**Java Opdrachten**

**Opdracht 1 - Basic Syntax**

Voordat je Java kan gