code daarvoor natuurlijk nog twee keer kopiëren... Maar daar houden we bij programmeren niet van. Niet alleen is dit heel onhandig, want als je toch iets wilt veranderen moet je dat op drie plekken aanpassen. Daarnaast komt het ook vaak voor dat we van tevoren niet precies weten of we het drie, vier of twaalf keer willen doen. We weten immers niet altijd hoe groot een lijst is, of hoe "ingesorteerd" een lijst is. Daarom kunnen we in Java gebruik maken van for-loops en while-loops. Het grootste verschil tussen deze twee, is dat een for-loop wordt gebruikt wanneer het aantal herhalingen van tevoren bekend is. Dus wanneer je een lijst van getallen hebt (1, 7, 2, 8) en van ieder getal wil printen of het getal even of oneven is. We kunnen direct zien hoe lang de lijst is en zullen alle getallen slechts één keer langs hoeven te gaan. Je kunt aan een lijst namelijk vragen hoe lang de lijst is, zodat je deze waarde kunt gebruiken in de for-loop.

Wanneer het aantal herhalingen van tevoren nog niet bekend is, vaak omdat de conditie wat complexer van aard is, gebruik je een while-loop. Stel dat je een lijstje van getallen wilt sorteren door telkens twee niet-gesorteerde elementen om te wisselen in de lijst:

```

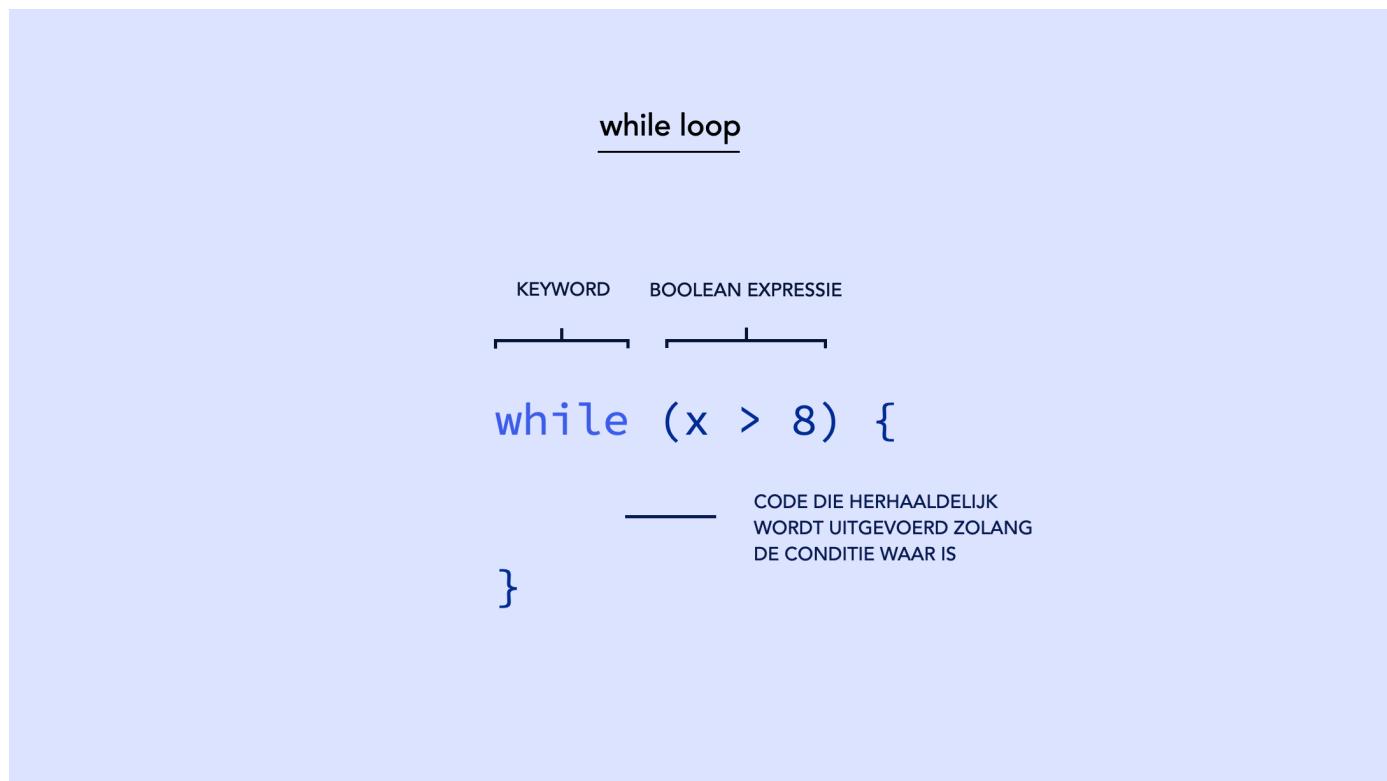
Iteratie 0: 1,7,3,6,2,5
Iteratie 1: 1,3,7,6,2,5
Iteratie 2: 1,3,6,7,2,5
Iteratie 3: 1,3,6,2,7,5
Iteratie 4: 1,3,2,6,7,5
...

```

In dit geval is het niet zo gemakkelijk om een for-loop te gebruiken. We weten van te voren niet wanneer we klaar zijn met sorteren. Bovendien is dat ook afhankelijk van de lijst die we binnen krijgen. De lijst 1,2,3 zou namelijk direct al klaar zijn! Dit is een mooi voorbeeld van een probleem dat je wil oplossen met een while-loop.

While-loops

While-loops zijn statements waarbij de code binnen het codeblok herhaaldelijk wordt uitgevoerd, net zolang tot de boolean expressie van de while loop false blijkt te zijn. Let op: dit kan dus ook 0 keer zijn!



In het geval van een while-loop zijn de meest simpele boolean expressies (true of false) natuurlijk niet zo betekenisvol meer. De loop while(false) {} betekent dat het codeblok niet wordt uitgevoerd, terwijl while(true) {} nooit meer stopt met uitvoeren. Dit wordt ook wel een **endless or infinite loop** genoemd. Het is logischer om de boolean expressie te gebruiken met een betekenisvolle variabele, zoals een getal. In onderstaand voorbeeld hebben we twee varianten van dezelfde code beschreven: één waarbij we alles handmatig uitschrijven en één waarbij we slim gebruikmaken van een while-loop:

```

public class MyClass {
    public static void main(String[] args) {
        // Versie 1
        int i = 0;
        System.out.println(++i);
        System.out.println(++i);
        System.out.println(++i);
        System.out.println(++i);
        System.out.println(++i);
        System.out.println(++i);
        System.out.println(++i);
        System.out.println(++i);
        System.out.println(++i);
        System.out.println(++i);

        // Versie 2
        int j = 0;
        while (j < 10) {
            System.out.println(++j);
        }
    }
}

```

Versie 2 is natuurlijk veel efficiënter. Laten we eens inzoomen op hoe deze code wordt gelezen:

1. We beginnen met het declareren van een variabele van het type **int** genaamd **j**, die direct de waarde 0 toegewezen krijgt.
2. Daarna wordt de expressie van de while-statement gecheckt om erachter te komen of de code die daarin staat, ook daadwerkelijk uitgevoerd moet worden. De expressie **j < 10** evalueert naar **0 < 10** wat evalueert naar **true** dus wordt het codeblok uitgevoerd.
3. Vervolgens mag **++j** geprint worden, maar ook dit zullen we eerst evalueren. **++j** betekent "Doe eerst **j** = **j + 1**, en haal **j** op". Het printen van **j** geeft daarom **1**.
4. Er staat verder niets meer binnen het codeblok van de while-loop, dus springen we terug naar de conditie.

5. We evalueren opnieuw de boolean expressie `j < 10` wat evalueert naar `1 < 10` wat evalueert naar `true`, dus wordt het codeblok nogmaals uitgevoerd.

... En ten slotte zullen we uitgekomen zijn bij het moment waarop `j` de waarde `10` bevat. Wanneer de conditie nu opnieuw geëvalueerd wordt, evalueren we `j < 10` naar `10 < 10` wat evalueert naar `false`. Dit betekent dat het codeblok nu wordt overgeslagen en het programma verder gaat met de statements na de closing bracket `}`. In ons voorbeeld betekent dit het einde van ons programma.

Naast while-loops bestaan er ook do-while loops. Een do-while loop wordt, in tegenstelling tot de while-loop, een stuk minder vaak gebruikt. Voor de volledigheid vermelden we hem hier wel nog even. Het verschil tussen deze twee loops, is dat een while-loop nooit uitgevoerd kan worden als de conditie de eerste keer al evalueert naar false. Bij een do-while-loop wordt de code eerst uitgevoerd en wordt de conditie achteraf gecheckt. Daardoor wordt een do-while-loop altijd minimaal één keer uitgevoerd.

De syntax voor de do-while loop is:

```
do {  
    // jouw code  
} while(conditie);
```

For-loops

Een for-loop wordt dus, net als een while-loop, gebruikt om een stuk code meerdere malen te kunnen herhalen. Bij een for-loop weten we echter van tevoren al hoe vaak we willen dat dit gebeurt: we hebben een vast aantal herhalingen in gedachten. Een for-loop bestaat daarom uit drie hoofdonderdelen:

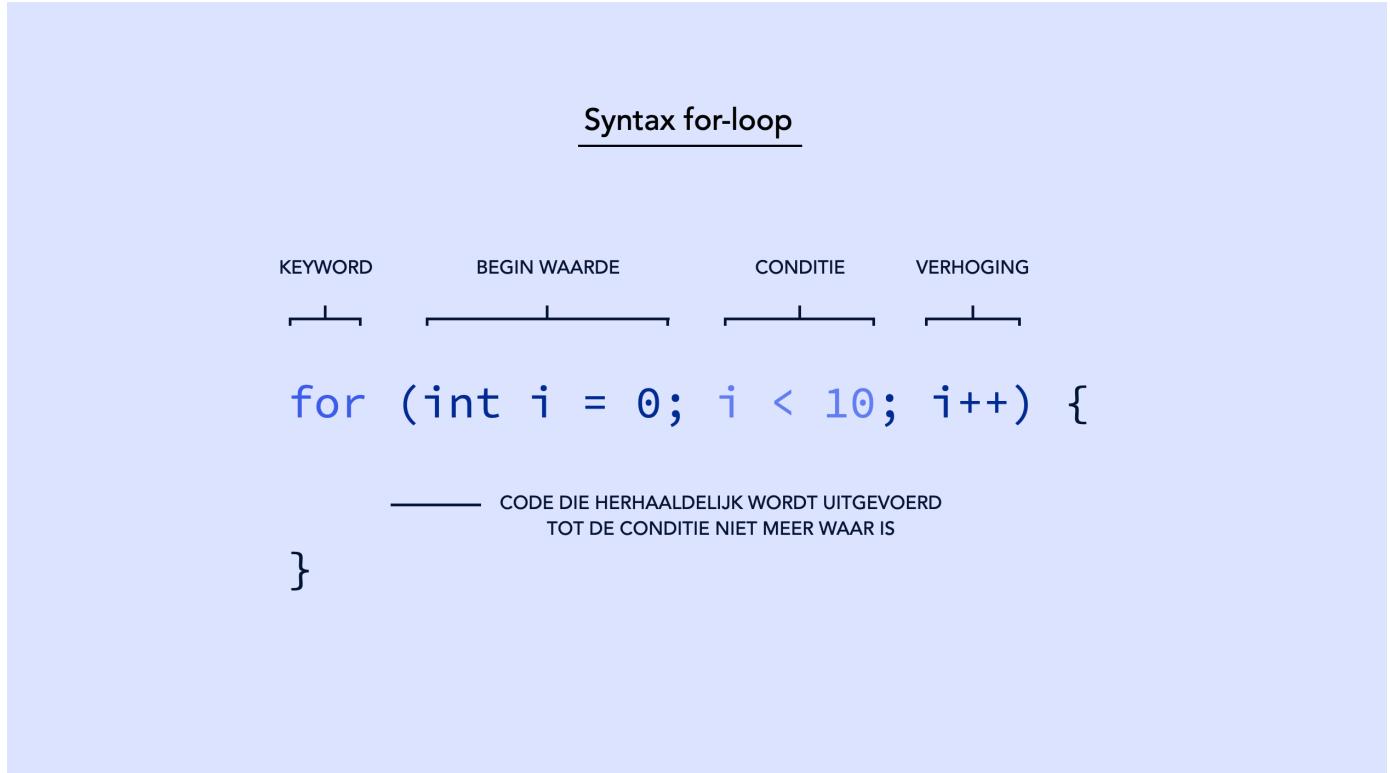
INITIALISATIE	CONDITIE	UPDATE
Declareer een variabele die zich als een soort teller gevraagd en bepaal de beginwaarde	De loop zal zich blijven herhalen zolang de teller kleiner is dan het gestelde maximum	Aan het einde van iedere iteratie wordt de teller met deze hoeveelheid verhoogd
<code>int i = 0;</code>	<code>i < 10;</code>	<code>i++</code>

1. De **initialisatie**: hierin declareren we een soort teller en wijzen een beginwaarde toe. Vaak is dit 0, maar dit kan ook 10 of 100 zijn.

2. De **conditie**: hierin wordt na iedere herhaling opnieuw een expressie geëvalueerd waarmee bepaald wordt of de loop stoppt, of nog een herhaling mag maken. Deze conditie stellen we altijd op basis van de waarde van de teller: bij een positieve for-loop zullen we stellen hoe *laag* de teller maximaal mag worden.

3. De **update**: hierin bepalen we hoe groot de stappjes zijn waarop de teller na iedere herhaling wordt verhoogd of verlaagd, zoals `+1` of `-2` o.i.d.

De syntax van een klassieke for-loop die tien keer herhaalt, ziet er als volgt uit:



Wanneer we er bijvoorbeeld voor willen zorgen dat er in de terminal tot 10 geteld wordt, zouden we dat als volgt beschrijven:

```
for (int i = 0; i < 10 ; i++) {
    System.out.println(i);
}
```

Probeer bovenstaand voorbeeld zelf maar eens uit om te zien wat er gebeurt! Maar hoe komen we tot dit eindresultaat?

Wanneer Java de for-loop tegenkomt, wordt als eerste de variabele `i` gedeclareerd. Deze variabele krijgt direct de waarde 0 toegewezen. Vervolgens controleert Java of de variabele `i` nog voldoet aan de boolean expressie. `i < 10` evalueert naar `0 < 10` evalueert naar `true`. Het codeblock wordt dus uitgevoerd. Vervolgens sprint Java terug naar eerst regel van de for-loop en incrementeert de variabele `i`, in dit geval met `i++`.

Java controleert opnieuw of de inhoud van de variabele (op dit moment `1`) nog voldoet aan de boolean expressie. `i < 10` evalueert naar `true`, dus opnieuw wordt het codeblock uitgevoerd. Dit blijft zichzelf herhalen tot `i` een waarde heeft van `10`. `i < 10` evalueert naar `10 < 10` evalueert naar `false`, dus de loop stoppt.

Een for-loop wordt dus altijd gebruikt in combinatie met een `int`. Het gebruik van de variabele genaamd `i` is niet verplicht, maar wel gebruikelijk. `i` staat dan voor "index" of "iterator". Je kunt echter elke variabele van elk type gebruiken, zolang je maar een declaratie, conditie en update hebt.

Een grappig voorbeeld (met een knipoog naar naar het hoofdstuk over `floats`) is de volgende for-loop:

```
for (float a = 0; a <= 1; a += 0.1f) {
    System.out.println(a*10);
}
```

Deze code print:

```
0.0
1.0
2.0
3.0
4.0
5.0
6.0
7.0000005
8.000001
9.000001
```

Je ziet dat de for-loop prima werkt met een `float`, maar dat de `float` na 6 iteraties zijn precisie verliest waardoor er fracties bijkomen. Dat is ook de reden dat 10 niet meer geprint wordt. Na de laatste iteratie is `a` 0.9000001. Wanneer de daar 0.1 bij optellen is dat 1.0000001. De boolean expressie `a <= 1` evalueert naar `1.0000001 <= 1` evalueert naar `false`.

Java kent ook een variant van de for loop, de for-each loop genaamd. Deze zullen we later in deze cursus behandelen.

3.3 Jump statements

Java kent twee **jumpstatements** die de flow van for- en while-loops kunnen beïnvloeden: de **continue**-statement en **break**-statement. Deze twee statements zul je over het algemeen tegenkomen in combinatie met een if-statement in while-loops, maar kunnen ook in for loops gebruikt worden. Dat komt omdat continue en break onafhankelijk van de conditie van de loop acteren.

Dus hoe werken ze? **continue** zorgt ervoor dat de loop voortijdig beëindigd wordt en door gaat naar de volgende herhaling. Als de conditie van de loop nog steeds `true` is, wordt het codeblok opnieuw uitgevoerd. **Break** werkt net ietsje anders: dit statement zorgt ervoor dat de loop in z'n geheel beëindigd wordt. Er worden geen verdere herhalingen meer uitgevoerd.

Dit kun je in je voordeel gebruiken wanneer je bijvoorbeeld alle even getallen tussen 0 en 50 wil printen. Kijk maar eens naar onderstaand voorbeeld:

```
public class MyClass {
    public static void main(String[] args) {
        int i = 0;

        while (true) {
            i++;
            if (i > 50) {
                break;
            }
            if (i % 2 != 0) {
                continue;
            }
            System.out.println(i);
        }
    }
}
```

}

We declareren de variabele `i` en wijzen direct de waarde 0 toe. Daarna starten we een while-loop. De conditie van de while-loop evalueren altijd naar `true`: deze loop zal uit zichzelf dus nooit eindigen. Vervolgens incrementeren we variabele `i`. De if-statement heeft een conditie die evalueren naar `false`. Dit codeblok wordt tijdens deze herhaling genegeerd.

De volgende *if*-statement, $1 \leq 2 != 0$, evaluateert naar $1 \neq 2 = 0$ evaluateert naar $1 \neq 0$ evaluateert naar true. Dit codeblok wordt deze herhaling wel uitgevoerd. De statement `continue` zorgt er echter voor dat het codeblok vroeglijdig wordt beëindigd: de `print`-statement wordt niet uitgevoerd en we beginnen opnieuw in de volgende herhaling. De conditie van de loop evaluateert altijd naar true, dus we incrementeren de variabele i . $2 > 50$ evaluateert nog steeds naar false, maar $2 \leq 2 != 0$ evaluateert naar $0 \neq 0$ wat evaluateert naar false. Nu negeren we dus ook dit codeblok, waardoor we wel uitkomen bij `System.out.println(i) // 2`. Dit proces herhaalt zich totaal $i = 50$ evaluateert naar $50 \leq 50$ wat evaluateert naar true.

Nu wordt het codeblock met break uitgevoerd, waardoor de loop beëindigd wordt en stopt met herhalen.

3.4 Switch statements

Als laatste kent Java een **switch**-statement. Switch-statements opereren als het ware op één specifieke variabele, waaraan verschillende uitkomsten - ofwel "cases" gekoppeld kunnen worden. In onderstaand voorbeeld zie je een voorbeeld van het gebruik van een switch-statement, waarbij gekeken wordt naar de waarde van browser in de statement switch(*browser*).

```
String browser = "Chrome";  
  
switch (browser) {  
    case "Chrome":  
        System.out.println("Google");  
        break;  
    case "FireFox":  
        System.out.println("Mozilla");  
        break;  
    default:  
        System.out.println("Some Other Browser");  
}
```

Dit programma zal de code uitvoeren van de eerste case waarvan diens waarde overeenkomt met browser. Java blijft daarna regels uitvoeren tot het einde van het codeblock - of tot de compiler een break statement tegenkomt. Vergeet je de break statement? Dan zullen dus alle cases na de machtigste case worden uitgevoerd.

De veelzijdigheid en mogelijkheid om zoveel verschillende opties toe te voegen geeft switch-statements veel voordeelen boven de if-else-if-else-constructie.

```
String browser = "Chrome";  
  
switch (browser) {  
    case "Safari OSX":  
    case "Safari iOS":  
        System.out.println("WebKit");  
        break;  
    case "Chrome":  
    case "Chromium":  
        System.out.println("Blink");  
        break;  
    case "IEexplorer":  
    case "Edge":  
        System.out.println("EdgeHTML");  
        break;  
    case "FireFox":  
        System.out.println("Gecko");  
        break;  
    default:  
        System.out.println("I have no clue");  
}
```

3.5 Opdracht: controlflow-constructies

Je kunt deze opdracht maken door het project te clonen of te downloaden naar jouw eigen computer via [deze GitHub repository](#). Hier vind je ook de gedetailleerde instructies van de opdracht zelf. De uitwerkingen staan op de branch `uitwerkingen`.

4. Methoden

4.1 Inleiding

Zonder dat je het doordat, heb je al meedere keren gebruik gemaakt van een methode toen je de opdrachten en voorbeelden van de voorgaande hoofdstukken doormaak. Een **methode** is een gegroepeerde set statements met een naam, die een specifieke taak uitvoeren. Dit maakt de code leesbaarder en makkelijker herbruikbaar.

Laten we dit eens vergelijken met een praktisch voorbeeld. Stel je voor dat je de inhoud van iedere willekeurige doos wil kunnen uitrekenen. Dan zou dit bestaan uit de volgende, specifieke stappen:

- Neem de "lengte" en de "breedte" van de doos;
 - Vermenigvuldig deze met elkaar. Sla dit op als "grondoppervlak";
 - Gegeven het "grondoppervlak", vermenigvuldig dit met de "hoogte";

Wanneer we deze stappen doorlopen voor één van de dossiers is dit prima. Maar als we 100 dozen te berekenen hebben, zijn we er na een tijdje goed klaar mee. We willen de specifieke instructies, de implementatie details, één keer beschrijven en er daarvan slechts naar refereren. Iedere keer als we de inhoud van een dossier willen uitlezen, willen we alleen zeggen: "Bereik inhoud op basis van lengte, breedte en hoogte".

En dit is precies wat methoden ons laten doen! Door specifieke stappen te bundelen tot één instructie die we herhaaldelijk kunnen uitvoeren, wordt onze code schoner en beter herbruikbaar.

Tot nu hebben we al enkele code geschreven in de main-methode, maar daar komt nu een einde aan. In dit hoofdstuk zul je kennis maken met verschillende soorten methoden die je kunt bouwen in Java. We zullen eerst vooraf gaan kijken naar wat een methode precies is, hoe je een methode maakt en ook hoe je deze vervolgens kunt toepassen.

Wanneer we een methode maken om de inhoud van een doos te berekenen, oefen **declareren**, zal alles wat eraan niet direct worden uitgevoerd. De statements die we hebben gedeclareerd in onze methode worden pas uitgevoerd op het moment dat we de methode uitdrukkelijk vragen om zijn taak uit te voeren. Dit noem je ook wel het **aanroepen** van de methode ("Bereken de inhoud").

Vaak is het ook zo dat methodes ook nog aanvullende informatie nodig hebben om hun taak uit te kunnen voeren. Voor de inhoud van een doel wordt dat ook: we zullen bij iedere nieuwe berekening de methode aanroepen met een andere lengte- en breedte. Stel dat jouw methode de oppervlakte van een ruimte berekent: de waarden waarop gerekend moet worden zijn anders bij de keuken dan bij de woonkamer. Er valt, iedere keer als deze methode wordt aangeroepen, altijd een lengte- en een breedte-maat aangeleverd moeten worden. Anders is dat om elkaar te vermenglijden! Wanneer een methode waardes heeft om zijn taak uit te voeren, zoals een lengte en een breedte, noem je dit **parameters**. Mocht je nou verwachten dat de methode niet alleen een taak uitvoert, maar ook daadwerkelijk een waarde teruggeeft - zoals de inhoud van de doos - dan spreken we van een **return value**.

4.2 De main methode

Laten we het allereerste voorbeeld erbij pakken, namelijk de bekende main-methode die je tot nu toe telkens voorbij hebt zien komen.

```
// MyClass.Java  
  
public class MyClass {  
  
    public static void main(String[] args) {  
        System.out.println("Hello World!"); // Prints "Hello World!"
```

```

    }
}

```

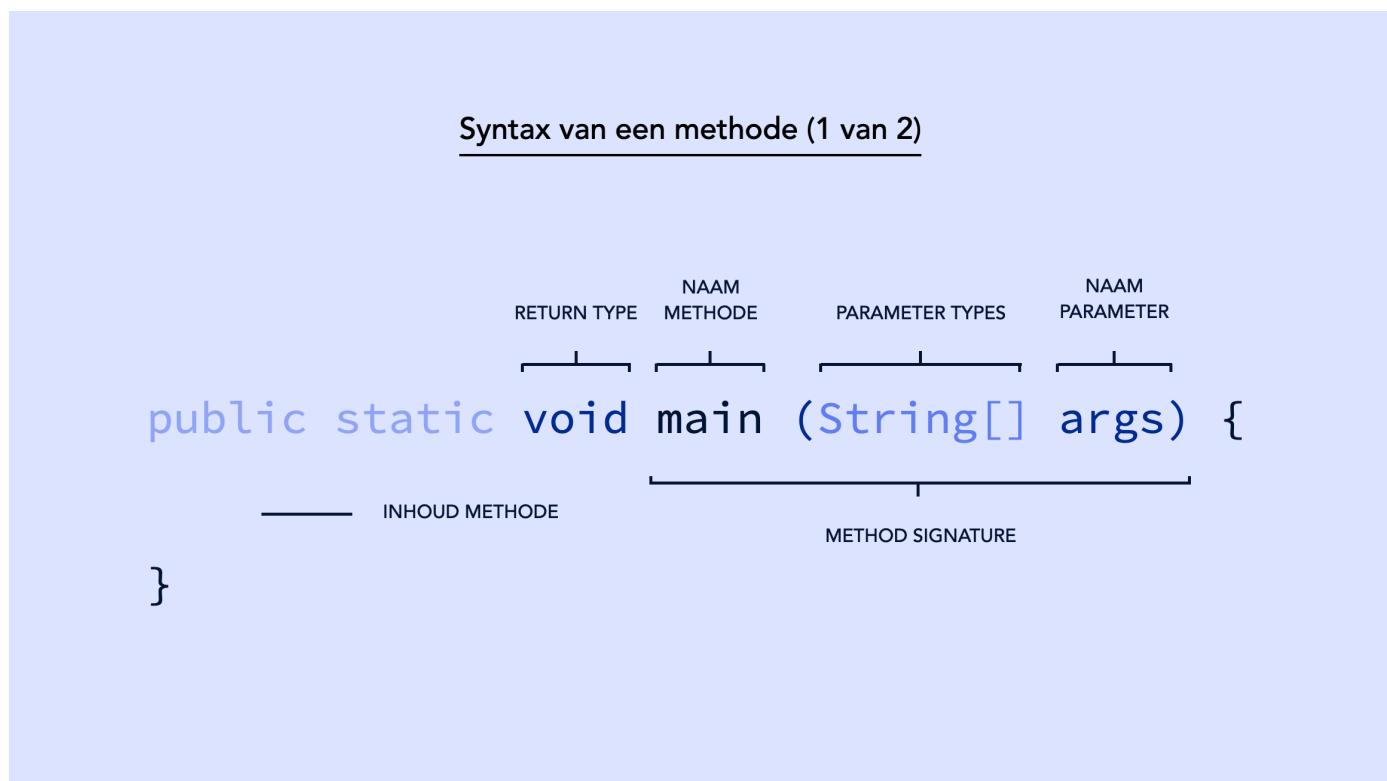
In dit voorbeeld zien we een class (`MyClass`), één methode met een bijzondere betekenis en één statement met één expressie. Het resultaat van deze methode is dat de string "Hello World" in de terminal geprint wordt. Laten we de syntax van deze methode eens analyseren:

```
public static void main (String[] args) {}
```

Omdat de betekenis en werking van de accesmodifier `public` en het keyword `static` pas belangrijk worden wanneer we het over classes gaan hebben, zullen we deze voor nu nog even negeren. We zullen ons focussen op de rest van de syntax:

```
void main (String[] args) {}
```

De elementen die je hierin terug ziet komen - het **return-type**, de **method signature** en het **codeblok** - zul je nodig hebben voor bijna iedere methode die je in Java zult schrijven.



Laten we van links naar rechts gaan: het **return-type** van deze specifieke methode is `void`. Dat betekent dat deze methode *niets* teruggeeft. Wanneer je `void` letterlijk vertaald naar het Nederlands, betekent het "leegte". Het zal je dan ook niets verbazen dat een methode met een return-type `void` geen resultaat oplevert. Dit klinkt misschien een beetje gek, maar we zullen ook methodes gaan bekijken die *wél* resultaat opleveren.

De rest van de syntax noem je **samenvatend de method signature**. Deze begint altijd met de naam van de methode. In dit geval is dat `main`, maar wanneer je nieuwe methodes gaat schrijven mag je deze in theorie ook aanpassen of `doSomethingCool` noemen. Daarna worden de parameters benoemd: `String[] args`. Een parameter bestaat, net als een variabele, uit een type (`String[]`) gevolgd door een naam (`args`). Wat parameters precies zijn en waar je ze voor gebruikt, zullen we later in dit hoofdstuk behandelen.

Ten slotte volgt het codeblok `{}` van de methode, waarin we statements mogen neerzetten.

4.3 Een methode als verzameling van statements

Oké, je weet nu misschien hoe we de verschillende onderdelen van een methode declaraties noemen, maar wat doen al deze onderdelen nou precies? In de komende paragrafen zullen we vanaf de simpelste methode gestaag opbouwen naar complexere methodes, waardoor de betekenis en het gebruik van methode declaraties duidelijker zullen worden. We beginnen met het declareren van een hele simpele methode binnen onze `MyClass` class:

```
// MyClass.java
public class MyClass {
    public static void main(String[] args) {
        System.out.println("Hello World!"); // Prints "Hello World!"
    }

    private static void sayHello() {
        System.out.println("Hello");
    }
}
```

De access-modifier `private` mag je, net als `public` en het keyword `static`, voor nu negeren. Ook mag je voor nu aannemen dat iedere methode (behalve de `main`-methode) de access-modifier `private` krijgt. Laten we even inzoomen op de rest van de methode:

```
void sayHello(){
    System.out.println("Hello");
}
```

Het return type van de methode is `void`: deze methode levert dus geen resultaat op. Er zijn ook geen parameters gedefinieerd tussen de `()`. Hiermee kun je stellen dat de signature van deze methode dicteert dat er geen input is (parameters) en ook geen output (return value).

De naam van deze methode is `sayHello`. Ik denk dat je wel kunt raden waarom we deze naam gekozen hebben: dit beschrijft het beste wat onze methode doet, zonder dat je alle statements eerst hoeft te lezen. Het is conveniente om methode-namen altijd met een *kleine letter* te laten beginnen. Mocht de naam uit meerdere woorden bestaan, dan schrijf je ieder opvolgend woord met een hoofdletter. Deze schrijfwijze wordt **CamelCase** genoemd. De naam van een methode kan geen spaties bevatten.

Side note: Net als bij classes mag een methode een naam hebben die bestaat uit letters, hoofdletters en lage streepjes. Het maakt voor Java dus niet uit hoe je de methode noemt. Voor je collega's wel, die worden een beetje verdrietig als je jouw methodes v4h_shw00 of PRACTICAL_PanInI noemt, gezien dit niet in één oogopslag duidelijk maakt waar de methode voor dient.

Als we bovenstaande class compileren en runnen, print het programma net als in het eerste hoofdstuk "Hello World!", maar geen "Hello". Nouja zeg! En onze `sayHello` methode dan?

Een void methode zonder parameters

De reden dat onze nieuwe methode compleet wordt overgeslagen, komt omdat we nergens de uitdrukkelijke instructie hebben gegeven om deze methode uit te voeren. Wanneer we de methode willen gebruiken, zullen we deze dus eerst aan moeten roepen.

Gek genoeg hoeftend we dit niet te doen bij de main-methode. Dat komt omdat Java bij het opstarten van een programma altijd eerst op zoek gaat naar een main-methode in het project. Deze methode wordt gezien als het startpunt en zal dan ook automatisch als eerste worden uitgevoerd. Bij alle methodes die we daarna toevoegen, werkt dat anders. Het aanroepen van een methode doen we door de signature te *matchen* op de plek waar je naar de methode wil springen om deze uit te voeren. Dat klinkt natuurlijk nog een beetje abstract, dus laten we de volgende code eens regel voor regel bekijken:

```
// MyClass.Java
public class MyClass {
    public static void main(String[] args) {
        sayHello();
        System.out.println("Hello World!"); // Prints "Hello World!"
    }
    private static void sayHello() {
        System.out.println("Hello"); // Prints "Hello"
    }
}
```

Dit code print:
Hello
Hello World!

Bij het starten van het programma roept Java dus automatisch de main-methode aan. Het eerste statement wat we daarin tegenkomen is de methode-aanroep `sayHello();`. In tegenstelling tot statements als `if`, `while` en `for` - wat keywords zijn in Java - heeft de compiler geen idee wat `sayHello` betekent. De compiler gaat dus op zoek naar een methode-declaratie die daarmee overeenkomt, ofwel: `matcht`. Omdat onze `sayHello`-methode dezelfde signature heeft (de naam komt overeen én de lijst met parameters is in beide gevallen leeg) springt het programma naar de eerste regel van deze methode. Vervolgens worden alle statements in de `sayHello`-methode uitgevoerd ("Hello" wordt geprint) en springt het programma terug naar de plek in het programma waar we waren voordat deze methode werd aangeroepen. Daar wordt "Hello World!" geprint.



Om het voorbeeld wat meer kracht bij te zetten, draaien we in onderstaande code de methode-aanroep en de `println`-statement om:

```
// MyClass.Java
public class MyClass {
    public static void main(String[] args) {
        System.out.println("Hello World!"); // Prints "Hello World!"
        sayHello();
    }
    private static void sayHello() {
        System.out.println("Hello"); // Prints "Hello"
    }
}
```

Nu print deze code:
Hello World!
Hello

De plek waar je de methode aanroeft heeft dus invloed op de volgorde waarin statements worden uitgevoerd. De plek waar je de methode declareert, heeft daar geen invloed op.

4.4 Een methode met argumenten

Er is je waarschijnlijk al een verschil opgevallen tussen onze main-methode en sayHello-methode. De main-methode verwacht namelijk input, ofwel **parameters**. Dit kun je zien in de declaratie: er is een lijst met parameters gedeclareerd als `(String[] args)`. Het datatype `String[]` hebben we natuurlijk nog niet behandeld, maar dit zal later zeker aan bod komen. Wat voor nu vooral belangrijk is, is om goed te kijken naar hoe we waardes *mee kunnen geven* wanneer we een methode aanroepen.

De reden dat we waardes - opgeslagen in variabelen - moeten meegeven, komt door een principe dat we **scope** noemen. Hier zullen we in het hoofdstuk over Classes dieper op in gaan, maar voor nu kun je werken met het ezelsbruggetje dat een variabele alleen "leeft" tussen de accolades `{}` waarbinnen hij is gedeclareerd:

```
// MyClass.Java
public class MyClass {
    public static void main(String[] args) {
```

```

    // shouldPrintExclamation bestaat "van hier...
boolean shouldPrintExclamation = true
System.out.println("Hello World!"); // Prints "Hello World!"
sayHello();
sayWorld();
// ...tot hier
}

private static void sayHello() {
    System.out.println("Hello"); // Prints "Hello"
}

private static void sayWorld() {
    // deze regel geeft een foutmelding, omdat shouldPrintExclamation hier niet bestaat!
    if (shouldPrintExclamation) {
        System.out.println("World!");
    } else {
        System.out.println("World");
    }
}
}

```

In bovenstaand voorbeeld wordt de boolean `shouldPrintExclamation` gedeclareerd binnen de accolades van de `main`-methode. Dit betekent dat we deze boolean in de `sayWorld`-methode niet kunnen gebruiken, ook al proberen we dit hier wel. Wat de `sayWorld`-methode eigenlijk wil bereiken, is de string "World" printen wanneer de waarde van `shouldPrintExclamation` true is, of "World" printen wanneer de waarde false is.

De beslissing om het uitropteken wel of niet te printen wordt dus eigenlijk al genomen in de `main`-methode, want dat is de plek waar deze variabele is gedeclareerd en toegewezen. Daarom moeten we de waarde vanuit de `main`-methode naar de `sayWorld`-methode "brengen". Dit doen we met behulp van **parameters en argumenten**.

Men gebruikt de woorden parameter en argument vaak door elkaar heen, maar er bestaat een belangrijk verschil tussen de twee. Op het moment dat we een methode declareren en beschrijven welke waarden deze functie verwacht, hebben we het over parameters. De parameters `args`, `width` en `height` gedragen zich als placeholder die later "gevuld" worden met waarden. Op het moment dat we een methode aanroepen en specifieke waarden meegeven, hebben we het over argumenten. Deze waarden, zoals 4 en 18, worden tijdens deze iteratie toegewezen aan de parameters. Op het moment dat we de functie opnieuw aanroepen met andere argumenten, worden die waarden weer toegewezen aan de parameters.

We voegen eerst een parameter toe aan de signature van onze methode `sayWorld`:

```

private static void sayWorld(boolean shouldPrintExclamation){
    // deze regel mag nu wel, omdat shouldPrintExclamation is gedeclareerd als parameter
    if(shouldPrintExclamation){
        System.out.println("World!");
    } else {
        System.out.println("World");
    }
}

```

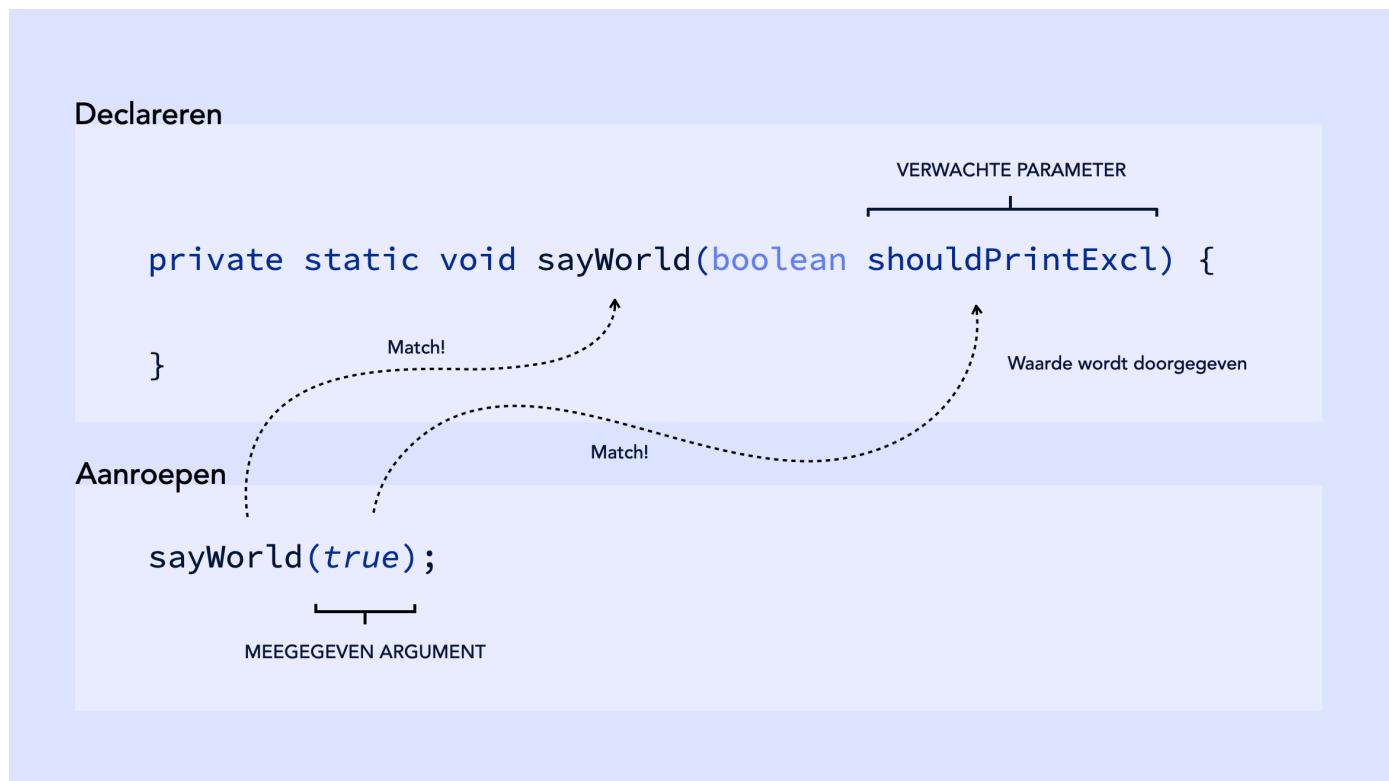
Het gebruik van parameters is een manier om de scope van {} te omzeilen. Wanneer een variabele als parameter van een methode gedeclareerd wordt, zal deze in het hele codeblok van de methode beschikbaar zijn. Maar juich niet te vroeg: wanneer we aangeven een parameter te willen ontvangen, moet deze ook daadwerkelijk worden meegegeven bij de aanroep. We kunnen een parameter namelijk niet toewijzen in de declaratie:

```

//Dit gaat fout, een parameter kan niet geassigned worden waar hij wordt gedeclareerd!
private static void sayWorld(boolean shouldPrintExclamation = false){
}

```

Hoe kunnen we onze boolean dan wel een waarde toekennen? Simpel! Dit doen we door de declaratie te matchen met de aanroep: we geven een argument mee. We vervangen de aanroep zonder argumenten (`sayWorld();`) daarom door een aanroep met argumenten (`sayWorld(true);`).



In plaats van true en false kunnen we als argument ook een variabele meegeven. Dit, mits de variabele van het juiste type is:

```

// MyClass.java
public class MyClass {

    public static void main(String[] args) {
        boolean shouldPrintExclamation = true;
        System.out.println("Hello World!"); // Prints "Hello World!"
    }
}

```

```

sayHello();
// De waarde van shouldPrintExclamation word doorgegeven aan de sayWorldMethod;
sayWorld(shouldPrintExclamation);
}

private static void sayHello() {
    System.out.println("Hello"); // Prints "Hello"
}

// shouldPrintExclamation = true
private static void sayWorld(boolean shouldPrintExclamation) {
    if (shouldPrintExclamation) {
        System.out.println("World!"); // Prints "World!"
    } else {
        System.out.println("World");
    }
}
}

```

4.5 Een methode met return value

De methodes die we tot nu toe hebben gezien hadden allemaal het return-type `void`, wat betekent dat de methode niets teruggeeft. Maar methodes kunnen naast het verwerken van input, ook output teruggeven. Sterker nog: dit is heel gebruikelijk.

Laten we kijken of we het voorbeeld uit de inleiding kunnen implementeren. We schrijven een set instructies die de inhoud van een doos kan berekenen en vervolgens in de terminal print. Eigenlijk is het heel erg logisch dat we deze instructies één keer omschrijven in een methode, zodat we de methode daarna alleen maar aan hoeven te roepen. We maken een methode en geven die een naam die beschrijft wat de methode doet:

```

private static int calculateBoxVolume(int length, int depth, int height){
}

```

Laten we onze method-signature weer eens onder de loep nemen en deze van links naar rechts analyseren:

```
int calculateBoxVolume(int length, int depth, int height)
```

In plaats van `void` is het return-type van deze methode `int`. Dit betekent dat deze methode bij het terugspringen naar de regel die hem aanriep, een `int` meegeeft. De naam van deze methode is `calculateBoxVolume` en verwacht drie parameter: `length`, `depth` en `height`, allemaal van het type `int`.

De methode verwacht dus drie `int`-waarden als input en geeft ook een `int`-waarde terug als output.

De signature van deze methode dient dat er drie inputwaarden zijn (parameters) en één outputwaarde (return value). Dit betekent dat we verplicht zijn om drie `int`-waarden mee te geven als argument, wanneer we deze methode willen aanroepen. Hierbij is het belangrijk om te weten dat we vrij zijn in het aantal parameters dat we willen declareren in een methode, maar dat we altijd maar één waarde mogen teruggeven.

Tijd om de implementatie details (de berekening) te beschrijven in het code block van onze methode:

```

private static int calculateBoxVolume(int length, int depth, int height){
    int base = length * depth;
    return base * height;
}

```

In bovenstaand voorbeeld zie je hoe we de parameters `length` en `depth` gebruiken om het grondoppervlak te berekenen. Vervolgens wordt deze variabele vermenigvuldigd met `height` en teruggegeven met behulp van het keyword "return". Om gebruik te kunnen maken van deze methode, zullen we hem in de main-methode aanroepen als `calculateBoxVolume(25, 30, 2)`. Gezien onze methode een `int` teruggeeft, willen we dit natuurlijk niet in de spreekwoordelijke prullenbak gooien. We willen deze waarde daarom direct toekennen aan een variabele zodat we deze kunnen printen!

```

// MyClass.Java

public class MyClass {
    public static void main(String[] args) {
        int volume = calculateBoxVolume(30,25, 50);
        System.out.println(volume);
    }

    private static int calculateBoxVolume(int length, int depth, int height){
        int base = length * depth;
        return base * height;
    }
}

```

Cool! We printen nu het volume in de terminal, maar eigenlijk willen we dit op een mooiere manier doen. We kunnen hier beter een aparte methode voor schrijven die het printwerk van de volume's voor ons afhandelt:

```

// MyClass.Java

public class MyClass {
    public static void main(String[] args) {
        int volume = calculateBoxVolume(30,25, 50);
        printBox(30, 25, 50, volume);
    }

    private static int calculateBoxVolume(int length, int depth, int height){
        int base = length * depth;
        return base * height;
    }

    private static void printBox(int length, int depth, int height, int volume) {
        System.out.println(" " + length + " | " + depth + " | " + height + " | " + volume);
    }
}

```

Hmmm... We missen nu eigenlijk wat begeleidende tekst, voordat we de waarden gaan printen. Laten we daar ook twee aparte methodes voor maken. Bovendien kunnen we in onze `printBox` methode ook controleren of alle dozen wel allemaal een standaard grootte hebben van `25 x 30`:

```

// MyClass.Java

public class MyClass {
    public static void main(String[] args) {
        printTitle();
        printHeader();

        int volumeTallBox = calculateBoxVolume(30,25, 50);
        printBox(30, 25, 50, volume);

        int volumeTinyBox = calculateBoxVolume(30,25, 2);
        printBox(30, 25, 2, volume);
    }

    private static int calculateBoxVolume(int length, int depth, int height){
        int base = length * depth;
        return base * height;
    }
}

```

```

}

private static void printBox(int length, int depth, int height, int volume) {
    boolean validSize = length == 25 && depth == 30;
    if(validSize) {
        System.out.println(" " + length + " | " + depth + " | " + height + " | " + volume);
    } else {
        System.out.println(" " + length + " | " + depth + " | " + height + " | Error: Box does not have valid sizes");
    }
}

private static void printTitle() {
    System.out.println("Welcome to Boxes.INC");
    System.out.println("All our boxes have a 25 by 30 base, but we can go TALL");
    System.out.println("10 to 90 is no problem at all!");
    System.out.println();
}

private static void printHeader() {
    System.out.println(" l   | d   | h   | volume");
    System.out.println("-----|-----|-----|-----");
}
}

```

Nu we alle voorbeelden hebben samengevoegd, kun je goed zien wat het verschil is tussen een void methode, methodes die input verwachten én methodes die input verwachten en output teruggeven. Kun jij onderscheiden wat de parameters en return types zijn? En zie je ook waarom het groeperen van statements in methoden (met een duidelijke naam) de leesbaarheid van de code verbeteren?

4.6 Refactoring

Wanneer je een tijdje bezig bent geweest met een project, zul je op ten duur merken dat je sommige statements of expressies wat efficienter kunt opzetten. Over het algemeen geldt de regel: hoe minder code, hoe beter. Het voorbeeld dat we in de vorige paragraaf hebben gemaakt beslaat nu best een behoorlijke lap code. Dat wordt mede veroorzaakt door het feit dat we best wel expliciet geweest zijn: alle waarden die we gebruiken hebben we eerst opgeslagen in een variabele. Echter, als we die variabele later niet nog een keer nodig hebben, kunnen we de uitkomst van expressies en methoden ook gewoon direct invoegen. Zolang je rekening houdt met de types, vindt Java het allemal best.

In de calculateBoxVolume-methode slaan we eerst het grondoppervlak van de doos op in de variabele base, voor we deze returnen. Maar ja, waarom eigenlijk? We kunnen hier ook direct de uitkomst van de expressie base * height returnen, omdat deze uiteindelijk evalueert naar een int.

```

private static int calculateBoxVolume(int length, int depth, int height){
    return length * depth * height;
}

```

Java begrijpt namelijk automatisch wat de types van expressies zijn: 1 == 1 is voor Java een boolean, 1 + 2 is voor Java een int, 1.1 + 2.2 is voor Java een double en 1.1f + 2.2f is voor Java een float. En dit geldt dus ook voor literals. Wanneer je ergens true typet, ziet Java een boolean. Wanneer je ergens 3 typet ziet Java een int, wanneer je ergens 3.3 typet ziet Java een double en wanneer je 3.3f typet ziet Java een float.

```

boolean b1 = true; // valide want `true` is een boolean
boolean b2 = b1; // valide want b1 is een boolean
boolean b3 = b1 == true; // valide want het resultaat van de expressie `b1 == true` is een boolean

if(true) {} // valide want if verwacht een boolean expressie en `true` is een boolean.
if(b1 == true) {} // valide want if verwacht een boolean expressie en `b1 == true` is een boolean.
if(b1) {} // valide want b1 is een boolean

```

Sterker nog, zelfs hele methodes worden door Java als een type gezien. boolean checkSomething() {} is voor Java een boolean, int add(int x, int y) {} is voor Java een int, double divide(int x, int y) {} is voor Java een double en float multiply(float x, float y) {} is voor Java een float.

Dit betekent dat we het volume van onze grote doos niet eerst in een variabele op hoeven te slaan voor we deze meegeven aan de printBox-methode: method:

```

public static void main(String[] args) {
    // We vervangen deze regels:
    int volumeTallBox = calculateBoxVolume(30,25, 50);
    printBox(30, 25, 50, volume);

    // ... Voor deze regel:
    printBox(30, 25, 50, calculateBoxVolume(30,25, 50));
}

```

Met deze kennis kunnen we ons eerdere voorbeeld toch weer een stukje korter opschrijven:

```

// MyClass.Java

public class MyClass {
    public static void main(String[] args) {
        printTitle();
        printHeader();

        printBox(30, 25, 50, calculateBoxVolume(30,25, 50));
        printBox(30, 25, 2, calculateBoxVolume(30,25, 2));
    }

    private static int calculateBoxVolume(int length, int depth, int height){
        return length * depth * height;
    }

    private static void printBox(int length, int depth, int height, int volume) {
        if(length == 25 && depth == 30) {
            System.out.println(" " + length + " | " + depth + " | " + height + " | " + volume);
        } else {
            System.out.println(" " + length + " | " + depth + " | " + height + " | Error: Box does nog have valid sizes");
        }
    }

    // - de methodes printTitle & printHeader blijven onveranderd -
}

```

4.7 Opdracht: methoden

Je gaat oefenen met het maken van methodes. Erst zul je dit doen door methodes te maken zonder *return value* (output) of parameters (input). Daarna ga je aan de slag met het gebruik van parameters en ten slotte schrijf je methodes die zowel parameters als *return values* hebben.

Je kunt deze opdracht maken door het project te clonen of te downloaden naar jouw eigen computer via [deze GitHub repository](#). Hier vind je ook de gedetailleerde instructies van de opdracht zelf. De uitwerkingen staan op de branch *uitwerkingen*.

5. Classes

5.1 Inleiding

Java is een **object georiënteerde** programmeertaal. Dit betekent dat de taal en onderliggende functionaliteit is georganiseerd rondom objecten, in plaats van acties. Objecten, in tegenstelling tot primitieve datatypes zoals int's en boolean's, zijn een verzameling van data en logica die iets met elkaar gemeen hebben. Door

gelijksortige informatie te kunnen groeperen, wordt onze code gemakkelijker onderhoudbaar en herbruikbaar. Daarnaast geeft het ons een manier om onze data te modelleren op een manier die lijkt op informatie uit de 'echte' wereld.

Zo kun je een Java-object bijvoorbeeld zien als het concept van een waterkoker. Want net als echte waterkokers, kan een Java-object eigenschappen hebben. Denk aan het merk, het type, de kleur en de inhoud. Aanvullend zal zo'n object ook waarden hebben die kunnen veranderen, zoals de hoeveelheid water die in de pot zit, de temperatuur van het water en de status van de deksel (open of dicht). En de waterkoker heeft natuurlijk ook functionaliteit: hij kan *verhitten*, *kooktijd berekenen*, of *stoppen*.

Wanneer de waterkoker het water gaat koken, zal de het aantal minuten dat het verwarmingselement moet verwarmen afhangen van de hoeveelheid water in de pot en de huidige watertemperatuur. Iedere seconde dat de waterkoker kookt, wordt de interne temperatuur met één tot twee graden verhoogd. De functionaliteit van het object heeft dus inherent invloed op de waarden binnen het object.

Door een verzameling te maken van eigenschappen, waarden en functionaliteit, kunnen we allerlei concepten modelleren. Hierin spelen **classes** (klassen, in het Nederlands) een belangrijke rol: objecten en classes zijn nauw verwant aan elkaar. Je kunt een class zien als een blauwdruk of template voor het object. Hierop staat bijvoorbeeld gespecificeerd dat wanneer je een waterkoker-object maakt, deze altijd een merk, type en kleur moet hebben. Bij het maken van het object bepaal jij dus zelf of je een zwarte of een witte waterkoker maakt. Maar omdat het object altijd gemaakt wordt met de blauwdruk van een class, dicteert deze welke onderdelen verplicht of optioneel zijn. En dit kun je zo ingewikkeld maken als je zelf wil: een class kan meerdere objecten bevatten met op hun beurt elk hun eigen data en methoden.



Wanneer we een "Smeg KLF05WHEU"-object hebben gemaakt met behulp van de "Waterkoker"-class, refereren we naar dit object als een instantie van die class. Alle data die we erin hebben gezet, zoals het merk, type, het huidige waterpeil en status van de deksel, noemen we ook wel de *properties* van het object. Het gedrag van het object, zoals verhitten of kooktijd berekenen, zijn beschreven met behulp van *methodes*.

Je zult al snel merken hoe krachtig classes en instanties zijn wanneer je complexe applicaties wil gaan bouwen. Daarom zullen we in dit hoofdstuk stap voor stap uitleggen hoe je kunt gebruiken.

5.2 De basis van een class

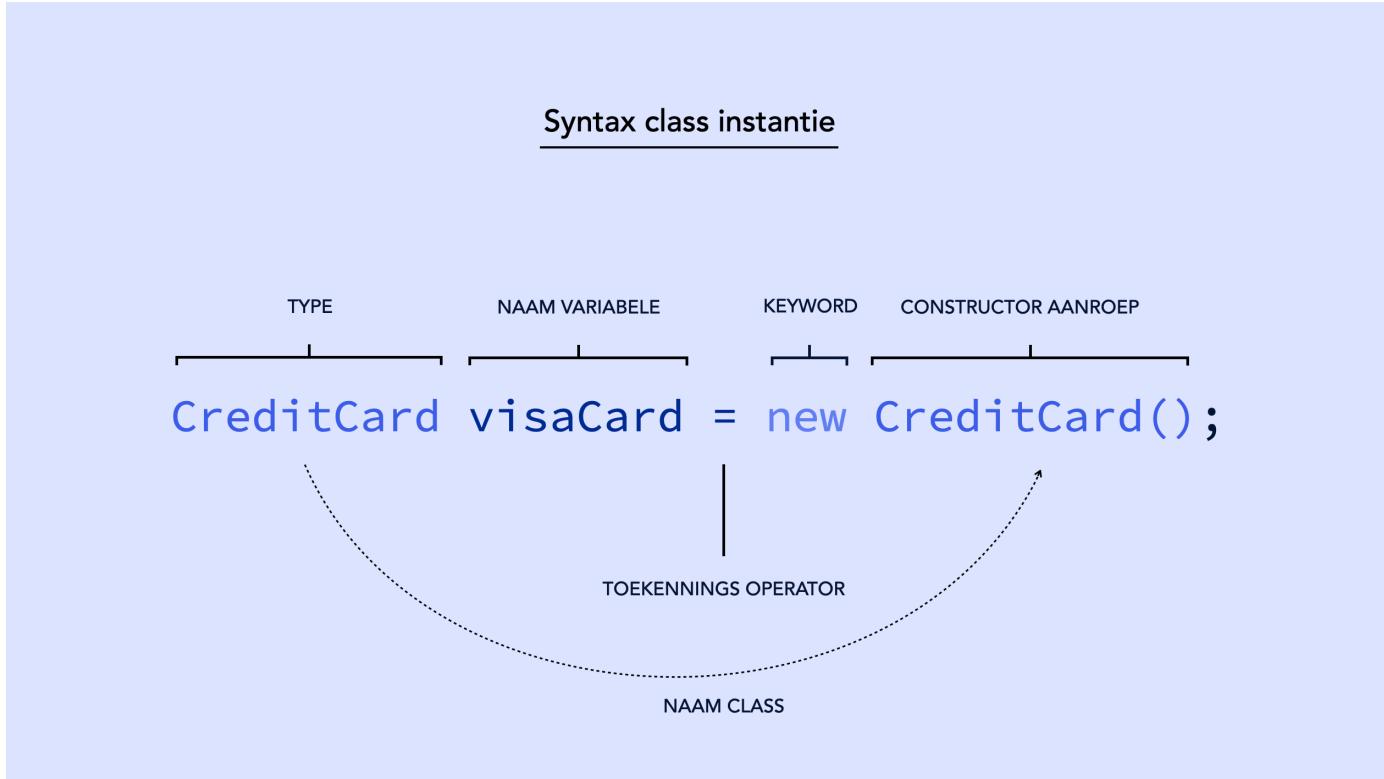
Je hebt inmiddels geleerd dat een class een blauwdruk is om instanties te maken. Daarvoor moet een class dus precies beschrijven welke data (properties) en welk gedrag (methoden) aanwezig moeten zijn om die data te manipuleren of uit te lezen.

In de class beschrijven we dus vrij exact hoe het object eruit moet komen te zien: de structuur ligt vast. Welke waarden er aan die structuur toegevoegd worden, verschilt per object.

Stel dat we bezig zijn om verschillende betaalmiddelen te programmeren, waaronder het gebruik van creditcards. Wanneer een gebruiker betaalt, moet het mogelijk zijn om de gebruikte kaart op te slaan. Zo kan de gebruiker de volgende keer sneller afrekenen. Laten we eens kijken naar een versimpeld voorbeeld, waar we eerst een algemene CreditCard class maken:

```
// CreditCard.java
class CreditCard {
    String nameOnCard;
    String number;
}
```

Wanneer een gebruiker zijn kaart wil opslaan, zullen we dus een instantie moeten maken van onze CreditCard-class. De syntax voor het maken van een nieuwe instantie ziet er als volgt uit:



Je bent vrij om de variabele waarin we ons object opslaan, iedere naam te geven die je wil. `Creditcard banaan = new CreditCard();`; had dus ook gemogen. Hierbij is het wel belangrijk dat het type van deze variabele overeenkomt met de class die we als blauwdruk gebruiken. Om de instantie daadwerkelijk te maken, hebben we het keyword `new` nodig. Daarachter roepen we de constructor van de `CreditCard`-class aan, door de ronde haken achter de naam van de class te plaatsen.

De inhoud van de properties die bij de class horen, `nameOnCard` en `number`, zullen bij iedere instantie anders zijn. Zo kunnen we bijvoorbeeld de `VisaCard` van "Max" en de `MasterCard` van "Henk" maken:

```
// Main.java
public class Main {
    public static void main(String[] args) {
        CreditCard visaCard = new CreditCard();
        visaCard.nameOnCard = "Max Verstappen";
        visaCard.number = "1234 5678 9012 3456";

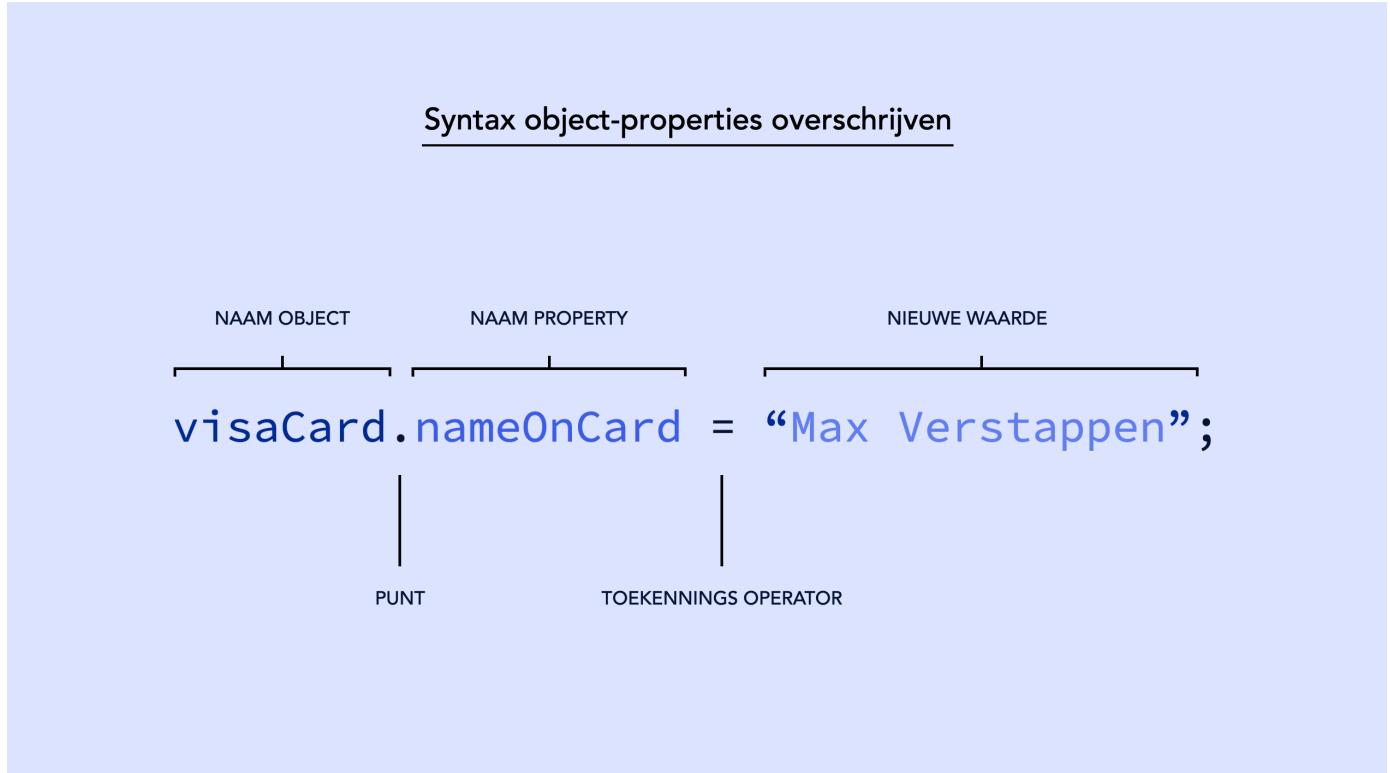
        CreditCard masterCard = new CreditCard();
        masterCard.nameOnCard = "Henk Pietersen";
        masterCard.number = "5678 9012 3456 7890";
    }
}
```

De reden dat we in staat zijn om instanties te maken van onze classes, komt dus door de **constructor**. Een constructor is een speciaal type methode die ervoor zorgt dat onze data in de juiste properties terecht komt. Grappig genoeg hebben wij zelf nooit een constructor geschreven voor onze `CreditCard`-class, maar toch zijn we in staat geweest om er instanties van te creëren... Hoe kan dat?

De constructor zorgt ervoor dat we instanties kunnen maken van een class

Dit was mogelijk omdat Java er eigenlijk vanuit gaat dat je altijd een instantie van een class wil kunnen maken: Er wordt een standaard constructor-methode gegenereerd. Deze is onderdeel van het standaard gedrag, waardoor je deze ook niet tussen de code ziet staan. En dat is ook de reden dat we *niets* hebben meegegeven tussen de ronde haken: `Creditcard visaCard = new CreditCard();` maakt alleen een leeg object, maar vult onze properties niet met informatie.

Om vervolgens de belangrijke properties die iedere creditcard moet hebben te vullen, hebben we de **punt-notatie** gebruikt. Met de punt-notatie kunnen we de properties op het object uitlezen of overschrijven.



Wanneer je werkt met classes, is het toekennen van properties op deze manier echter niet zo netjes. Het is beter om deze waarden direct mee te geven wanneer we deze instantie creëren:

```
CreditCard visaCard = new CreditCard("Max Verstappen", "1234 5678 9012 3456");
```

Om dit ook te laten werken, zullen we onze eigen constructor moeten maken. Een constructor lijkt gelukkig erg veel op een normale methode. Gezien we bij het aanroepen van de constructor twee argumenten (de naam en het nummer) mee willen geven, zullen we deze waarden ook moeten ontvangen als parameters wanneer we de constructor declareren in de class:

```
class CreditCard {
    String nameOnCard;
    String number;

    public CreditCard(String nameOnCard, String number) {
        // zet de juiste gegevens in de properties
        // nameOnCard en number bij het maken van een instantie
    }
}
```

Dus hoe onderscheid je een constructor-methode van een gewone methode? Aan het feit dat hij precies dezelfde naam heeft als de bijbehorende class én dat de methode-naam met een hoofdletter geschreven is. Daarnaast is het belangrijk dat een constructor geen *return-type* heeft. Nee, zelfs geen *void*!

Om de constructor vervolgens de ontvangen data te laten opslaan in de juiste properties, moeten we dit gedrag uitdrukkelijk implementeren. Onze parameters hebben op dit moment echter exact dezelfde naam als de properties in onze class... `nameOnCard = nameOnCard` en `number = number` zal dus niet werken. Om aan te geven welke `nameOnCard` van onze class is en welke `nameOnCard` de parameter is, duiden we de class-variant aan met `this`:

```
class CreditCard {
    String nameOnCard;
    String number;

    public CreditCard(String nameOnCard, String number) {
        this.nameOnCard = nameOnCard;
        this.number = number;
    }
}
```

Het keyword `this` geeft dus aan dat we het hebben over een property op deze class. Nu zorgt onze constructor ervoor dat bij het creëren van een instantie van de `CreditCard`-class, de meegegeven waarden gebruikt worden om de properties te vullen.

5.3 Class varianten van primitives

Eerder heb je kennis gemaakt met alle verschillende soorten variabelen in Java, specifiek de primitive. De primitive zijn bijzonder in de Java omgeving, omdat ze alleen een waarde vertegenwoordigen zoals `6` of `true`. Dat wil zeggen dat ze geen properties of methodes bevatten. Wat je echter nog niet wist, is dat Java standaard class-implementaties heeft van elke primitive variabele die je tot nu voorbij hebt zien komen!

Deze class-implementaties noemen we object types. Deze object types onderscheiden zich doordat het de primitive waarde als interne property bevat en deze verricht met methodes die relevant zijn voor dat specifieke type, net als onze `CreditCard`-class die we zojuist hebben aangemaakt. Zo heeft de `int` een `Integer`-object als tegenhanger, een `long` een `Long`, een `boolean` een `Boolean`, enz.

We zullen in de loop van dit hoofdstuk de significantie van deze object-types verder ontdekken. Voor nu is het voldoende om je bewust te zijn van het bestaan van deze objecten.

Net als de `CreditCard`-class een constructor heeft, heeft een object-variant van een primitive (zoals `Integer`) ook een constructor. Die zorgt er namelijk voor dat je er een waarde mee kunt aanmaken:

```
Integer x = new Integer(7);
```

Hoewel deze manier van aanmaken *werkt*, hoef je deze niet te gebruiken. Sterker nog: deze schrijfwijze wordt zelfs afgeraden. Dat komt omdat deze object typen zó vaak gebruikt worden dat het een standaard onderdeel van Java is. Deze standaard integratie noemen we *autoboxing* en zorgt ervoor dat het converteren van een primitive naar een object automatisch gebeurt.

```
Integer x = 7;
```

In bovenstaand voorbeeld kan de primitive waarde `7` (`int`) direct worden toegekend aan de variabele van type `Integer`, omdat Java de conversie voor ons regelt. Dit werkt ook in omgekeerde volgorde! Dit heet het *auto-unboxing* principe, waardoor het werken met de object types net zo makkelijk is als het werken met primitiveën:

```

Integer x = 7;
Integer y = 13;

// de waarden in Integer objecten x en y worden eerst automatisch primitieven gemaakt, daarna opgeteld en daarna wordt van het resultaat een nieuwe Integer gemaakt:
Integer z = x + y;

// Als we deze speciale syntax niet zouden hebben, zouden we het elke keer zelf moeten doen... zoals hier:
Integer x = new Integer(7);
Integer y = new Integer(13);

// gelukkig hoeven we niet deze bijna-onleesbare-variant te gebruiken!
Integer z = new Integer(x.intValue() + y.intValue());

```

String, het afwijkende object

In hoofdstuk 3 hebben we naast de primitieven ook het `String` object gebruikt. `String` is een afwijkend type binnen Java. `String` is namelijk wel een volwaardig object en eigenlijk een object-type variant van een échte primitieve, maar niet helemaal op de manier zoals `Integer` dat van `int` is of `Long` van `long`.

Het `String`-type heeft een identiteitscrisis. `String` is namelijk wel een volwaardig object. Het is eigenlijk een object-type-variant van een échte primitieve, maar niet helemaal op de manier zoals `Integer` dat van `int` is, of `Long` van `long`. Wat de `String` afwijkend maakt is dat hij, anders als alle andere volledige objecten, een soort autoboxing-achtige manier van aanmaken heeft.

```

// we hoeven geen constructor te gebruiken voor het maken van strings
String tekst = "Directe toekening";

```

Zoals je ziet is bestaat een echte primitieve van een `String` niet, maar als deze zou bestaan dan wordt deze gerepresenteerd door een array van individuele karakters:

```

// we hoeven geen constructor of iets dergelijks te gebruiken voor het gebruik van strings
String tekst = "Directe toekening";

// Als het geen speciale syntax zou hebben, zou het anders moeten worden aangemaakt. Ook hier hoeven we dit gelukkig niet zo te doen:
String tekst = new String(new char[] { 'D', 'i', 'r', 'e', 'c', 't', ' ', 'e', ' ', 't', ' ', 'o', ' ', 'k', ' ', 'e', ' ', 'n', ' ', 'i', ' ', 'n', ' ', 'g' });

```

5.4 Packages

Je kunt je waarschijnlijk wel voorstellen dat wanneer we het betaalsysteem van een échte webshop gaan bouwen, we daarvoor een hoop classes nodig hebben. Gezien dit snel onoverzichtelijk kan worden, is het gebruikelijk om gerelateerde classes te groeperen in een `package`. Een package is klinkt een stuk speciaalder dan het eigenlijk is: het is simpelweg een mapje in jouw project waarin je classes van hetzelfde type ophaalt.

Naast creditcard's, willen we gebruikers ook de mogelijkheid geven om betalingsvormen als debitcards (IDEAL) of PayPal te gebruiken. Hier hebben we dan drie aparte classes voor geschreven:

```

class CreditCard {
    // ...
}

class DebitCard {
    // ...
}

class PayPal {
    // ...
}

```

Dit zijn allemaal aan elkaar gerelateerde en kunnen we daarom mooi groeperen in een package. Het is gebruikelijk om iedere class op te slaan in een apart bestand: dit houd alles gemakkelijk doorzoekbaar. Bij conventie geven we packages altijd een naam in kleine letters en gebruiken we hierin geen spaties of lange streepjes. Een passende naam voor deze package zou bijvoorbeeld `paymentMethods` kunnen zijn. De projectstructuur zou er dan zo uit kunnen zien:

```

└── paymentMethods
    ├── CreditCard.java
    ├── DebitCard.java
    └── PayPal.java

```

In bovenstaand voorbeeld zie je dat we binnen de `paymentMethods`-package drie classes hebben, in ieder hun eigen `.java`-bestand. Om dit goed te laten werken, zullen we bovenaan ieder afzonderlijk bestand de naam van de package moeten benoemen:

```

// CreditCard.java
package paymentMethods;

class CreditCard {
    // ...
}

```

De naam van de map (`paymentMethods`) moet hierin *exact overeenkomen* met de packagenaam die we bovenaan de bestanden benoemen (`paymentMethods`). Maak je hier een spellfout in? Dan zullen de classes niet herkend worden als onderdeel van deze package.

Naming conventions

Met drie classes in onze package staat ons project nog in de kinderschoenen. Maar hoe meer we ons project uitbreiden en voorzien van groeiende functionaliteit, hoe groter de kans is dat we een `name collision` krijgen. Een name collision treedt op wanneer namen van variabelen, classes en methoden meerdere keren voor komen waardoor het programma niet weet welke het moet hebben en daardoor in de war. De kans dat dit probleem optreedt wordt groter naarmate een project groter wordt en componenten (libraries) geschreven door anderen gaan gebruiken, anderen die mogelijk dezelfde namen hebben gebruikt, zeker bij namen zo generiek als `paymentMethods`.

Omtrent dit probleem te voorkomen is er een speciale conventie, de `reverse domain name notation`, die wereldwijd voorkomt dat dit probleem optreedt zonder hierbij overleg of afstemming nodig te hebben. Bij deze conventie wordt gebruik gemaakt van iets dat al wereldwijd uniek is, namelijk de URL van jezelf, het bedrijf, de organisatie of de onderwijsinstelling. Omdat we al weten dat deze uniek is en afgesproken hebben dat we alleen gebruik zullen maken van URL's waar we zelf zeggenschap over hebben, weten we zeker dat er geen overlappende naamgevingen zullen ontstaan.

Wanneer je commerciële projecten maakt, in dit voorbeeld vanuit de organisatie met de URL `www.novi.nl`, ziet de structuur van packages er daarom zo uit:

```

n1
└── novi
    └── paymentMethods
        ├── CreditCard.java
        ├── DebitCard.java
        └── PayPal.java

```

```
// CreditCard.java
package nl.novi.paymentMethods;

class CreditCard {
    // ...
}
```

De benamingen `n1` en `novi` in de package-naam `nl.novi.paymentMethods` maken deel uit van de naam van de organisatie die de package maakt. Werk je voor Capgemini? Dan noem je de package `com.capgemini.paymentMethods`. Door jouw packages deze aanduiding mee te geven, kun jij zonder problemen externe packages gebruiken die gemaakt zijn door andere organisaties, ook al hebben de classes dezelfde naam. Het importeren van een `paymentMethods`-package van Microsoft zal dan geen name collision opleveren.

Hoewel het gebruik van deze naamgeving niet verplicht is, is dit wel zeer gebruikelijk binnen het Java-landschap.

5.5 Opdracht: Auto database

Je gaat een class schrijven die we kunnen gebruiken om oneindig veel Auto-instanties te kunnen creëren. Zo kun je een mooie start maken met het aanmaken van classes, instanties en alle bijbehorende logica.

Je kunt deze opdracht maken door het project te克lonen of te downloaden naar jouw eigen computer via [deze GitHub repository](#). Hier vind je ook de gedetailleerde instructies van de opdracht zelf. De uitwerkingen staan op de branch `uitwerkingen`.

5.6 Encapsulatie

We hebben natuurlijk een mooi begin gemaakt met onze CreditCard-class. Maar hoewel dit een goede eerste stap is, voldoet onze class nog niet aan het **encapsulatie principe** van object georiënteerd programmeren. Encapsulatie is een manier om de data en methoden van een class als het ware "in te pakken", zodat deze niet toegankelijk zijn van buitenaf. Voor de oplende lezer: we gaan je nu een-de-lijk vertellen waar die woorden `public` en `private` voor staan!

Onze CreditCard-class voldoet dus niet aan het encapsulatie principe. Tja. Dat mag dan best zo zijn, maar wat maakt het eigenlijk uit als een andere class toegang heeft tot data uit onze CreditCard-class?

Onze CreditCard-class is nu natuurlijk nog redelijk eenvoudig, maar wellicht kun je je voorstellen dat we onze class ooit zullen uitbreiden met methoden om geld af te schrijven. De CreditCard-class begrijpt wat er allemaal moet gebeuren bij de aankoop van een product: eerst moet de huidige bestedingslimiet worden gecheckt, daarna moet berekend worden of het aankoopbedrag de limiet niet overschrijft en ten slotte kan een transactie worden gemaakt.

Echter, als een andere class ook toegang heeft tot die logica, zou deze zomaar de helft van de stappen kunnen overslaan. De transactie kan worden gemaakt zonder dat er eerst gecontroleerd is of de limiet dit wel toelaat. En als je niet goed oplet, zou zomaar onze bestedingslimiet worden met "Banana". En daar kun je geen €134,- voor sportschoenen vanaf trekken...

Wanneer belangrijke methoden en properties niet worden afgeschermd, kan dit leiden tot **gegevensbeschadiging**: in ons geval ongelijke transacties of een onbruikbare limiet. Naast foute implementaties door developers zelf, zijn ook cyberaanvallen een reëel gevaar. Zonder encapsulatie zou iedere hacker - nog voor hij zijn eerste kopje koffie op heeft - een simpele class kunnen schrijven die met behulp van de methodes op onze CreditCard-class spullen bestelt met de creditcard van Max!

Encapsulatie helpt dus om onze data te beschermen tegen wijzigingen, door toegang te beperken voor klassen en objecten van buitenaf. Naast het feit dat veiligheid essentieel is voor een goedwerkende applicatie, zorgt het ook voor een hoger abstractieniveau. We kunnen classes namelijk ontwerpen om complexe werkingen uit te voeren, zonder dat de gebruiker van die class alle details hoeft te begrijpen. Wanneer we willen weten of de creditcard nog geldig is, hoeven we immers niet te weten hoe dit precies gevalideerd wordt. We zijn alleen geïnteresseerd in de uitkomst: `true` of `false`. We houden de logica dus zoveel mogelijk verborgen en beheren dit centraal op één plek.

Om de zichtbaarheid (access) van onze properties en methoden te beheren, gebruiken we in Java **access modifiers**. Zo kunnen we bepalen wat het toegangs niveau, ook wel **access level** van onze data is. De vier belangrijkste zijn `public`, `protected`, `package-private` en `private`. Wanneer we in onderstaande uitleg refereren naar de "eigenaar", bedoelen we daarmee het object waar de data in staat. Bijvoorbeeld de `masterCard` van Max.

- `Public`: data van dit niveau kan door iedere class of methode bereikt en gebruikt worden (zowel uitlezen als wijzigen);
- `Protected`: protected data kan alleen bereikt worden door de eigenaar, de package en afgeleide classes. Het principe van afgeleide classes (overerving) wordt in de volgende paragraaf behandeld.
- `Package-private`: dit is de `default` toegang. Deze geeft de eigenaar én alle classes in dezelfde package toegang tot de data van die eigenaar;
- `Private`: alleen de eigenaar van de data kan deze data lezen en wijzigen.

modifier	class	package	subclass	andere classes
<code>private</code>	ja	nee	nee	nee
	ja	ja	nee	nee
<code>protected</code>	ja	ja	ja	nee
<code>public</code>	ja	ja	ja	ja

Let op: voor `package-private` is geen keyword beschikbaar: een element is package-private als er géén access modifier voor dat element staat. Onze CreditCard class die we eerder in deze paragraaf hebben gemaakt, is dus package-private:

```
class CreditCard {
    String nameOnCard;
    String cardNumber;
}
```

Over het algemeen geldt de regel dat we de properties van een object vaak `private` maken met behulp van access modifiers. Dit toegangs niveau heeft de voorkeur voor alle data in een class. Alleen de methoden van de class *zelf* kunnen de private data bewerken, waardoor er niet van buitenaf (per ongeluk) een verkeerde wijziging kan plaatsvinden. De methoden die bij dat object horen, maken we in veel gevallen `public`:

```
class CreditCard {
    private String nameOnCard;
    private String cardNumber;

    public CreditCard(String nameOnCard, String cardNumber) {
        this.nameOnCard = nameOnCard;
        this.cardNumber = cardNumber;
    }
}
```

Oké, maar hoe kunnen we dan ooit nog het creditcardnummer van Max uitlezen of wijzigen? Alle properties zijn nu afgeschermd!

Hier maken we methodes voor die dit op een veilige manier afhandelen: **getters en setters**. Wil je het creditcardnummer van Max weten? Dat mag, maar dan moet je dat via onze get-methode doen. En we vinden het ook prima als je het creditcardnummer van Max wil wijzigen, maar dat nieuwe nummer moet wel aan een aantal voorwaarden voldoen. Zo mag het nieuwe nummer niet leeg zijn en moet het nummer exact 16 cijfers bevatten. Daar zorgt de controle in onze set-methode voor:

```
class CreditCard {
    private String nameOnCard;
    private String cardNumber;

    public CreditCard(String nameOnCard, String cardNumber) {
        this.nameOnCard = nameOnCard;
        this.cardNumber = cardNumber;
    }

    public String getNumberOnCard() {
        return this.cardNumber;
    }

    public void setNumber(String cardNumber) {
        if (cardNumber != null && cardNumber.length() == 16) {
            this.cardNumber = cardNumber;
        }
    }

    public String getNameOnCard(){
        return this.nameOnCard;
    }
}
```

}

Let op: het gebruik van setters is niet gekoppeld aan het gebruik van getters. In veel gevallen willen we namelijk niet dat een van de waarden van ons object zomaar kan veranderen, terwijl we de waarde wel moeten kunnen lezen. In dat geval maken we alleen maar getters en blijven we de toekenningen in de constructor doen. Op deze manier kunnen we de waarde alleen bij het aanmaken bepalen en garanderen dat bij elke `get()` altijd dezelfde waarde wordt teruggegeven. Dit principe wordt ook wel het **immutability pattern** genoemd. De object varianten van de primitieven zijn bijvoorbeeld allemaal immutable: onveranderbaar.

5.7 Overerving

Eerder in deze cursus heb je geleerd dat we tijdens het programmeren altijd zorgen dat we zo min mogelijk stukken code hoeven te herhalen. Niet omdat we lui zijn, maar omdat dit de kwaliteit van onze code ten goede komt. Dit doen we bijvoorbeeld door een for- of while-loop te gebruiken, of door onze logica te bundelen in methoden die we telkens opnieuw aan kunnen roepen. Dit zorgt er niet alleen voor dat onze logica consistent is (het veranderen van het creditcardnummer gebeurt altijd op dezelfde manier), maar ook dat we niet gaan huilen wanneer de stakeholders van het bedrijf laten weten dat er iets gewijzigd moet worden. Besluit de opdrachtgever dat creditcardnummers voortaan moeten beginnen met een unicorn-emoji ? Dan hoeven wij dat slechts op één plek te veranderen.

Dit principe noem je **DRY**, wat staat voor Don't Repeat Yourself. Dit principe is een wereldwijde best practice en kunnen we binnen Java ook nog op een andere manier toepassen. Dit kunnen we doen door **overerving**, ook wel *inheritance* genoemd. Overerving, zoals de naam al verklapt, zorgt ervoor dat de **ene class** specifieke properties of methoden kan **erven** van een andere class.

Dus in plaats van straks alle onze classes (`CreditCard`, `DebitCard` en `PayPal`) individueel te voorzien van een `withdraw`-methode, kunnen we ook een superclass maken die dit gedrag doorgeeft. In dit geval is het logisch om hier een `PaymentMethod`-class voor te schrijven:

```
// PaymentMethod.java
class PaymentMethod {
    private int balance = 0;

    public void withdraw(int amount) {
        balance = balance - amount;
    }
}
```

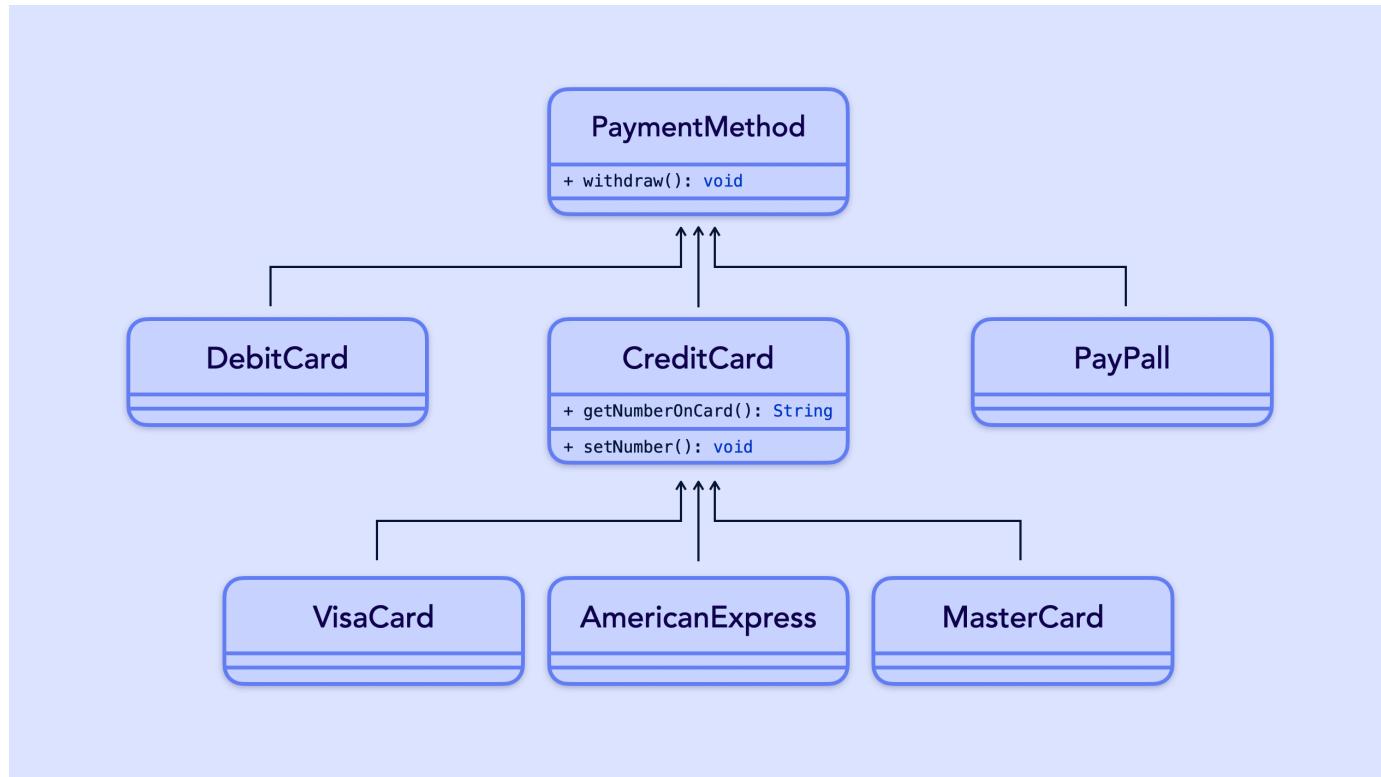
Deze class heeft geen constructor nodig, omdat de `balance`-property bij iedere nieuwe instantie op 0 wordt gezet en dus niet meegegeven hoeft te worden wanneer er een nieuwe instantie wordt gemaakt. Om vervolgens aan te geven dat onze andere classes de properties en methoden van de `PaymentMethod`-class moeten erven, moeten we aangeven dat ze **afstammelingen** zijn van deze class. Dit doen we door het `extends` keyword te gebruiken:

```
// CreditCard.java
class CreditCard extends PaymentMethod {
    // ...

// DebitCard.java
class DebitCard extends PaymentMethod {
    // ...

// PayPal.java
class PayPal extends PaymentMethod {
    // ...
```

Dit kunnen we echter nog verder uitbreiden! CreditCards komen natuurlijk in allerlei soorten en maten en hebben allemaal hun eigen implementaties. Daarom is het geen gek idee om de `CreditCard` ook een superclass te maken en hier de subclasses (afstammelingen) `VisaCard`, `MasterCard` en `AmericanExpress` mee te maken.



Dit begint al op een echte applicatie te lijken! Er is nu echter wel iets waar we rekening mee moeten houden. In tegenstelling tot onze `PaymentMethod`-class, die gebruik maakt van een default-constructor, verwacht onze `CreditCard`-class argumenten voor diens properties: `cardNumber` en `nameOnCard`. Zo hebben we onze constructor van de `CreditCard`-class immers ingechikt.

Dit betekent dat wanneer we in de toekomst instanties willen maken van een `VisaCard`, `MasterCard`- of `AmericanExpress`-class, we deze argumenten moeten doorgeven aan de constructor van de superclass (`CreditCard`). Als je nu al direct zwetende handpalmen hebt gekregen: geen zorgen! Dit is gelukkig heel simpel. Er bestaat een standaard methode die we kunnen gebruiken om de constructor van de superclass aan te roepen: `super()`.

Bij het inrichten van onze subclass `VisaCard`, moeten we diens constructor dus de argumenten laten ontvangen die de superclass nodig heeft (`cardNumber` en `nameOnCard`). Hier doen we verder niets mee, want we geven deze argumenten direct door aan de `super()`-aanroep die zorgt dat ze op de juiste plek terecht komen:

```
// VisaCard.java
class VisaCard extends CreditCard {
    public VisaCard(String nameOnCard, String cardNumber) {
        super(nameOnCard, cardNumber);
    }
}
```

Uiteraard kun je dit nog uitbreiden met properties die alleen de VisaCard-class verwacht, zoals het type van de kaart (Gold of Platinum bijvoorbeeld):

```
// VisaCard.java
class VisaCard extends CreditCard {
    private String cardType;

    public VisaCard(String nameOnCard, String cardNumber, String cardType) {
        super(nameOnCard, cardNumber);
        this.cardType = cardType;
    }
}
```

Wanneer we nu een VisaCard instantie maken, zijn we verplicht om drie argumenten mee te geven: het nummer, de naam op de kaart en het type van de kaart. De methodes die we op een VisaCard instantie kunnen gebruiken zijn getNumberOnCard (van CreditCard), setNumber (van CreditCard) en withdraw (van PaymentMethod).

```
// Main.java
public class Main {
    public static void main(String[] args) {
        VisaCard goldCard = new VisaCard("Max Verstappen", "1234 5678 9012 3456", "Gold");

        System.out.println("Deze kaart is van " + goldCard.getNameOnCard());
        goldCard.withdraw(400);
    }
}
```

Super is niet altijd nodig

Waarschijnlijk denk je nu: "Wacht eens even! Maar dan hadden we die super-aanroep in de CreditCard-class toch ook moeten maken? Die is een afstammeling van PaymentMethod!" En dat klopt ook. Maar aangezien onze PaymentMethod-class geen argumenten verwacht (deze gebruikt de ingebouwde constructor) zouden we hiervoor een lege super()-aanroep gebruiken:

```
// CreditCard.java
class CreditCard extends PaymentMethod {
    private String nameOnCard;
    private String cardNumber;

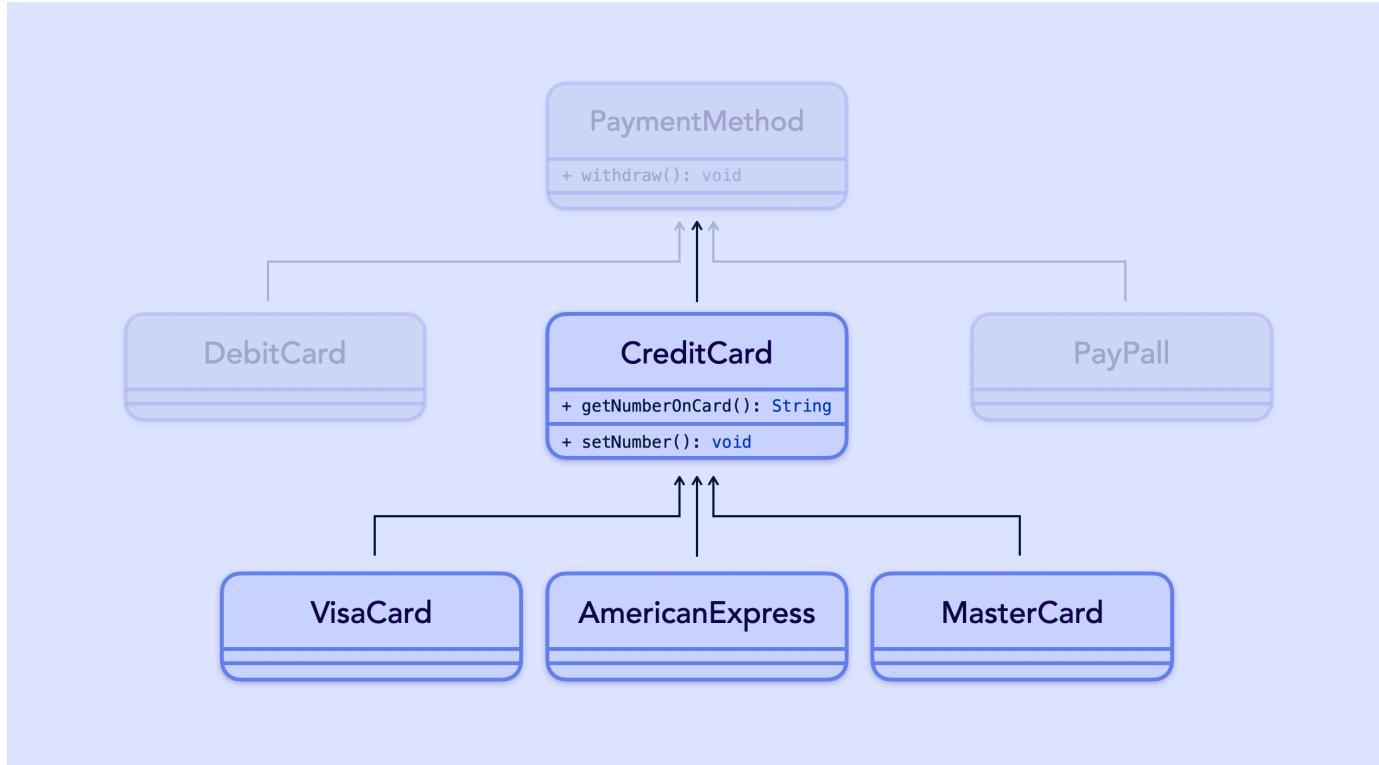
    public CreditCard(String nameOnCard, String cardNumber) {
        super();
        this.nameOnCard = nameOnCard;
        this.cardNumber = cardNumber;
    }
    // andere methoden...
}
```

Om ons als developers wat werk te besparen hoeven we lege super()-aanroepen niet persé neer te zetten: deze methode wordt namelijk standaard (onzichtbaar) aangeroepen wanneer we een afstammeling maken van een superclass. Laat je 'm in dit voorbeeld weg, dan werkt alles nog steeds. Sterker nog: het is netter om deze aanroep er niet bij te zetten, gezien 'ie de boel alleen maar vervuld.

Overerving is dus een mooie manier om, naast het gebruik van packages, aan te geven dat classes bij dezelfde categorie of groep horen. Wanneer je de declaratie van onze PayPal-class leest, zie je direct dat we deze class gebruiken als betaalmiddel. En wanneer je onze VisaCard-class leest, zie je direct dat dit een creditcard is. In de PaymentMethod-class zit dan gedrag die door alle betaalmiddelen gedeeld wordt.

5.8 Abstracte classes

Naast reguliere classes gebruiken we in Java ook wel eens **abstracte classes**. Een abstracte class is een class die niet op zichzelf kan bestaan. Hij kan enkel als basis voor andere classes gebruikt worden. Wanneer we naar onze betaalmiddelen kijken, valt het op dat onze CreditCard-class (en de DebitCard-class eigenlijk ook) op zichzelfstaand vrij nutteloos is geworden. Met de huidige opzet is het niet eens meer de bedoeling om een CreditCard-class instantie te maken! Bij het opstellen van een kaart maken we namelijk altijd een VisaCard-, AmericanExpress- of MasterCard- instantie aan die afstamt van een CreditCard.



Ondanks het feit dat we geen instanties meer van de CreditCard-class willen maken, gebruiken we deze class om een generieke set aan properties en methoden te definiëren die door meerdere classes in de applicatie kunnen worden gedeeld, zoals het wijzigen van een kaartnummer. Maar er zullen ook methoden of properties nodig zijn die bij alle soorten creditcards net een beetje anders ingericht moeten worden. Alle kaarten moeten de creditcardlimiet kunnen teruggeven, maar bij American Express tellen tijdelijke reserveringen op de kaart niet mee, terwijl bij Visa alle transacties gezien worden als vermindering op de limiet.

Omdat je van abstracte classes geen instanties mag maken, is het logisch (en veiliger) dat we onze CreditCard-class omschrijven naar een abstracte class:

```
abstract class CreditCard extends PaymentMethod {
    // ...
}
```

Naast reguliere methodes die door iedere subclass worden geërfd, kun je in een abstracte class ook **abstracte methodes** definiëren: methodes zonder body. Subclasses van de abstracte class zijn dan zelf verantwoordelijk voor de implementatie van deze methode. Hoe ze dit invullen bepalen ze zelf, maar het feit dat de methode ingevuld moet worden wordt wel verplicht door de abstracte class.

Later we ook dit eens toe passen op onze CreditCard-class. De eerdere methodes en properties die we hadden gemaakt blijven hetzelfde, maar we voegen daarbij nog een nieuwe, abstracte methode toe:

```
// CreditCard.java
abstract class CreditCard extends PaymentMethod {
    private String nameOnCard;
    private String cardNumber;

    public CreditCard(String nameOnCard, String cardNumber) {
        this.nameOnCard = nameOnCard;
        this.cardNumber = cardNumber;
    }

    public String getNumberOnCard() {
        return this.cardNumber;
    }

    public void setNumber(String cardNumber) {
        if (cardNumber != null && cardNumber.length() == 16) {
            this.cardNumber = cardNumber;
        }
    }

    public String getNameOnCard(){
        return this.nameOnCard;
    }

    // declaratie abstracte methode:
    public abstract double getCreditLimit();
}
```

Zoals je ziet hebben we hier een mooie mix tussen reguliere methoden en een abstracte methode. Dit betekent dat onze subclasses VisaCard-, AmericanExpress- en MasterCard nu verplicht zijn om een implementatie van de `getCreditLimit`-methode te bevatten. Dit zullen we in de VisaCard-class als volgt oplossen:

```
// VisaCard.java
class VisaCard extends CreditCard {
    private String cardType;
    private double creditLimit;

    public VisaCard(String nameOnCard, String cardNumber, String cardType, double creditLimit) {
        super(nameOnCard, cardNumber);
        this.cardType = cardType;
        this.creditLimit = creditLimit;
    }

    // implementatie van de abstracte methode:
    public double getCreditLimit() {
        return creditLimit;
    }
}
```

We hebben invulling gegeven aan de abstracte methode `getCreditLimit` door een private property `creditLimit` op deze class te zetten die teruggeven wordt wanneer de methode wordt aangeroepen. Daarnaast hebben we de constructor uitgebreid met een `creditLimit`, zodat deze waarde meegegeven moet worden wanneer we een nieuwe instantie maken:

```
// Main.java
public class Main {
    public static void main(String[] args) {
        VisaCard goldCard = new VisaCard("Max Verstappen", "1234 5678 9012 3456", "Gold", 2340.34);

        System.out.println("Deze kaart is van " + goldCard.getNameOnCard());
        goldCard.withdraw(400);
    }
}
```

5.9 Opdracht: overerving

In deze opdracht gaan we leren hoe we overerving kunnen gebruiken. Je kunt deze opdracht maken door het project te clonen of te downloaden naar jouw eigen computer via [deze GitHub repository](#). De uitwerkingen staan op de branch `uitwerkingen`.

5.10 Interfaces

Een speciale vorm van abstracte classes zijn **interfaces**. Een interface heeft geen properties of geïmplementeerde methodes. Het enige dat erin staat, zijn lege methodes die ingevuld moeten worden door de class die de interface implementeert. Een interface is dus als het ware een contract: een belofte dat deze class die methodes zal invullen.

Zo zouden we een interface kunnen maken die ervoor zorgt dat alle CreditCard-classes verplicht zijn om de methode `isExpired` te implementeren. Hiermee geef je als het ware een garantie aan de gebruiker (en ontwikkelaar) dat er altijd een implementatie van `isExpired` beschikbaar is. De syntax om een interface te maken en te gebruiken in de CreditCard-class, ziet er als volgt uit:

```
interface Expiring {
    boolean isExpired(int month, int year);
}

class CreditCard implements Expiring {
    public boolean isExpired(int month, int year) {
        // implementatie van deze methode...
    }
}
```

Hierbij is het belangrijk om te weten dat wanneer je een interface in een abstracte class implementeert, dat de abstracte class niet verplicht is om deze methode te implementeren. De verantwoordelijkheid ligt dan bij de subclass van CreditCard:

```
abstract class CreditCard implements Expiring {
    // geen implementatie van de isExpired-methode
}
```

Een goed voorbeeld van een interface die bepaalde functionaliteit belooft én afdwingt in de implementerende classes, is `List` in de Java standaard library: `List`. Door deze implementatie kun je erop vertrouwen dat een `ArrayList` zeker een `isEmpty()` methode zal moeten implementeren. De implementatie hangt af van de soort lijst die de interface `List` implementeert, maar deze zal gegarandeerd bestaan!

5.11 Scope

Binnen Java (en overigens bijna iedere programmeertaal) zijn onze variabelen gebonden aan **scope**. Je kunt scope zien als de context waarin variabelen beschikbaar of toegankelijk zijn.

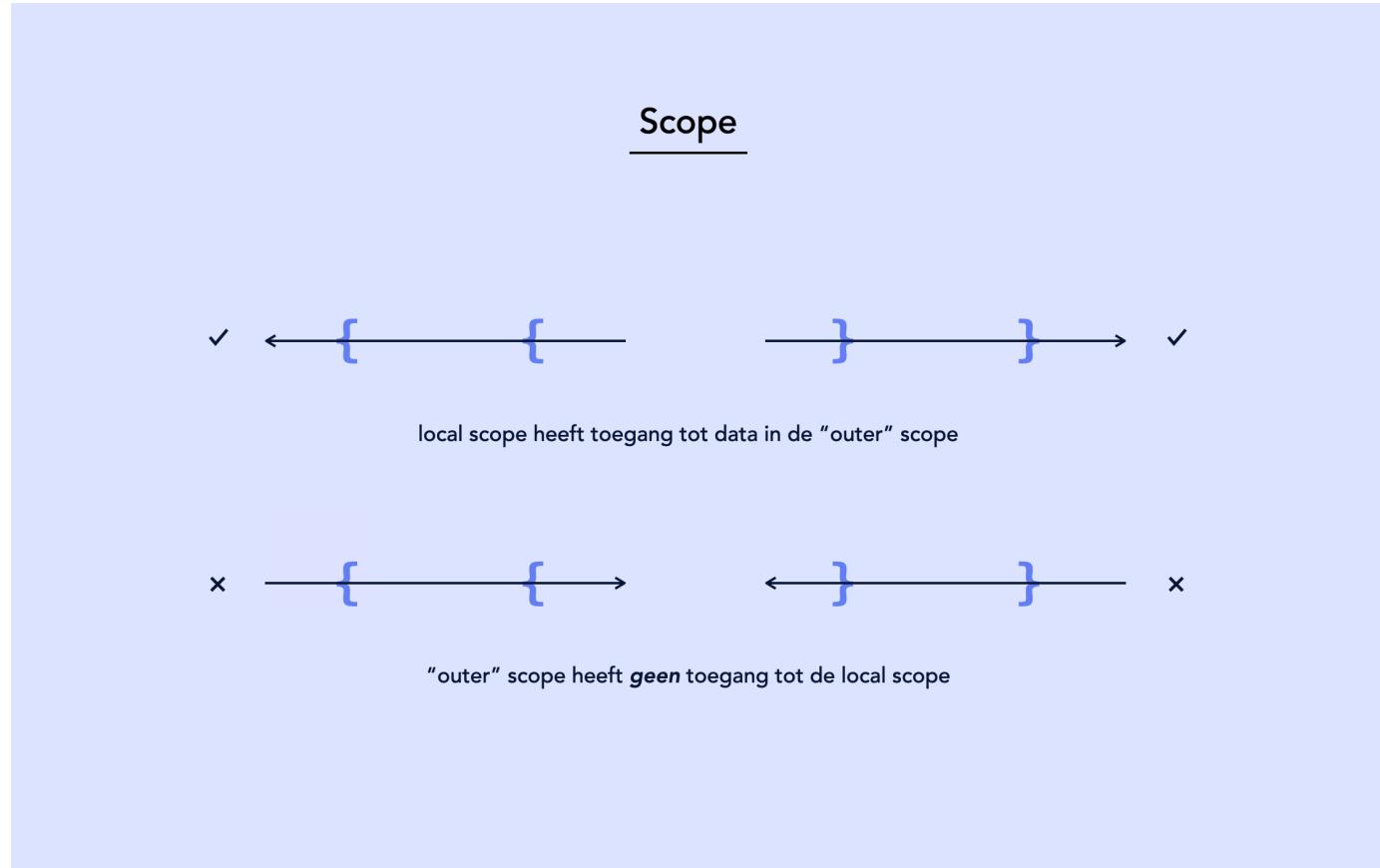
Nu denk je misschien: wat heeft het voor zin om toegankelijkheid van variabelen te beperken en niet alles overal in de code beschikbaar te hebben? Het biedt een stukje veiligheid. Je kunt het vergelijken met de "rollen" die je gebruikers binnen software kunt geven. Kijk maar naar EdHub: we hebben studenten-accounts, docenten-accounts en admin-accounts. Als student kun je geen aanpassingen maken in de content van cursussen. Docenten kunnen op hun beurt geen wachtwoorden resetten voor andere gebruikers. Dat mogen alleen de admins doen. Als iedereen op EdHub alle bevoegdheid van een admin zou hebben, is de kans veel groter dat er dingen mis gaan. Als content per ongeluk verwijderd wordt of accounts opgeheven, heb je bovenend geen idee wiens schuld dat was. Daarom zorgen we dat iedereen alleen toegang heeft tot de functionaliteit die ze nodig hebben. Dit noem je *The Principle of Least Access*.

Scoping doet dit ook. Het helpt daarom om de foutgevoeligheid van code naar beneden te brengen en het maakt het makkelijker om bugs op te sporen als ze toch opduiken. Scope is echter iets anders dan het gebruik van access modifiers. Access modifiers kunnen we zelf controleren en toepassen, maar scope is simpelweg een programmeerprincipe waar we rekening mee moeten houden.

Wanneer we variabelen declareren, zullen deze *alleen beschikbaar zijn binnen het codeblok (tussen de {}) waar deze gedeclareerd zijn*. Dit zorgt ervoor dat we namen van variabelen zonder problemen kunnen herhalen en ze zitten elkaar nooit in de weg. Het werkt als volgt:

- Declareer je een variabele in een class? Dan kun je de variabele overal in de class bereiken;
- Declareer je een variabele in een methode? Dan kun je de variabele alleen binnen de methode bereiken;
- Declareer je een variabele binnen een lus? Dan kun je die variabele alleen binnen de lus bereiken;
- Declareer je een variabele binnen een statement die {} gebruikt, zoals een if-else-statement? Dan kun je de variabele alleen binnen die statement gebruiken.

Scope beperkt toegankelijkheid dus van buiten naar binnen, maar niet van binnen naar buiten.



Laten we hier eens wat praktische voorbeelden van bekijken. Eerst zullen we kijken naar een CreditCard-class, waarin we de variabele expiresInYear hebben gedeclareerd. Deze variabele staat op het hoogste niveau (in de class) dus alle methoden en {}-statements kunnen deze variabele bereiken. De variabele die we binnen de methode isExpired hebben gedeclareerd, difference, is echter alleen beschikbaar *binnen* deze methode. Wanneer we nog een andere methode zouden schrijven, kan deze niet bij de variabele difference. Probeer je dit wel, dan krijg je een foutmelding.

```
public class CreditCard {
    private int expiresInYear;

    public boolean isExpired(int currentYear) {
        int difference = expiresInYear - currentYear;
        if (difference > 0) {
            return true;
        } else {
            return false;
        }
    }

    public void thisWontWork() {
        System.out.println(difference); // geeft een foutmelding
    }
}
```

Ditzelfde geldt wanneer we een loop schrijven. Bij het declareren van een for-loop, gebruiken we altijd een variabele die we *i* noemen. Deze *i* is echter alleen beschikbaar binnen de {} van de lus. We kunnen de *i* daarbuiten niet printen of gebruiken.

```
for (int i = 0; i < 10 ; i++) {
    System.out.println(i); // Werkt!
}

System.out.println(i); // Geeft een foutmelding!
```

Scope beperkt toegankelijkheid dus van buiten naar binnen, maar niet van binnen naar buiten.

5.12 Hiërarchie

We hebben onze methodes nu verdeeld over verschillende classes, abstract classes en interfaces om onszelf werk te besparen. Maar er is nog een andere belangrijke reden dat we een soortgelijke structuur willen aanhouden. Laten we de volgende stap nemen en zorgen dat we onze betaalmiddelen daadwerkelijk kunnen gebruiken. We maken daarvoor een Payment-class:

```
// Payment.java
class Payment {
    int amount;
    DebitCard debitCard;
    VisaCard visaCard;
```

```

AmericanExpress americanExpress;
MasterCard masterCard;
PayPal paypal;

public void setAmount(int amount) {
    this.amount = amount;
}

public void setDebitCard(DebitCard debitCard) {
    this.debitCard = debitCard;
}

public void setVisaCard(VisaCard visaCard) {
    this.visaCard = visaCard;
}

public void setAmericanExpress(AmericanExpress americanExpress) {
    this.americanExpress = americanExpress;
}

public void setMasterCard(MasterCard masterCard) {
    this.masterCard = masterCard;
}

public void setPayPal(PayPal paypal) {
    this.payPal = paypal;
}

public void submit() {
    // implementatie voor uitvoeren van de betaling
}
}

```

Deze Payment-class willen we vervolgens gebruiken in onze Main-class, maar daar lopen we tegen een probleempje aan:

```

// Main.java
public class Main {
    public static void main(String[] args) {
        Payment payment = new Payment();
        payment.setAmount(100);

        // Is het een debit card?
        payment.setDebitCard(kaart);

        // ... of Visa?
        payment.setVisaCard(kaart);

        // ... AmEx misschien?
        payment.setAmericanExpress(kaart);

        // ... Master dan?
        payment.setMasterCard(kaart);

        // ... Pay Pal?
        payment.setPayPal(kaart);
    }
}

```

Zoals je al ziet wordt het best ingewikkeld om de logica te schrijven voor de afhandeling van de betaling. Eerst moeten we gaan bepalen welke betaalmethode er is gekozen en daarna moeten we extra regels code schrijven om te bepalen welke setter er zou moeten worden aangeroepen. Hmm...

Gelukkig kunnen we dit op een betere manier implementeren, omdat we overerving hebben gebruikt in onze classes. Dit betekent dat we onze creditcard-variabele (van het super-type CreditCard) mogen vullen met een sub-type, zoals VisaCard. In code ziet dat er als volgt uit:

```

// Dit mag, omdat VisaCard in de hierarchie onder CreditCard staat
CreditCard creditCard = new VisaCard(...);

```

Andersom (een sub-type vullen met een super-type) mag absoluut niet, omdat CreditCard niet onder VisaCard staat in de hiërarchie:

```

// Dit mag niet en compileert niet. Zelfs als CreditCard niet abstract was!
VisaCard visaCard = new CreditCard(...);

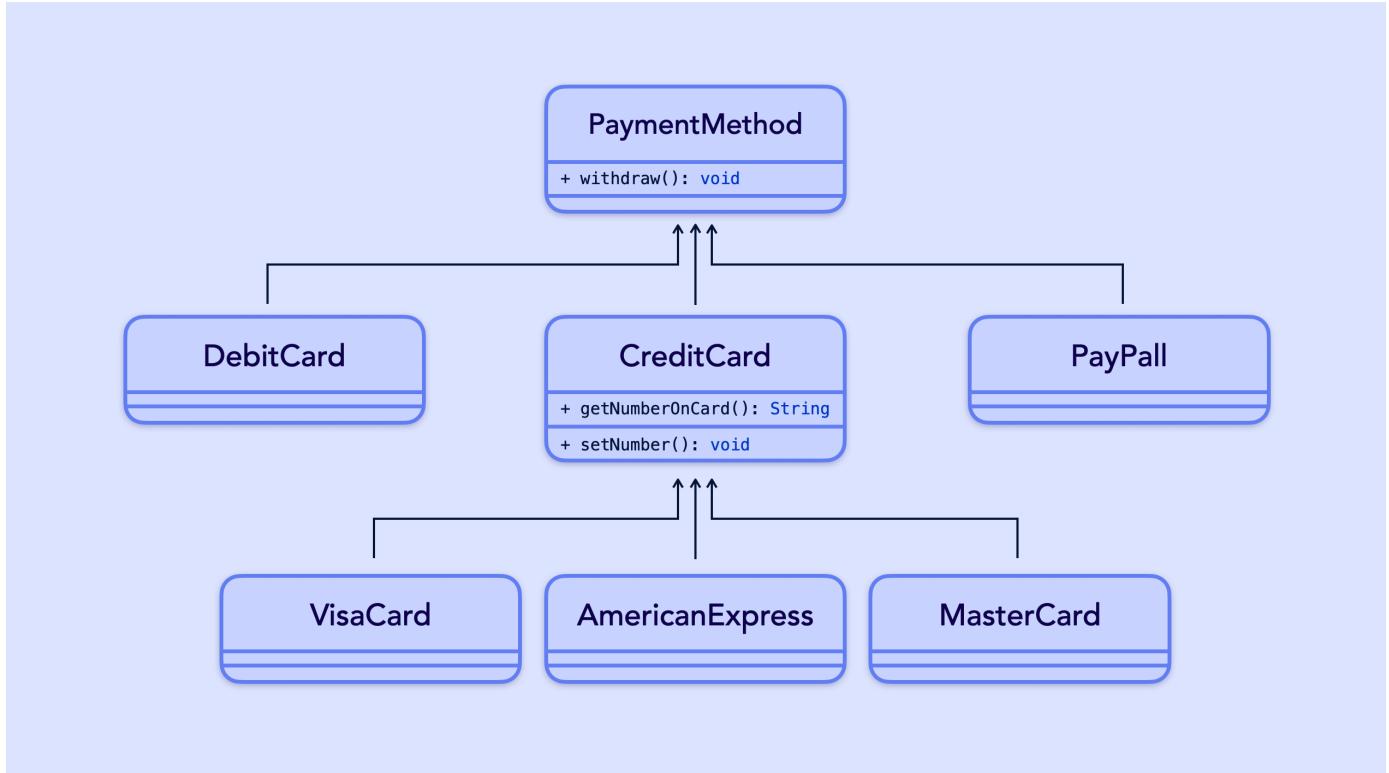
```

Iets wat ook niet mag, is het toekennen van een DebitCard aan een variabele van het type CreditCard. Dat komt omdat DebitCard en CreditCard elkaar in de hiërarchie niet kruisen:

```

CreditCard creditCard = new DebitCard(...); // Dit mag niet en compileert niet

```



We mogen dus alleen sub-types gebruiken die onder het super-type vallen. Dit beperkt zich echter niet tot één niveau, want onderstaand stukje code is ook toegestaan:

```

// Dit mag, omdat VisaCard ergens verderop in de hierarchie ook onder PaymentMethod valt
PaymentMethod paymentMethod = new VisaCard(...);

// Ook dit mag, een interface die wordt geïmplementeerd wordt ook gezien als hierarchies bovenstaand
ExpiringCard = new VisaCard(...);

// Dit mag altijd, omdat alle classes in Java altijd als bovenste class in de hierarchie Object hebben staan.
Object object = new PayPal(...);
  
```

Dus als we nu de code aanpassen, dan kunnen we onze Payment-class zo schrijven:

```

class Payment {
    int amount;
    PaymentMethod paymentMethod;

    public void setAmount(int amount) {
        this.amount = amount;
    }

    public void setPaymentMethod(PaymentMethod paymentMethod) {
        this.paymentMethod = paymentMethod;
    }

    public void submit() {
        // Implementatie voor uitvoeren van de betaling
    }
}

public class Main {
    public static void main(String[] args) {
        ... // Code waar de kaart wordt opgevoerd

        Payment payment = new Payment();
        payment.setAmount(100);
        // De aanroeper kan gewoon elke betaalmethode doorgeven en hoeft niet meer zelf uit te zoeken wat het is!
        payment.setPaymentMethod(kaart);
    }
}
  
```

Niet alleen is het nu veel gemakkelijker geworden om de betaling uit te voeren, het is ook veel gemakkelijker om vanuit de Main-class een betaalmethode te kiezen. We kunnen alle betaalmethodes namelijk gewoon via dezelfde setPaymentMethod-methode doorgeven, ongeacht of dit nu een creditcard, debitcard of PayPal-transactie is.

5.13 Generics

Hierarchie geeft ons de mogelijkheid om algemene stukjes code slechts op één plek te hoeven schrijven, omdat we properties en variabelen kunnen voorzien van generieke, hoger gelegen super-class-types. Dat is natuurlijk super handig. Sterker nog, deze manier van code schrijven is een **best practice**. Het zorgt ervoor dat we maximale flexibiliteit in onze code behouden en niet telkens alles overhoop hoeven te gooien wanneer er bijvoorbeeld een nieuw betaalmiddel bij komt. Dit is vanuit het perspectief van het declareren van een class dus heel fijn. Echter, bij het gebruiken van een class die generiek is opgesteld, kan dit principe juist weer voor problemen zorgen.

Stel dat we de Payment-class in sommige gevallen willen beperken tot het gebruik van creditcards. Bijvoorbeeld omdat we de extra verzekering die een creditcard biedt willen gebruiken en de minder verzekerde betaalmiddelen willen uitsluiten. Dit kunnen we bereiken door een subclass te maken die die CreditCard als variabel-type heeft, in plaats van PaymentMethod. Dat zou echter jammer zijn, want we hadden net alle code zo mooi hergebruikt...

Gelukkig hoeft dat dan ook niet. We kunnen dit omzeilen door gebruik te maken van **generics**. Generics stellen ons in staat om, zoals het woord al zegt, de code van de class generiek te houden en pas specifiek te maken op het moment van gebruik. Dat doen we door de class te voorzien van een speciale variabele, die tussen <> wordt gedefinieerd:

```

// we definiëren hier de generic als letter 'P', maar dit mag net als een variabele van alles zijn
class Payment<P> {
    int amount;
  
```

```
// we vervangen hier het type object met de variabele van de generic
P paymentMethod;

public void setAmount(int amount) {
    this.amount = amount;
}

// ... en hier vervangen we ook het type object met de variabele van de generic
public void setPaymentMethod(P paymentMethod) {
    this.paymentMethod = paymentMethod;
}

public void submit() {
    // implementatie voor uitvoeren van de betaling
}
```

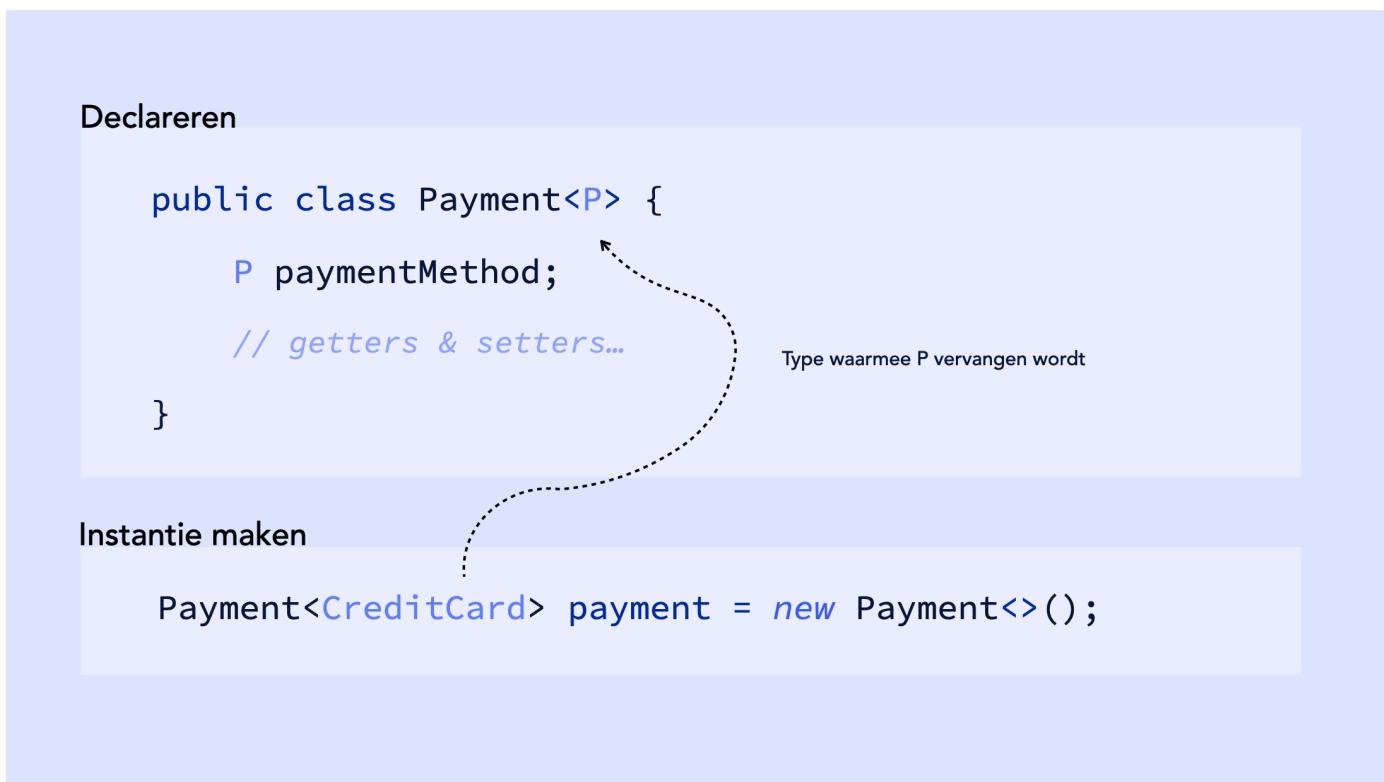
Let op: bovenstaande syntax is expres heel simpel gehouden. Generics zijn zeer uitgebreid en kunnen heel veel, meer dan past in dit hoofdstuk. In dit hoofdstuk zullen we ons daarom vooral richten op het gebruiken van generieke klassen.

Doordat we generics hebben gebruikt, hoeven we het exacte type niet meer te definiëren bij het declareren van de class. We hoeven het exacte type nu pas mee te geven wanneer we een instantie maken van de class:

```
Payment<PaymentMethod> payment = new Payment<>();
```

Zoals je ziet, zul je bij een class die generics gebruik altijd gebruik moeten maken van de <>, zowel in het type van de variabele als bij het gebruik van de constructor. Door deze schrijfwijze levert dit nu dezelfde werking op als in onze originele Payment-class. Het type dat tussen de <> wordt beschreven, wordt toegekend aan de <> in de declaratie en 'vervangt' alle P generic variabelen. Door deze syntax is het nu ook mogelijk geworden om alleen creditcards toe te staan als betaalmiddel:

```
Payment<CreditCard> payment = new Payment<>();
```



Wanneer we in deze situatie zullen proberen om het betaalmiddel in te stellen op debitcards, krijgen we een foutmelding:

```
Payment<CreditCard> payment = new Payment<>();
payment.setPayment(new DebitCard(...));
```

Een debitcard werkt nu niet meer, omdat DebitCard geen hiërarchische afgeleide is van CreditCard. We hebben nu zonder een nieuwe implementatie van Payment te maken toch een beperking kunnen realiseren. Willen we alleen PayPal toestaan? Simpel.

```
Payment<PayPal> payment = new Payment<>();
```

Zoals je ziet zijn generics een ontzettend krachtig middel om de herbruikbaarheid van onze classes te ondersteunen.

5.14 Opdracht: interfaces

In deze opdracht gaan we leren hoe we interfaces kunnen gebruiken. We gaan hier door op de vorige opdracht waar we een Customer hebben geschreven. Je mag verder werken in je eigen uitwerking, of deze opdracht clonen via [deze github repository](#).

6. Collecties

6.1 Inleiding

Je hebt immiddels kennis gemaakt met primitive datatypes, maar er bestaan ook **structurele** datatypes. Een **array** is daar één van. Een array is het best te omschrijven als een lijst, met een vaste lengte, waarin elementen van hetzelfde type worden opgeslagen. Je kunt dus stellen dat die waarden altijd iets met elkaar gemeen hebben, zoals namen, leeftijden of temperaturen.

Zo zou je bijvoorbeeld een lijst van studenten van komend startmoment willen opslaan in een array:

1. Marieke Jansen
2. Freek Hertenbos

3. Robin Toegracht
4. Meike van Maalsweert

Omdat een array gekenmerkt wordt door volgordeelheid, staan we daarmee ook direct het nummer van hun vaste stoel in het klaslokaal op. Later kunnen we dan altijd vragen wie er bijvoorbeeld op stoel 3 zit (Robin).

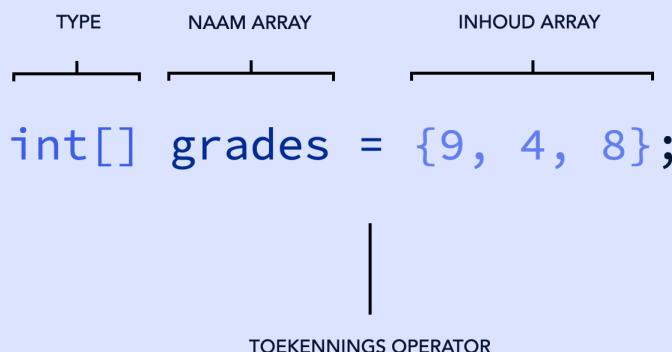
In Java worden deze "standaard" arrays niet vaak gebruikt, omdat Java andere developer-vriendelijke constructies kent om gemakkelijker met lijsten te werken. Deze constructies baseren zich in de basis wel vaak op arrays, waardoor het belangrijk is om dit concept van Java goed onder de knie te krijgen. Later in deze cursus zullen we de Collections API van Java beter gaan bekijken, waar we onder andere ArrayLists zullen tegenkomen. Het grootste verschil tussen deze twee is dat een array een statische verzameling van elementen is, met een vaste grootte. Wanneer de array eenmaal aangemaakt is, is het niet meer mogelijk deze uit te breiden. Een ArrayList is een dynamische array die meer functionaliteit bevat, zoals het toevoegen, verwijderen of sorteren van elementen.

In dit hoofdstuk behandelen we hoe we arrays kunnen maken en gebruiken.

6.2 Nieuwe array maken

Wanneer je gebruik wil maken van een array zul je deze, net als alle andere variabelen, eerst moeten declareren. Omdat in Java alle elementen in een array van hetzelfde type moeten zijn, typen we de hele array. Een int array met lengte 5 kan iedere combinatie van 5 int's bevatten, bijvoorbeeld 1,2,3,4,5 of 1,65,-8,-4,3. Een boolean array met lengte 3 kan iedere combinatie van 3 boolean's bevatten, bijvoorbeeld false,false,false of true,true,true.

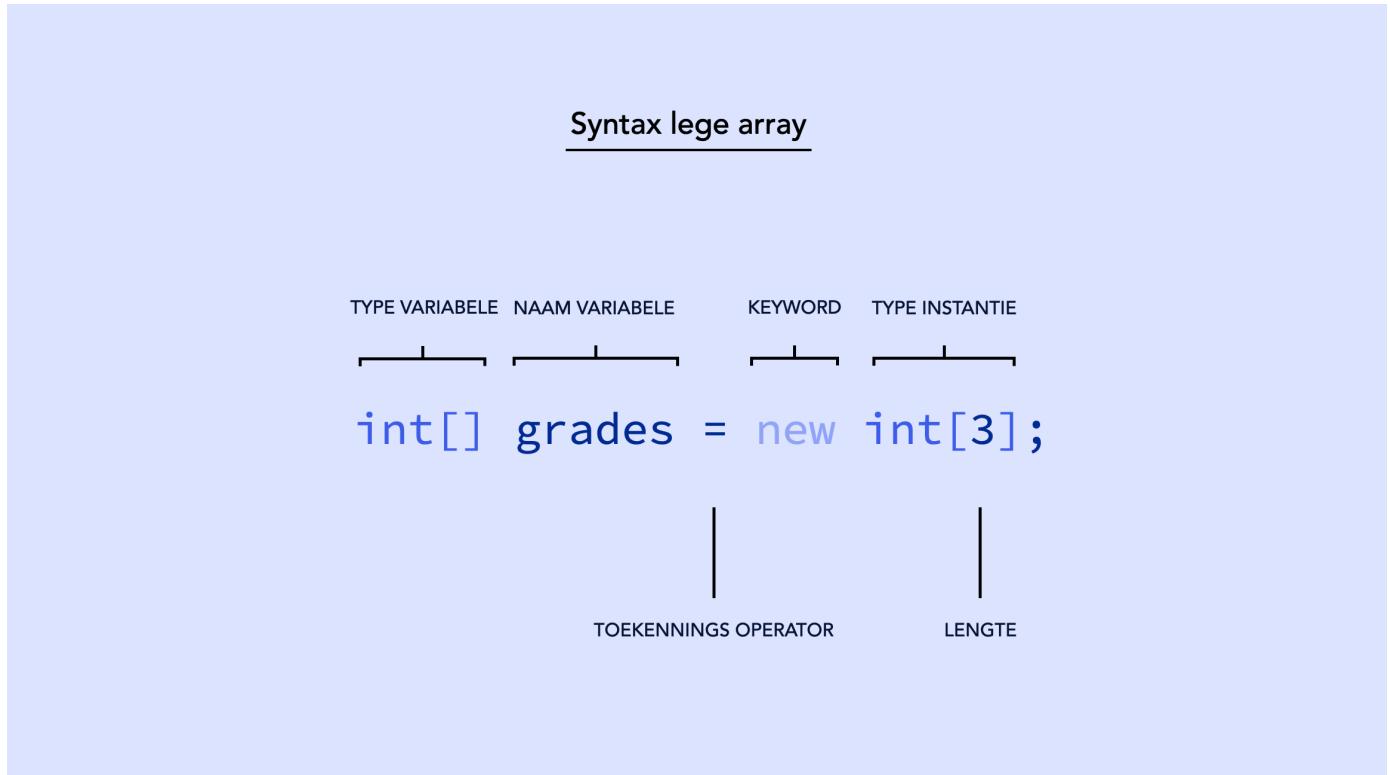
Syntax voorgedefinieerde array



We maken duidelijk dat we een array willen maken door *blokhaakjes* achter het type te plaatsen. In bovenstaand voorbeeld kondigen we aan een array van int's te willen declareren, onder de naam grades. Hiermee stellen we ook direct vast wat de lengte wordt, gezien we alle items direct invoegen. Wanneer we een array met booleans zouden willen declareren onder de naam hasGraduated, doen we dat als volgt:

```
boolean[] hasGraduated = {true, false, true};
```

Dit is echter niet de enige manier om een array te declareren. Wanneer we vooraf niet direct willen bepalen wát we in de array willen opslaan, kunnen we ook een lege array declareren. Het enige dat je wel vooraf moet vaststellen, is de lengte van de array. Dit getal zetten we tussen de blokhaken:



In bovenstaand voorbeeld kondigen we aan een array van `int`'s te willen declareren, onder de naam `grades`. Vervolgens initialiseren we een array van `int`'s, met het keyword `new`. Hoe dit keyword precies werkt, zullen we in een later hoofdstuk behandelen.

Oké, leuk, we hebben een lege array gemaakt van drie integers... En nu? Nou, de array is niet *echt* leeg. Binnen Java heeft ieder primitief datatype een standaard waarde, ook wel **default value** genoemd. Dit betekent dat als we niets toewijzen, een `boolean` bijvoorbeeld standaard op `false` wordt gezet en een `int` standaard op 0.

Datatype	Default Value
<code>byte</code>	0
<code>short</code>	0
<code>int</code>	0
<code>long</code>	0L
<code>float</code>	0.0f
<code>double</code>	0.0d
<code>char</code>	'\u0000'
<code>boolean</code>	<code>false</code>

Onze lege array van integers is dus eigenlijk een array met 0, 0, 0 die we later nog kunnen veranderen in de gewenste integers.

6.3 Array gebruiken

Stel dat we een array declareren met de namen van onze klasgenoten:

```
String[] students = {"Marieke", "Freek", "Robin", "Meike"}
```

Je hebt toegang tot waardes in een array alsof ze in een genummerde lijst zouden staan. Let wel: de nummering van arrays begint altijd bij 0. Dat komt omdat elk item in de array automatisch een nummer toegewezen krijgt die we de **index** noemen. Als we naar de index-nummers van onze `students` array zouden kijken, zou dat er zo uitzien:

Indexnummers van een array

```
String[] students = {"Marieke", "Freek", "Robin", "Meike"};
```

0	"Marieke"
1	"Freek"
2	"Robin"
3	"Meike"

```
students[1] // geeft Freek
```

We kunnen individuele items uit een array dus aanspreken aan de hand van hun index-nummer. In het geval dat we de naam van één van onze klasgenoten zouden willen printen, benoemen we de naam van de array *inclusief* het index-nummer tussen blokhaken: `students[1]`.

```
public class MyClass {
    public static void main(String[] args) {
        String[] students = {"Marieke", "Freek", "Robin", "Meike"}

        System.out.println(students[0]); // print "Marieke"
        System.out.println(students[2]); // print "Robin"
        System.out.println(students[34]); // Bestaat niet! Er wordt een foutmelding geprint en het programma crashed
    }
}
```

Hierbij is het wel belangrijk dat het index-nummer daadwerkelijk *bestaat*. Zo zal je merken dat wanneer je `students[34]` probeert te printen, het programma crashed met een `ArrayIndexOutOfBoundsException` en de melding: "Exception in thread "main" java.lang.ArrayIndexOutOfBoundsException: Index 34 out of bounds for length 4".

Naast het gebruiken van de elementen in een array is het ook mogelijk om elementen binnen een array aan te passen. Hiervoor kun je dezelfde syntax gebruiken als bij gewone variabelen. In het voorbeeld van de `students`-array kun je "Marieke" aanpassen naar "Marijke" met de volgende statement:

```
students[0] = "Marijke";
```

Wanneer je gebruik maakt van een array die bestaat uit integers, is het ook mogelijk om de increment en decrement operatoren te gebruiken indien je een waarde in de array wil verhogen of verlagen:

```
int[] ints = {1,2,3};
ints[0]++;
```

Hierdoor wordt het cijfer 1 verhoogd met 1, dus zal de array de reeks 2, 2, 3 bevatten. Let er wel op dat de lengte van de array nog steeds vast staat. Je kunt dus nog steeds geen index aanspreken die buiten de lengte van de array valt.

```
ints[3] = 4; //Exception in thread "main" java.lang.ArrayIndexOutOfBoundsException: Index 3 out of bounds for length 3
```

6.4 Arrays in combinatie met loops

De meest gebruikte toepassing van een for-loop is het bewerken of uitlezen van arrays. Dit kan in theorie ook met een while-loop, maar gezien we precies weten hoe lang de loop zich moet blijven herhalen is een for-loop hierbij praktischer.

Stel dat we meerdere arrays hebben met daarin getallenreeksen. In het geval dat we op basis van de getallenreeksen alleen de *even* getallen willen printen, zou het handig zijn om daar een methode voor te schrijven. Deze methode moet dus in staat zijn om iedere willekeurige array van getallen te ontvangen en deze getal voor getal uit te lezen. Laten we beginnen met de declaratie van onze methode:

```
void printEvenNumbers(int[] arrayOfIntegers) {
    // 1. Lees de array waarde voor waarde uit
    // 2. Is de waarde even? Print dan de waarde
    // 3. Is de waarde oneven? Print dan niets
}
```

Het type van onze methode is `void` gezien we alleen iets zullen printen, maar geen waardes teruggeven (returnen). Nu kunnen we een for-loop gaan toevoegen. We willen dat de for-loop iedere waarde van de array uitleest en bepaalt of het een *even* of *oneven* getal is. Maar we weten van tevoren niet hoeveel items onze array zal bevatten... Dus hoe gaan we dat oplossen?

Geen zorgen, daar hebben we de `length` property voor! Deze kunnen we toepassen op *iedere* array. Voor onze `students`-array zou dat `students.length` zijn (met uitskomst 4) en voor onze `grades`-array zou dat `grades.length` zijn (met uitskomst 3). In ons geval willen we de lengte van de (nog onbekende) parameter `arrayOfIntegers` gebruiken om aan te geven wanneer de for-loop mag stoppen met herhalen:

```
void printEvenNumbers(int[] arrayOfIntegers) {
    for (int i = 0; i < arrayOfIntegers.length; i++) {
        // - Is de waarde even? Print dan de waarde
        // - Is de waarde oneven? Print dan niets
    }
}
```

Onze for-loop blijft dus net zo lang itereren, totdat de `i` even groot is geworden als de lengte van onze array. En na iedere iteratie wordt de `i` met `+1` geïncrementeerd. Dat is handig, want die `i` kunnen we gebruiken als indexnummer om individuele waarden uit de array uit te lezen! Dus net zo goed als we eerder `System.out.println(students[0])` hebben gebruikt, kunnen we het letterlijke cijfer `0` nu vervangen met de waarde van `i`:

```
void printEvenNumbers(int[] arrayOfIntegers) {
    for (int i = 0; i < arrayOfIntegers.length; i++) {
        System.out.println(arrayOfIntegers[i]);
    }
}
```

Oké, we zijn er bijna. Op dit moment wordt simpelweg iedere waarde uit de array geprint, maar we willen dat alleen laten gebeuren wanneer het een *even* getal is. *Even* getallen kun je delen door 2, dus kunnen we hier `%` voor gebruiken:

```
void printEvenNumbers(int[] arrayOfIntegers) {
    for (int i = 0; i < arrayOfIntegers.length; i++) {
        if (arrayOfIntegers[i] % 2 == 0) {
            System.out.println(arrayOfIntegers[i]);
        }
    }
}
```

Ultstekend! Nu moeten we de methode nog wel aanroepen. En gezien de methode ook een array met integers verwacht, zullen we altijd een argument mee moeten geven:

```
public class MyClass {
    public static void main(String[] args) {
        int[] grades = {9, 4, 8};

        // roep onze methode aan en geef de eerder gedeclareerde grades-array mee
        printEvenNumbers(grades); // Prints 4, 8

        // roep onze methode aan en geef een nieuwe array mee
        printEvenNumbers({2, 4, 5, 9, 2}); // Prints 2, 4, 2
    }

    private static void printEvenNumbers(int[] arrayOfIntegers) {
        for (int i = 0; i < arrayOfIntegers.length; i++) {
            if (arrayOfIntegers[i] % 2 == 0) {
                System.out.println(arrayOfIntegers[i]);
            }
        }
    }
}
```

Bij onze eerste aanroep geven we de `grades`-array mee. Deze array heeft een lengte van 3, dus de iteraties zullen er als volgt uitzien:

- `arrayOfIntegers[0]` is 9, wat oneven is en niet wordt geprint;
- `arrayOfIntegers[1]` is 4, wat even is en dus wordt geprint;
- `arrayOfIntegers[2]` is 8, wat even is en dus wordt geprint.

Bij de tweede aanroep geven we een nieuwe array zonder naam mee. Deze array heeft een lengte van 5, dus de iteraties zullen er als volgt uitzien:

- `arrayOfIntegers[0]` is 2, wat even is en dus wordt geprint;
- `arrayOfIntegers[1]` is 4, wat even is en dus wordt geprint;
- `arrayOfIntegers[2]` is 5, wat oneven is en niet wordt geprint;
- `arrayOfIntegers[3]` is 9, wat oneven is en niet wordt geprint;
- `arrayOfIntegers[4]` is 2, wat even is en dus wordt geprint.

6.5 Opdracht: arrays

Je gaat oefenen met simpele array-constructies en het ophalen van waardes. Dit ga je zowel handmatig doen als door gebruik te maken van loops.

Je kunt deze opdracht maken door het project te clonen of te downloaden naar jouw eigen computer via [deze GitHub repository](#). Hier vind je ook de gedetailleerde instructies van de opdracht zelf. De uitwerkingen staan op de branch `uitwerkingen`.

6.6 Lists

Inmiddels weet je dat iedere primitive een object type tegenhanger heeft. Dit geldt ook voor de array. Een "gewone" array is een statische verzameling van elementen met een vaste grootte. Echter, in veel applicaties hebben we voornamelijk verzamelingen nodig die we kunnen sorteren, inkorten, verlengen of aanpassen. Denk aan de reacties onder een Instagram-post, producten in het winkelmandje op Coolblue.nl of de lijst met favoriete films op Netflix.

Daarom hebben we als tegenhanger van de gewone array de interface `List` - of specifieker de `ArrayList` - omdat deze in de basis hetzelfde gedrag vertoont. Een `ArrayList` is echter uitgebreid met methodes die relevant zijn voor de array, maar is niet voorzien van automatische omzetting (`auto boxing`). Dit betekent dat we een gewone array niet zomaar kunnen zetten naar een `ArrayList` (of andersom). Je moet voordat je beginnt met programmeren dus goed bedenken welk van de twee je nodig hebt - en, spoiler! - in de praktijk is dat bijna altijd een `ArrayList`.

De `ArrayList` is afgeleid van de interface `List`. `List` is, net als in het creditcard-voorbeeld uit hoofdstuk 5, een loegespistie variant van wat in de basis een verzameling van objecten is. Op diens beurt is `List` weer een subinterface van de generieke interface `Collection`. Deze interface, die we ook vaak de Collections API noemen, bevat alleen methodes die generiek zijn voor alle vormen van verzamelingen. Het heeft mogelijkheden om elementen toe te voegen, te verwijderen, te controleren of het element een object bevat en de mogelijkheid om elementen uit te lezen. `List` breidt hierop uit door, als belangrijkste onderdeel, een vaste volgorde af te dwingen. Deze functionaliteit van een vaste volgorde (index-nummers) is specifiek voor compatibiliteit met de gewone array.

Om de bruikbaarheid van een `List` (maar ook andere implementaties binnen de Collections API) zo breed mogelijk te houden, is het mogelijk om in verzamelingen gebruik te maken van objecten en classes. Deze functionaliteit is mogelijk, omdat Collection gebruik maakt van generics, waardoor het type van alle elementen in de verzameling pas gedefinieerd hoeft te worden wanneer er een nieuwe instantie wordt gemaakt. Hoe generics precies werken, leer je wanneer je later in deze cursus, wanneer het concept van classes wat beter onder de knie hebt. Op die manier wordt er ook altijd gecontroleerd of alle objecten in de verzameling van hetzelfde type zijn.

Stel dat we een boodschappentlijstje willen gaan implementeren. Ieder item op onze boodschappentlijst heeft een naam en een aantal. Hiervoor maken we eerst een `OrderItem`-class:

```
// OrderItem.java
public class OrderItem {
    private String product;
    private Integer amount;

    public OrderItem(String product, Integer amount) {
        this.product = product;
        this.amount = amount;
    }

    public String getProduct() {
        return product;
    }
    public Integer getAmount() {
        return amount;
    }
}
```

We kunnen natuurlijk een gewone array gebruiken als we weten hoeveel items we gaan ordenen, maar als het lijstje moet kunnen groeien (en wellicht krimpen) dus dan is een `List`, en dan specifiek een `ArrayList` een betere keuze. Omdat deze types gebruik maken van generics, is de syntax voor het maken van een nieuwe `ArrayList` als volgt:

```
List <OrderItem> shoppingList = new ArrayList<>();
```

We willen onze `List` vullen met objecten van het type `OrderItem` en we maken een nieuwe instantie van onze lijst door de constructor van het sub-type `ArrayList` te gebruiken. We hoeven hier niet aan te geven of vast te leggen hoe lang de lijst wordt, we kunnen op ieder gewenst moment elementen toevoegen aan de lijst via de `add`-methode:

```
shoppingList.add(new OrderItem("stokbrood", 3));
```

Wanneer we op de hoogte zijn van het index-nummer waarop onze stokbroden zijn ingevoegd, kunnen we deze weer oproepen met de bekende syntax:

```
shoppingList.get(0);
```

... en natuurlijk ook weer verwijderen:

```
OrderItem stokbrood = new OrderItem("stokbrood", 3);
shoppingList.add(stokbrood);
shoppingList.remove(stokbrood);
```

Iedere `List` - en dus `ArrayList` - is `iterabel`, wat ervoor zorgt dat wanneer we geen idee hebben op welke plek onze stokbroden staan, we deze terug kunnen vinden met behulp van een for-loop:

```
// Main.java
public class Main {
    public static void main(String[] args) {
        List<OrderItem> shoppingList = new ArrayList<>();
        shoppingList.add(new OrderItem("stokbrood", 3));
        shoppingList.add(new OrderItem("paprika", 1));
        for (OrderItem orderItem : shoppingList) {
            if (orderItem.getProduct().equals("stokbrood")) {
                System.out.println("Je moet " + orderItem.getNumber() + " stokbroden halen");
            }
        }
    }
}
```

6.7 Maps en Sets

Map

Je snapt inmiddels wel dat een `list` ontzettend handig is in gebruik. Een nadeel van deze interface is echter dat je geen snelle toegang hebt tot de objecten in de verzameling. Je zult door de hele lijst moeten itereren om een object te vinden dat voldoet aan het door jou opgegeven criterium (zoals "Stokbrood"). Dit gaat best goed voor beschrijvende lijsten, maar als we alle creditcards van een Nederlandse bank moeten doorzoeken... *you get the point*.

In het geval dat het belangrijk is om snel te kunnen zoeken, kun je in plaats van een `List` beter een `Map` gebruiken. Een `Map` gebruikt, in tegenstelling tot `List`, geen volgordeelheid of index-nummers. Een `Map` gebruikt de unieke de hash code van een object als sleutel om het object mee terug te vinden. In de praktijk gebruiken we hiervoor `HashMap`, de implementatie van `Map`:

```
Map<String, CreditCard> creditCards = new HashMap<>();
```

Zoals je ziet worden hier niet één, maar twee types gebruikt: een type voor de hash code en een type voor de daadwerkelijke objecten die in de lijst komen te staan. Wanneer we elementen willen toevoegen aan een `HashMap` gebruiken we daar niet de `add`- maar de `put`-methode voor. *Let op:* de unieke sleutel die je nodig hebt voor ieder element in de lijst zal je zelf moeten kiezen. Kies daarom altijd iets wat uniek is, zoals bijvoorbeeld iemands creditcard- of klant-nummer:

```
VisaCard firstCard = new VisaCard("Max", "1234...", "Gold", 0.00);
creditCards.put(firstCard.getNumberOnCard(), firstCard);
```

```
// Main.java
public class Main {
    public static void main(String[] args) {
        Map<String, CreditCard> creditCards = new HashMap<>();
        VisaCard firstCard = new VisaCard("Max", "1234...", "Gold", 0.00);
        VisaCard secondCard = new VisaCard("Jos", "2276...", "Platinum", 0.00);
        creditCards.put(firstCard.getNumberOnCard(), firstCard);
        creditCards.put(secondCard.getNumberOnCard(), secondCard);
        // etc.
        if (creditCards.containsKey("1234...")) {
            System.out.println("Credit card met nummer 1234... is van " + creditCards.get("1234...").getNameOnCard());
        }
    }
}
```

De zoekactie naar een bepaald creditcard-nummer zal zeer snel zijn. Dit komt door de eigenschappen van de `String` hascodes en de eigenschappen van de `HashMap` implementatie. Hierbij is het wel belangrijk dat de sleutels altijd uniek zijn: een `HashMap` kan niet twee keer dezelfde sleutel bevatten.

Set

Naast `Lists` en `Maps` heb je dan ook nog... `Sets`! Je kunt een `Set` gebruiken wanneer je absolut geen dubbele waarden in de verzameling wil hebben. Denk bijvoorbeeld aan een lijst met gebruikersnamen voor een social media platform: je wil geen twee gebruikers met dezelfde gebruikersnaam hebben, anders weten we niet meer wie wie is. In zo'n geval is het handig om een `Set` te gebruiken - en dan specifiek de implementatie `HashSet`:

```
Set<String> gebruikersnamen = new HashSet<>();
```

In onderstaand voorbeeld kun je zien hoe we proberen om twee keer dezelfde gebruiker (`novi_de_best`) toe te voegen aan onze set. Als we echter alle gebruikersnamen uit de `Set` printen, zien we slechts één keer "`novi_de_best`" in de output staan. Is je al iets opgevallen aan de volgorde van de lijst...?

```
public class Main {
    public static void main(String[] args) {
        Set<String> gebruikersnamen = new HashSet<>();
        gebruikersnamen.add("novi_de_best");
        gebruikersnamen.add("master1337");
        // etc.
        gebruikersnamen.add("novi_de_best");
        gebruikersnamen.add("BuurmanBob");
        for(String gebruikersnaam : gebruikersnamen){
```

```
        }  
    }  
}
```

Het moeite van het gebruik van een Set, is dat er alleen waarde waardes in kunnen staan. Het nadeel is echter dat er geen volgorde wordt aangehouden, dus waarden worden niet opgeslagen in de volgorde dat ze zijn toegevoegd. Zodoende is het ook niet mogelijk om een `get()`-methode aan te roepen op een Set. Om waardes uit te lezen, zit je altijd voor een `for-loop` moeten gebruiken zoals hierboven gedaemonstreerd is.

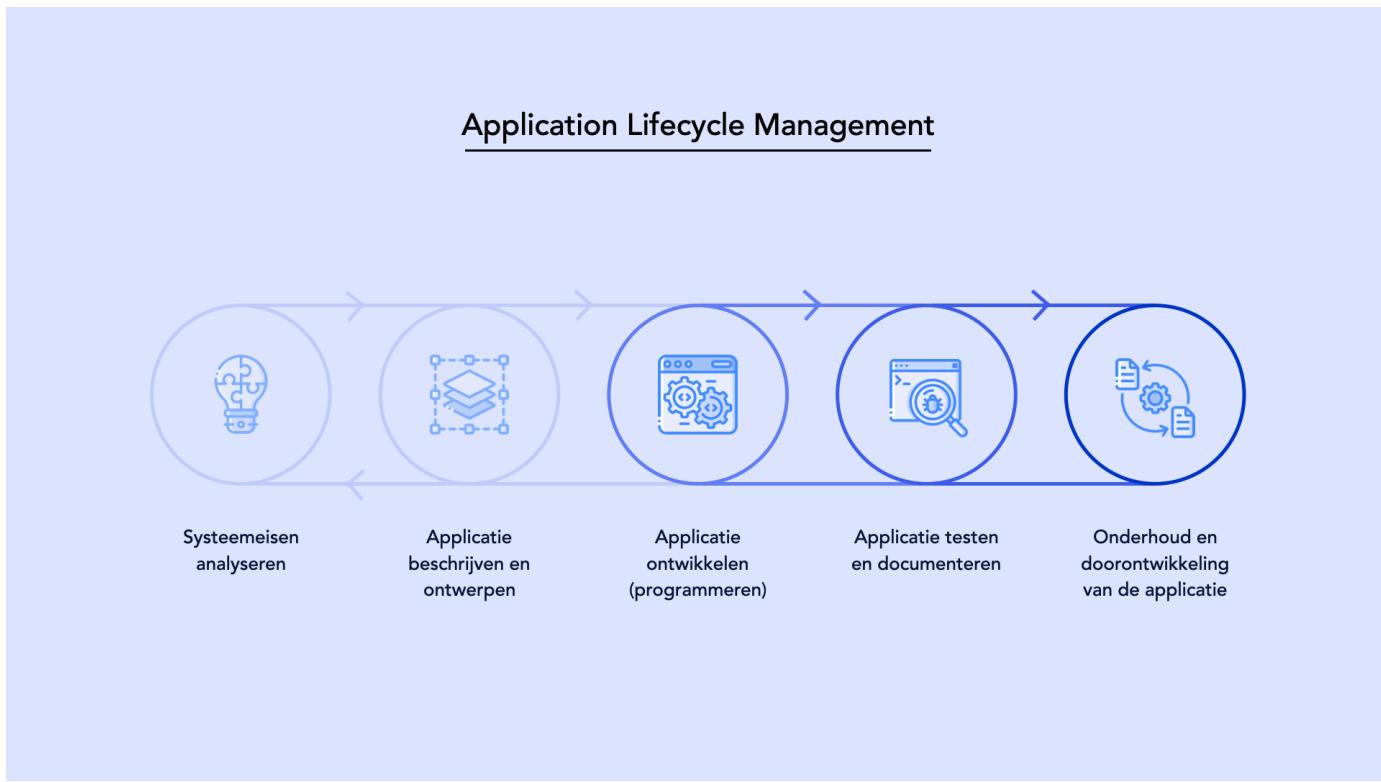
7. Build tooling

7.1 Build tools en dependencies

Hoewel je tijdens deze leerlijn nog voornamelijk in Jeentje aan projecten werk, werk je in het bedrijfsleven eigenlijk zelden alleen. In de praktijk ontwikkel je software samen met meerdere collega-developers in een team - en vaak ook nog in samenwerking met andere development-teams! Om die onderlinge samenwerking mogelijk te maken, werken ween met een breed scala aan tools:

1. **Communicate:** Slack, Teams, Meet, Zoom, etc.
 2. **Versiebeheer:** Github, Gitlab, Bitbucket, etc.
 3. **Operate:** Amazon Web Services, Google Cloud, Microsoft Azure, Oracle Cloud, IBM Cloud, etc.
 4. **Integrate:** Jenkins, Tekton, CircleCI, Github Actions, Gitlab Runners, etc.
 5. **Deployment:** ArgoCD, Ansible, Terraform, etc.

Om al deze processen te integreren gebruiken we **Build Tooling**. Build tooling is een set aan tools die alle bovengenoemde randprocessen voor ons voor ons automatisiert. Dit zorgt er niet alleen voor dat *jij* als developer kunt focussen op het belangrijkste - het bouwen van de software! - maar ook dat dit bij iedere developer op dezelfde manier gebeurt. Dankzij build tooling kunnen we onze code namelijk met simpele commando's **compilieren** (draaien), **verpakken** (uitvoerbaar maken voor andere systemen) en **implementeren** (in gebruik nemen) in verschillende omgevingen. Build tooling speelt dus een hele grote rol: *kijk maar naar onze Application Lifecycle!*



In drie van de vijf fases worden we door build tooling ondersteunt. Super belangrijk dus! Naast automatisering gebruiken we build tooling óók nog voor iets anders. We beheren er de **dependencies** in ons project mee. Maar voor we daarin duiken: wat zijn dependencies eigenlijk?

Technisch gezien is het mogelijk om een volledige webapplicatie te bouwen met alleen jouw eigen Java code. Maar... Je hebt wel uitzonderlijk veel geduld en uitdoughing vereist omdat voor ons zijn geweest die dezelfde oplossingen voor dezelfde problemen hebben gescrept. Dat is natuurlijk niet de bedoeling want van onze侧 Jaafraam staat het internet tegenwoordig wel met Open Source Code die specifieke functionaliteit aansleut, zoals de verzending en ontvangst van e-mails of het integreren van een database.

Om deze code in ons eigen project te gebruiken, hoeven we dit gelukkig niet handmatig te kopiëren-en-plakken. We kunnen het simpelweg installeren en toepassen in ons project, omdat de code beschikbaar is gemaakt in een **library**. Dat klinkt heel fancy, maar een library is niets meer dan een gesloten bundeltje Java code. Dit bundeltje kan één regel code tot tientallen classes bevatten. Die classes zijn gecomprimeerd, ofwel ingepakt, tot een JAR: een Java ARchive. Dus net zoals je de vakantiefoto's voor je zus in een ZIP-je inpakt voor je ze naar haar maalt, wordt de Java-code ingepakt in een JAR zodat libraries gemakkelijker te verspreiden zijn.

Wanneer we besloten om zo'n library te gebruiken, is onze applicatie vanaf dat moment afhankelijk van dat stukje code. Geen reden tot zorgen, dit is heel gebruikelijk! We noemen zo'n library, nadat we dit hebben geïntegreerd in ons project, daarom een **dependency**. En om het installeren en beheren van al deze dependences te gemakkelijk mogelijk te houden, gebruiken we daarvoor **npm** en **node.js**.

Wanneer je een build tool voor jouw Java project wil gebruiken heb je verschillende smaken, zoals Gradle, SBT en Apache Maven. Wij zullen gebruik gaan maken van [Apache Maven](#).

We gebruiken Maven om buildprocessen te automatiseren en onze dependencies te beheren.

Wanneer je in IntelliJ werkt, hoef je niets speciaals doen om gebruik te maken van Maven. Maven wordt namelijk standaard meegeleverd. Werk je met een andere IDE? Dan zul je Maven nog even handmatig op je besturingssysteem moeten installeren.

In veel projecten is het echter gebruiklijker om een **Maven Wrapper** te gebruiken: dit is een verpakte versie van Maven die samen met het project wordt meegeleverd. Dit zorgt ervoor dat het project altijd met dezelfde versie van Maven wordt opgebouwd, ongeacht welke versie aanwezig is op jouw lokale besturingsysteem. Dit biedt een stukje veiligheid en consistentie! Je kunt er vanuitgaan dat de opdrachten die je op EdHub vindt, allemaal voorzien zijn van een Maven Wrapper.

7.2 Maven in gebruik

van Maven begin je gebruik maken van jouw eigen versie van Maven, begin je gebruik maken van Maven. Toen we dat doormelder van commando's die we in de terminal uitvoeren. Hierbij is het belangrijk om onderscheid te maken tussen een project met een lokale versie van Maven, of een project met een meegeleverde Maven-wrappet. Wanneer je gebruik maakt van jouw eigen versie van Maven, begin je gebruik maken met:

三

Dit commando doet zelf nog niets, maar wordt aangevuld met een actie. Dit zul je later in deze paragraaf tegenkomen.

Indien je gebruik maakt van een Maven wrapper is het belangrijk het commando aan te passen op basis van het besturingssysteem van jouw computer. Dan begint ieder commando met

// Mac OS en Linux:
./mynw

77 Windows 7

mvnw.cmd

Het basiscommando kunnen we vervolgens uitbreiden om verschillende soorten acties uit te voeren met Maven. Voor we verder in deze commando's duiken, is het belangrijk om de verschillen tussen onderstaande begrippen goed te begrijpen. Wanneer we het hebben over het **compilieren** van de code, betekent dit enkel het omzetten van de door ons geschreven code naar code die uitvoerbaar is voor de JVM (Java Virtual Machine). Hierbij worden verschillende target-bestanden gecreëerd. Dit is niet hetzelfde als het **runnen** van de code. Wanneer we de code runnen, betekent dit dat de omgezette code uit de target-map wordt uitgevoerd door de JVM en het project "draai". Ten slotte spreken we vaak over het **builden** van een project: hiermee wordt het gehele proces van compileren, testen, verpakken en deployen bedoeld.

Laten we eens kijken naar de individuele commando's:

- validate wordt gebruikt om de pom.xml op structuurfouten te controleren. Hierover later meer;
- compile wordt gebruikt om alle Java classes in het project om te zetten naar JVM bytecode;
- test wordt gebruikt om alle unittests in het project uit te voeren. Indien compileren nodig is, wordt die taak eerst uitgevoerd;
- package wordt gebruikt om het product in een JAR of een ander formaat (zoals in de pom.xml aangegeven met <packaging>...</packaging>, hierover later meer) te verpakken;
- verify zorgt voor validatie van de Maven configuratie, het compileren van de code én het uitvoeren van alle unittests in het project;
- deploy om het geteste, gerecgonpleteerde en verpakte product (de *build*) naar een GitHub repository te sturen. In de meeste gevallen zorgt dit er dan ook voor dat het project online komt te staan voor de eindgebruiker, vanwege externe koppelingen met GitHub.
- clean wordt gebruikt om de target-map, waar onder andere alle omgezette JVM bytecode wordt geplaatst, helemaal leeg te maken zodat je met een schone lei kunt beginnen. Dit wordt over het algemeen alleen gebruikt wanneer je vindt dat de resultaten van het verify commando niet naar verwachting zijn;

Maven geeft ons dus een hoop mogelijkheden die we samen met het basiscommando kunnen gebruiken in de terminal. Om een voorbeeld te geven, het compileren van de code doen we met een lokale versie van Maven als volgt:

mvn package

Let op: wanneer je Maven als onderdeel van IntelliJ gebruikt (je hebt het dus niet handmatig op je OS geïnstalleerd) druk je niet op Enter, maar op CTRL + Enter (Windows) of CMD + Enter (Mac). Als je alleen op Enter drukt, zal dit commando niet herkend worden.

Naast de basisprocessen is het ook mogelijk om onze commando's te voorzien van [extra opties](#) of configuraties. Dit zijn commando's die we aan onze standaard commando's vast kunnen plakken:

Commando	Betekenis	Afkorting
--threads N	Gebruik maximaal n CPU cores voor de bouw-acties (hierbij moet n vervangen worden door een getal)	-T N
--batch-mode	Gebruik geen kleuren in de uitvoer in de terminal	-B
--fail-fast	Stop met builden wanneer er een fout geconstateerd wordt	-ff
--fail-at-end	Stop niet met bouwen mochten er fouten geconstateerd worden	-fae
--activate-profiles	gebruik één of meer (komma gescheiden) profielen, zoals gedefinieerd in de pom.xml en/of settings.xml	-P
--version	Toon de versie van Maven, JDK en het OS. Indien je andere commando's en opties gebruikt in combinatie met deze optie, worden deze genegeerd.	-v
--show-version	Toon de versie van Maven, JDK en het Operating System en voer daarna de gespecificeerde bouw doelen uit	-V
--quiet	Toon alleen mogelijke foutmeldingen tijdens het bouwen, geen andere informatie	-q
--update-snapshots	Check de repositories op nieuwe versies van de dependencies	-U

Op deze manier kunnen we tijdens het compilieren bijvoorbeeld aangeven dat we geen kleuren, maar wel de Maven-, JDK- en Operating System-versies willen zien in de terminal:

```
mvn compile --batch-mode --show-version
```

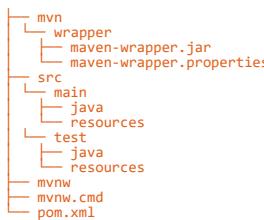
De volgorde waarop je de opties achter het commando plakt, is niet van belang. Indien het voor jouw project belangrijk is om sommige opties altijd mee te geven, is het beter om deze in een configuratiebestand te plaatsen zodat ze automatisch worden meegenomen. Dit doe je normaliter in een maven.config in de .mvn map.

7.3 Configuratie

Je hebt inmiddels gelezen dat Maven een ontzettend krachtige tool is om onze projecten mee te bouwen en te beheren. Echter, de manier waarop Maven een project moet compileren, testen en deployen komt niet zomaar uit de lucht vallen. Hiervoor gebruikt Maven altijd een **POM-bestand**. POM staat voor Project Object Model en is een soort configuratie-bestand waarin alle instellingen en informatie over het project staan op geslagen. Zonder POM in jouw project zou Maven geen idee hebben wat er zou moeten gebeuren wanneer je vraagt om het project te compileren. Het POM-bestand moet daarom altijd op root-niveau in jouw project aanwezig zijn:



Indien er een Maven Wrapper aanwezig is, voegt deze wrapper een tweetal uitvoerbare bestanden én een mvn-map toe aan het project. Deze zul je in de huiswerkopdrachten al vaker zijn tegengekomen:



Een POM heeft de bestandsextensie **.xml**. XML is een mark-up taal die wordt gebruikt om data op te slaan en te delen. Het staat voor Extensible Markup Language en is ontworpen om gegevens te verzenden en te ontvangen tussen verschillende systemen. Het maakt gebruik van tags die verschillende objecten en data

identificeren en lijkt een beetje om het welbekende **HTML: Hypertext Markup Language**.

De meest minimale pom.xml bevat alleen wat algemene informatie over het project:

```
<project xmlns="http://maven.apache.org/POM/4.0.0"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemalocation="http://maven.apache.org/POM/4.0.0 https://maven.apache.org/xsd/maven-4.0.0.xsd">

<modelVersion>4.0.0</modelVersion>

<groupId>group-id</groupId>
<artifactId>project-id</artifactId>
<version>1.0</version>
</project>
```

Zoals je kunt zien is deze informatie ingesloten in tags. In de <version>-tag staat bijvoorbeeld wat de versie van ons project is, en in de <modelVersion>-tag staat welke versie van Maven gebruikt wordt in dit project. De onderdelen <groupId>, <artifactId> en <version> zijn verplichte onderdelen in een POM. Dit noem je daarom ook wel de **coördinaten** van een Maven project.

Wanneer je jouw project beschikbaar zou willen maken als open source library, zul je het project moeten deployen naar de officiële Maven repository. De coördinaten in jouw pom.xml zorgen ervoor dat het **artifact** - het opgeleverde project - vindbaar is tussen alle andere open source projecten die Maven rijk is.*

Een normaal POM-bestand is echter altijd een stuk uitgebreider. Al deze onderdelen zullen we in de volgende paragrafen stapsgewijs toevoegen.

*Voor de eindopdracht hoeft het artifact uiteraard niet gepubliceerd te worden naar een Maven repository.

7.4 POM: projectinformatie

In het eerste gedeelte van de POM staat normaliter alle informatie over het project. We noemen onze projecten ook wel producten, gezien de bedoeling van deployment is dat eindgebruikers gebruik kunnen maken van onze applicatie. Bij de projectinformatie benoem je normaliter de naam van het product met een korte beschrijving, het webadres waarop het product draait en het oprichtingsjaar, om een indicatie te geven van hoe oud het product is:

```
<project xmlns="http://maven.apache.org/POM/4.0.0"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemalocation="http://maven.apache.org/POM/4.0.0 https://maven.apache.org/xsd/maven-4.0.0.xsd">

<modelVersion>4.0.0</modelVersion>

<groupId>n1.novi</groupId>
<artifactId>eindopdracht</artifactId>
<version>1.2.4</version>
<packaging>jar</packaging>
<name>Mijn eindopdracht product</name>
<description>Een applicatie die aan alle verwachtingen van de Integrale eindtoets Backend voldoet!</description>
<url>https://www.eindopdracht-novi.nl</url>
<inceptionYear>2023</inceptionYear>

</project>
```

In de <packaging>-tag geef je aan welke vorm van packaging gebruikt wordt. Hierin heb je de keus tussen jar, war, ear, pom, rar, maven-plugin of ejb, maar de standaard is JAR. Hier zullen we tijdens deze leerlijn dan ook altijd gebruik van maken.

Gedeelte informatie

Hoewel je hier tijdens de leerlijn nog weinig mee te maken krijgt, komt het in de praktijk vaak voor dat een bedrijf meerdere kleine producten maakt die onderdeel zijn van één overkoepelend product. Dit noem je ook wel **microservices**. Een goed voorbeeld daarvan is Netflix.

Zo maakt de Netflix applicatie zoals jij 'm kent gebruik van een ander product dat verantwoordelijk is voor de opslag van alle films en series op het platform. Deze microservice beheert de database en geeft de applicatie toegang tot deze resources. Zo hebben ze ook een product dat enkel en alleen berekent welke andere films je leuk zou kunnen vinden als je The Hangover en We're The Millers hebt gekozen. Zo bestaat het product Netflix dus uit tientallen microservices (al merkt de eindgebruiker daar natuurlijk niets van).

Je kunt je voorstellen dat veel van de POM-instellingen voor die microservices hetzelfde zullen zijn. Daarom is het mogelijk om een **parent** op te nemen in de POM die allerlei configuraties importeert zodat je deze niet telkens opnieuw hoeft te schrijven. De POM van de microservice die films voorstelt zou er dan zo uit kunnen zien:

```
<project xmlns="http://maven.apache.org/POM/4.0.0"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemalocation="http://maven.apache.org/POM/4.0.0 https://maven.apache.org/xsd/maven-4.0.0.xsd">

<modelVersion>4.0.0</modelVersion>

<parent>
  <groupId>com.netflix</groupId>
  <artifactId>shared</artifactId>
  <version>1.2.5</version>
</parent>

<artifactId>recommendation-engine</artifactId>
<version>1.2.4</version>
</project>
```

Omdat we nu alle instellingen overnemen uit de POM van onze denkbeeldige Netflix parent, hoeven we zelf bijna niets meer toe te voegen. Wel is het mogelijk om alle configuraties die we overnemen uit deze parent te overschrijven. Dit hebben we dan ook gedaan voor de <artifactId>- en <version>-tag, gezien deze natuurlijk wel uniek zijn voor deze specifieke microservice.

Anderzins zou je ook via de parent kunnen aangeven dat je deze POM-configuratie wil delen met verschillende sub-projecten of microservices. Dit doen we middels de <modules>-tag:

```
<project xmlns="http://maven.apache.org/POM/4.0.0"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemalocation="http://maven.apache.org/POM/4.0.0 https://maven.apache.org/xsd/maven-4.0.0.xsd">

<modelVersion>4.0.0</modelVersion>

<groupId>com.netflix</groupId>
<artifactId>parent</artifactId>
<version>2.0</version>
<packaging>pom</packaging>

<modules>
  <module>recommendation-engine</module>
  <module>data-storage</module>
  <module>content-encoder</module>
</modules>
</project>
```

Hoewel je dit tijdens de leerlijn niet nodig zult hebben, hebben we dit onderdeel toch behandeld gezien je dit in het werkveld nog vaak zult tegenkomen.

7.5 POM: dependencies

Wanneer we libraries willen toevoegen aan ons project hoeven we hier gelukkig niet het internet voor af te struinen. Maven beheert alle libraries op één centrale verzamelplek: **Maven Central**. Dit is een repository waarin de open-source Java-projecten worden bewaard. Het wordt door veel ontwikkelaars gebruikt om hun projecten en libraries te publiceren en te delen met anderen. Ook jij kunt een library bouwen en deze toevoegen aan Maven Central!

Wanneer we een library aan ons project willen toevoegen als dependency, zorgt Maven ervoor dat die library uit de Maven Central repository gedownload wordt wanneer je deze toevoegt aan jouw pom.xml. Daarom zijn die coördinaten zo belangrijk: dit is de enige manier waarop we verschillende libraries (artifacts) van elkaar kunnen onderscheiden in die gigantische verzameling!

Wanneer je hebt besloten een dependency te willen gebruiken, zet je deze tussen de `<dependencies>`-tag in de POM. Ieder `<dependency>`-element moet minimaal de volgende gegevens bevatten:

- `groupId` (het bedrijf of de persoon die de library maakt heeft)
- `artifactId` (de naam van de library)
- `versienummer` (wel je de meest recente versie of eenige van twee jaar terug?)

```
<project xmlns="http://maven.apache.org/POM/4.0.0"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 https://maven.apache.org/xsd/maven-4.0.0.xsd">

<modelVersion>4.0.0</modelVersion>
<!-- De projectinformatie staat hier.. -->
<dependencies>
  <dependency>
    <groupId>nl.novi</groupId>
    <artifactId>bibliotheek</artifactId>
    <version>1.3.5</version>
    <scope>test</scope>
    <type>jar</type>
    <optional>false</optional>
  </dependency>
</dependencies>
</project>
```

Het is echter ook mogelijk om de `<scope>`, `<type>` en `<optional>`-tag toe te voegen. Laten we hier eens wat dieper op ingaan.

Scope

De `<scope>`-tag geeft aan binnen welke context we deze dependency willen gebruiken. Dit heeft allemaal te maken met de lifecylefasen van Maven. De verschillende soorten scopes zijn:

- `Compile`: dependencies met deze scope zijn in alle scopes beschikbaar;
- `Test`: deze dependency is alleen beschikbaar tijdens het uitvoeren van unittests. Een goed voorbeeld daarvan is de JUnit library, die wij ook zullen gaan gebruiken.
- `Runtime`: deze dependency is alleen beschikbaar tijdens het uitvoeren van de applicatie. En dan niet als we ons project zelf op onze laptop draaien, maar wanneer het product gedeployt is en op een server draait. "In productie", noemen we dit ook wel. Een voorbeeld hiervan is een library die alle uitgaande en inkomende API's logs, zodat wij als developers kunnen weet welke endpoints van onze API bibliotheek het populairst zijn.
- `Provided`: deze dependency wordt via een andere weg beschikbaar gemaakt, bijvoorbeeld via het project dat deze provided dependency als parent heeft.
- `System`: vrijwel identiek aan `provided`, maar nu wordt er vanuit gegaan dat de JAR altijd beschikbaar is, zonder download vanaf een repository. Wordt deze scope gebruikt, dan moet ook een `systemPath` opgegeven worden!

Type

Niet als wij de mogelijkheid hebben ons project in verschillende formaten te verpakken met de `<packaging>`-tag, kunnen we met de `<type>`-tag aangeven in welk bestandsformaat we de library willen gebruiken. Wanneer we geen type opgeven, wordt er aangenomen dat je een JAR (.jar) wil hebben. Dit is dusdanig gebruikelijk dat we er rustig vanuit kunnen gaan dat je dit veld bijna nooit hoeft te gebruiken.

Optional

De `<optional>`-tag verwacht altijd een `true` of `false` waarde: een boolean. Dit noemen we daarom ook wel een `flag`: het is een instelling die we aan of uit kunnen zetten. Soms is het zo dat een library alleen nuttig is tijdens het compileren, maar niet tijdens runtime. In dat geval kunnen we deze dependency `flaggen`, waarmee we aangeven dat 'ie niet vereist is om het project te kunnen uitvoeren.

8. Unit tests

8.1 Inleiding

Het programmeren van een werkend stukje software vergt vaak nogal wat `trial and error`. Wanneer je denkt dat je de juiste methode hebt geschreven, zul je dat ongetwijfeld controleren door jouw methode meerdere malen aan te roepen met verschillende waarden. Zo check je of de methode doet wat je verwacht, maar check je ook of je aan alle afwijkende scenario's hebt gedacht. Ontwesel ben je dus al bijg geweest met het `handmatig` testen van je code.

Handmatig testen is echter ontzettend langzaam. Een klein stukje code testen kost je waarschijnlijk een paar minuten, maar in de praktijk moet het hele softwarepakket bij iedere update opnieuw getest worden. Dat betekent dus ook alle features die in de vorige versie al gelezen waren. Je kunt je voorstellen dat wanneer je dit handmatig moet doen, je hier dagen mee bezig bent.

Daarom is het in de praktijk gebruikelijk om `automated tests` te gebruiken: een programma dat test of de code doet wat het moet doen. En dit biedt een hoop voordelen:

- Wanneer de tests eenmaal geschreven zijn, is er geen "handarbeid" meer nodig. Ze zijn dus onzetend goedkoop om (herhaaldelijk) uit te voeren;
- Het is razendsnel, de computer zal er hoogstens een paar milliseconden tot seconden voor nodig hebben;
- Omdat het snel en goedkoop is, kun je heel vaak en veel testen. Dit is bij ontwikkeling van software erg belangrijk;
- Wanneer je vaak en veel test, kom je er direct achter als je iets per ongeluk stuk hebt gemaakt. Verschillende stukken code hangen in grote projecten nauw met elkaar samen, maar de afhankelijkheden zijn niet altijd even goed te overzien. Het is dus logisch dat je het niet altijd opmerkt dat het aanpassen van de `return` value op plek x en stuk maakt op plek y. Automatisch testen zorgt er dus voor dat je minder vaak achteraf ontdekt dat bestaande features niet meer werken. Als dit wel gebeurd, noem je dit `unintended feature change`.

Automatische testen zijn dus van groot belang. Binnen automatische testen onderscheiden we twee varianten: `unit testen` en `integratiestesten`. `Unit testen` zorgen ervoor dat we kleine stukjes code - "units" - kunnen testen op hun werking. Zo verwacht je dat een functie die twee getallen bij elkaar optelt, de `int 5` teruggeeft bij de input `2` en `3`. Over het algemeen proberen we de units, net als de logica in onze methodes, zo klein mogelijk te houden. Bij `Integratiestesten` wil je vooral testen of de applicatie in hoofdlijnen goed functioneert. Bij een webshop kun je dan denken aan testen die controleren of er bij het aanroepen van veelgebruikte pagina's geen foutmeldingen optreden. Maar ook het testen van een gebruiker kan inloggen en of de belangrijkste functionaliteiten - zoals het toevoegen van producten aan een winkelwagen - wel werken.

Als we echter willen testen of de Btw van de producten correct berekend wordt, zul je snel merken dat een integratietest daar niet geschikt voor is. We moeten namelijk de hele applicatie opstarten en meerdere stappen ondernemen voordat we bij de productpagina zijn. Daarnaast moeten we dan het Btw-bedrag uit een weppagina gaan wissen, wat vrij onhandig en onsmakelijk is. Waarom moeilijk doet als het ook makkelijk kan? In onze applicatie zit een class die verantwoordelijk is voor deze berekening: `BtwCalculator`. Als we alleen deze class testen, kunnen we makkelijk bewijzen dat onze Btw-berekening klopt. Hierdoor gebruiken we een unit test. Wanneer je al je classes op die manier test, geeft dit in de basis al een goed vertrouwen dat de applicatie werkt naar behoren.

8.2 JUnit

De standaard framework voor het schrijven van unit testen in Java is `JUnit`. Dit open source framework is voor het eerst openbaar gemaakt in 2000 en is immiddels één van de meest gebruikte testing libraries ter wereld!

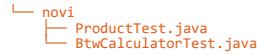
Wanneer je gebruik wil maken van JUnit, zul je de volgende code moeten toevoegen aan de pom.xml in de root van jouw project. In het hoofdstuk over Maven zullen we dieper ingaan op hoe dit precies werkt. Voor nu mag je de `<dependency>`-tag, inclusief inhoud, tussen de andere dependencies plaatsen:

```
...
<dependencies>
  ...
  <dependency>
    <groupId>org.junit.jupiter</groupId>
    <artifactId>junit-jupiter-engine</artifactId>
    <version>5.9.1</version>
    <scope>test</scope>
  </dependency>
</dependencies>
```

Maar, met alleen een testing framework heb je nog geen werkende unit test. Iedere test in onze applicatie zullen we zelf moeten schrijven. Hiervoor is het gebruikelijk om de test-code in een speciale test-class te zetten en in een aparte map op te slaan. Om ervoor te zorgen dat we desondanks begrijpen welke `test` voor welke class is, geven we dezelfde naam, maar krijgt de test-class het achtervoegsel `-test`. Baanbrekend, we know. We testen de class `BtwCalculator` dus met de test-class `BtwCalculatorTest`.

We slaan onze classes altijd op in `src/main/java`, maar onze test-bestanden slaan we op in `src/test/java`. Wanneer je jouw applicatie al had opgedeeld in packages, is het gebruikelijk (en bovendien erg handig!) om de package structuur van de test-classes gelijk te houden aan de gewone classes:





Dit betekent dat een class in `src/main/java/n1/novi/BtwCalculator` wordt gelezen met een class in `src/main/test/n1/novi/BtwCalculatorTest`. Leuk makkelijk!

Daarnaast worden de methodes van de test-classes geannoteerd met `@Test`. Daardoor herkent JUnit (en daarmee ook veel tooling zoals Maven en IntelliJ) dat dit een unit test betreft. Wanneer je deze annotatie gebruikt, zal IntelliJ direct vragen om de bijbehorende import toe te voegen. Je zult in IntelliJ ook zien dat je de tests allemaal tegelijk, of individueel kunt draaien met het groene pijltje naast de test!

The screenshot shows the IntelliJ IDEA interface. On the left, the project structure is visible under the 'webshop' project. It includes a '.idea' folder, '.mvn', and 'src' directory containing 'main' and 'test' packages. Under 'main/java', there are three classes: 'BTWCalculator', 'Main', and 'Product'. Under 'test/java', there are three test classes: 'BTWCalculatorTest' (which is currently selected), 'MainTest', and 'ProductTest'. Below these are files like '.gitignore', 'mvnw', 'mvnw.cmd', and 'pom.xml'. On the right, the code editor displays the 'BTWCalculatorTest.java' file. The code imports JUnit Jupiter annotations and defines a test class 'BTWCalculatorTest' with a private field 'productCalculator' of type 'BTWCalculator'. It contains a single test method 'testThatBTWCalculatorReturnsBTWPercentage()' which uses 'assertEquals' to check if the calculated BTW percentage is 21.00d given a product named 'Jas'.

Je weet nu hoe je JUnit in je project kunt toevoegen en welke naamgeving we gebruiken voor onze tests. Hoe je de tests daadwerkelijk moet opbouwen, leer je in de volgende paragraaf.

8.3 Unit tests

Om je uit te kunnen leggen hoe je een unit test schrijft, zullen we eerst even kijken naar de class die verantwoordelijk is voor het berekenen van het Btw-percentage op onze webshop-producten. Op sommige producten mag een laag Btw-tarief van 9% gerekend worden (zoals boeken) maar over de meeste producten rekenen we 21%. Dit is dus afhankelijk van het product in kwestie.

```
// BtwCalculator.java
class BtwCalculator {
    public double getBTWPercentage(Product product) {
        return product.getBTWPercentage();
    }

    public double getBTWAmount(Product product) {
        return product.getPriceExcludingBTW() * product.getBTWPercentage() / 100.00d;
    }
}
```

De BtwCalculator-class heeft geen properties, maar wel twee methodes die door iedere andere class gebruikt mogen worden. De eerste methode haalt het Btw-tarief van het product op en de tweede methode berekent het Btw-bedrag op basis van dat tarief en de productprijs.

Wanneer we testen gaan schrijven voor onze code, doen we dit in zo klein mogelijke units. Het zou in dit geval dus logisch zijn om beide methodes individueel te voorzien van een unit test. Bij het schrijven dan deze tests ga je vragen stellen over de werking van deze methodes. Zo kunnen we ons het volgende afvragen: "Haalt de methode `getBTWPercentage` echt het juiste Btw-percentage op?". Laten we eens kijken naar hoe dat zich zou vertalen naar code:

```
// BtwCalculatorTest.java
import static org.junit.jupiter.api.Assertions.assertEquals;
import org.junit.jupiter.api.Test;

class BtwCalculatorTest {
    // we maken een instantie van onze BtwCalculator-class,
    // zodat we toegang hebben tot de methodes;
    private BtwCalculator productCalculator = new BtwCalculator();

    @Test
    public void testThatBtwCalculatorReturnsBTWPercentage() {
        //Arrange:
        Product jacket = new Product("Jas", 50.00d, 21.00d);

        //Act:
        double btwPercentage = productCalculator.getBTWPercentage(jacket);

        //Assert:
        assertEquals(21.00d, btwPercentage);
    }
}
```

Let op: bovenstaand voorbeeld maakt gebruik van een Product-class die we in dit voorbeeld niet hebben uitgeschreven.

Binnen een testmethode zullen we altijd drie stappen doorlopen om onze test-vraag te beantwoorden:

1. **Arrange:** hier creëren we de context voor onze test. We zetten als het ware alle benodigde informatie klaar. Zo hebben we een werkende Product-instance nodig om de methodes van onze BtwCalculator mee te kunnen testen.
2. **Act:** hier voeren we de operatie uit. Dat betekent dat we de methode aanroepen of acties uitvoeren. Je kunt het lezen als: "Wanneer we met deze Product-instance de `getBTWPercentage`-methode aanroepen..."
3. **Assert:** hier maken we een bewering over de verwachte uitkomst. Dit kun je lezen als: "... dan verwacht ik dat de uitkomst 21% is". Dit doen we met behulp van `asserts`. Een assert zegt letterlijk: controleer of een expressie deze verwachte waarde heeft. Is de uitkomst van de assert `true`? Dan is de test geslaagd! Je mag zelfs meerdere asserts maken binnen dezelfde test, wanneer je meerdere resultaten wil controleren. Alle asserts moeten dan `true` zijn om de test te laten slagen.

Laten we nu ook nog een extra testmethode toevoegen om onze andere methode te testen. Hier vragen we ons het volgende af: "Berekent de methode `getBTWAmount` het juiste Btw-bedrag?"

```
// BtwCalculatorTest.java
import static org.junit.jupiter.api.Assertions.assertEquals;
import org.junit.jupiter.api.Test;

class BtwCalculatorTest {
    // we maken een instantie van onze BtwCalculator-class,
    // zodat we toegang hebben tot de methodes;
    private BtwCalculator productCalculator = new BtwCalculator();

    @Test
    public void testThatBtwCalculatorReturnsBTWPercentage() {
        //arrange:
        Product jacket = new Product("Jas", 50.00d, 21.00d);

        //act:
        double btwPercentage = productCalculator.getBTWPercentage(jacket);

        //assert:
        assertEquals(21.00d, btwPercentage);
    }

    @Test
    public void testThatBtwCalculatorReturnsBTWAmount() {
        //arrange:
        Product jacket = new Product("Jas", 50.00d, 21.00d);

        //act:
        double btwAmount = productCalculator.getBTWAmount(jacket);

        //assert:
        assertEquals(10.50d, btwAmount);
    }
}
```

*Let op: bovenstaand voorbeeld maakt gebruik van een Product-class die we in dit voorbeeld niet hebben uitgeschreven.

In bovenstaande voorbeeld hebben we telkens de assert `assertEquals` gebruikt. Deze assert verwacht altijd twee argumenten: de verwachte waarde en de uitkomst van de operatie. De waarden kunnen dan met elkaar worden vergeleken.

Naast `assertEquals` zijn er nog meer handige asserts, zoals:

- `assertTrue`: checkt of de conditie `true` is;
- `assertFalse`: checkt of de conditie `false` is;
- `assertNull`: checkt of de waarde gelijk is aan `null`;
- `assertNotEquals`: checkt of de waardes (of inhoud van de objecten) niet overeenkomen;
- `assertSame`: checkt of beide objecten naar hetzelfde origineel wijzen (dus `hetzelfde` zijn). Let op: twee objecten kunnen dezelfde `inhoud` hebben, maar dat maakt ze niet `hetzelfde`.

Mocht je arrays willen vergelijken, dan kun je gebruik maken van `assertArrayEquals`: die checkt of de `inhoud` van beide arrays overeenkomt.

Unit testen bieden veiligheid

Het is belangrijk om altijd te controleren of onze code doet wat we verwachten. Maar een ander essentieel onderdeel van unit testen is het feit dat we onze collega's (en onszelf!) behoeden voor **unintended feature change**. Omdat projecten normaliter erg groot zijn, kan het zomaar zo zijn dat het aanpassen van de `return value` op plek x iets stuk maakt op plek y.

Ook in onze BtwCalculator-class is dit risico groot. Neem de volgende praktijksituatie, waarin jij de opdracht hebt gekregen om de prijzen van onze webshop-producten op te slaan in bedragen *inclusief BTW*, in plaats van exclusief BTW. Dit wijzig jij in de Product-class, zonder te weten dat de BtwCalculator-class gebruik maakt van de `getPriceExcludingBTW`-methode.

De eerste "line of defense" (de eerste maatregel om een potentiële bug af te weren) is dat jij bij het maken van deze aanpassing de methode-naam waarschijnlijk veranderd in `getPriceIncludingBTW` (Zo zie je maar hoe belangrijk beschrijvende methodenamen voor jouw code zijn!).

Wanneer we in de Product-class deze methodenaam aanpassen, zal de compiler ons waarschuwen dat er in de BtwCalculator-class een methode gebruikt wordt die door jou niet meer bestaat. Dit noemen we **compiler safety**. Als we het hernoemen van de methode niet overal consequent doorvoeren zal de code niet complieën. Echter, misschien denk jij hier niet bij na en vervang je met een snelle *Find & Replace All*-actie in IntelliJ alle `getPriceExcludingBTW`-aanroepen met `getPriceIncludingBTW`.

Zo. Geen foutmeldingen meer, kunnen wij lekker gaan lunchen.

Dit soort foutjes slippen er gemakkelijker in dan je denkt. Daarom zijn unit testen een belangrijke "second line of defense". Bij het draaien van de unit tests - vaak iets dat standaard gebeurt wanneer je jouw wijzigingen naar GitHub pusht - faalt onze test:

```
@Test
public void testThatBtwCalculatorReturnsBTWAmount() {
    //arrange:
    Product jacket = new Product("Jas", 50.00d, 21.00d);

    //act:
    double btwAmount = productCalculator.getBTWAmount(jacket);

    //assert:
    assertEquals(10.50d, btwAmount);
}
```

Het Btw-bedrag van eenjas die €50,- kost komt nu namelijk niet meer overeen met €10.50. Dit zullen we nu snel gaan oplossen voor deze implementatie bij de eindgebruiker komt. Pfew! Gelukkig hadden we een unit test geschreven...

8.4 Mocking

Unit tests zijn ontzettend krachtig, maar in sommige gevallen niet werkbaar. Dit treft vooral op situaties waar we gebruik maken van data uit een database, of van gegevens die we ophalen van een externe partij (via een API). De data die we daaruit ophalen kan namelijk op ieder moment veranderen, waardoor onze tests niet betrouwbaar zijn.

In dit soort gevallen maken we gebruik van een principe dat **mocking** heet. Mocking wordt gebruikt om specifieke onderdelen van de code te isoleren door er een soort nep-versie van te maken. Zo kunnen we de code testen zonder afhankelijk te zijn van de echte databron. Daarnaast geeft het ons ook de vrijheid om specifieke scenario's of situaties te simuleren, zodat we de functionaliteit kunnen testen in een gecontroleerde omgeving.

Het mocken van classes is echt een vak apart. We zullen daarom tijdens deze cursus aan de oppervlakte blijven, maar het is erg belangrijk om dit concept in hoofdlijnen goed te begrijpen. Laten we een situatie schetsen waarin we de producten uit onze webshop niet zelf beheren, maar hier een externe service voor gebruiken. We kunnen onze producten via deze service terugvinden met behulp van product codes:

```
// BtwCalculator.java
class BtwCalculator {
    private final ProductService productService;

    public BTWCbtwCalculator(ProductService productService) {
        this.productService = productService;
    }

    public double getBTWPercentage(String productCode) {
        Product product = productService.getByProductCode(productCode);
        return product.getPriceExcludingBTW() * product.getBTWPercentage() / 100.00d;
    }

    public double getBTWAmount(String productCode) {
        Product product = productService.getByProductCode(productCode);
        return product.getPriceExcludingBTW() * product.getBTWPercentage() / 100.00d;
    }
}
```

Hoe zou in dit geval onze test eruit moeten zien? We willen nog steeds testen of de BtwCalculator correct werkt, maar we willen niet dat onze unit test afhankelijk is van een externe service. Wanneer het bedrijf stopt met het produceren van producten met de code "xyz", mag het niet zo zijn dat onze tests ineens falen.

Om dit op te lossen zullen we dus gebruik moeten maken van mocking. Dit doen we door een mock-class te maken: die gedraagt zich als een echte class, maar kunnen we vanuit de unit test "configureren". Om dit voor elkaar te krijgen, gebruiken we het framework **Mockito**. Het is bijna altijd mogelijk om een class te mocken: de enige voorwaarde is dat de mock zich aan dezelfde interface moet houden als de originele class.

Om Mockito te kunnen gebruiken in jouw project, zul je de volgende XML moeten toevoegen aan de pom.xml in de root-map:

```
...
<dependencies>
  ...
  <dependency>
    <groupId>org.mockito</groupId>
    <artifactId>mockito-core</artifactId>
    <version>5.0.0</version>
    <scope>test</scope>
  </dependency>
</dependencies>
```

Nu kunnen we onze bestaande test-class omschrijven naar testen die gebruik maken van een mock van de Product-service. We vervangen private ProductService productService; door private ProductService productService = Mockito.mock(ProductService.class);.

```
// BtwCalculatorTest.java
import static org.junit.jupiter.api.Assertions.assertEquals;
import org.junit.jupiter.api.Test;
import org.mockito.Mockito;

class BtwCalculatorTest {
  // we maken een mock van de productService
  private ProductService productService = Mockito.mock(ProductService.class);
  // we maken een instantie van onze BtwCalculator-class
  private BtwCalculator productCalculator = new BtwCalculator(productService);

  // testmethodes...
}
```

Nu hebben we een mock-ProductService die we kunnen controleren. Zo kunnen we bijvoorbeeld zelf een Product-instantie maken die de mock-service moet teruggeven wanneer een product met code "xyz" wordt opgevraagd. Dit doen we volgens de schrijfwijze Mockito.when(<methodcall>).thenReturn(<returnValue>); :

```
// We maken een jas aan om als product te gebruiken
Product jacket = new Product("Jas", 50.00d, 21.00d);
// Wanneer het product met code "xyz" wordt opgevraagd,
// Geeft onze mock-product-service altijd de jas terug
Mockito.when(productService.getByProductCode("xyz")).thenReturn(jacket);
```

We hebben onszelf er nu van verzekerd dat wanneer de methodes van de BtwCalculator in onze testclass een product opvragen met code "xyz", dat altijd hetzelfde product wordt teruggegeven. Dit zorgt ervoor dat onze tests de onderliggende functionaliteit kunnen blijven testen, ongeacht of we dit type jas over een paar jaar nog verkopen!

```
// BtwCalculatorTest.java
import static org.junit.jupiter.api.Assertions.assertEquals;
import org.junit.jupiter.api.Test;
import org.mockito.Mockito;

class BtwCalculatorTest {
  // we maken een mock van de productService
  private ProductService productService = Mockito.mock(ProductService.class);
  // we maken een instantie van onze BtwCalculator-class
  private BtwCalculator productCalculator = new BtwCalculator(productService);

  @Test
  public void testThatBtwCalculatorReturnsBTWPercentage()
  {
    //arrange:
    Product jacket = new Product("Jas", 50.00d, 21.00d);
    // Zorgt ervoor dat wanneer het product met code "xyz" wordt opgevraagd,
    // Het product dat wij zojuist hebben aangemaakt wordt teruggegeven.
    Mockito.when(productService.getByProductCode("xyz")).thenReturn(jacket);

    //act:
    double btwPercentage = productCalculator.getBTWPercentage("xyz");

    //assert:
    assertEquals(21.00d, btwPercentage);
  }

  @Test
  public void testThatBtwCalculatorReturnsBTWAmount()
  {
    //arrange:
    Product product = new Product("Jas", 50.00d, 21.00d);
    // Zorgt ervoor dat wanneer het product met code "xyz" wordt opgevraagd,
    // Het product dat wij zojuist hebben aangemaakt wordt teruggegeven.
    Mockito.when(productService.getByProductCode("xyz")).thenReturn(product);

    //act:
    double btwAmount = subject.getBTWAmount("xyz");

    //assert:
    assertEquals(10.50d, btwAmount);
  }
}
```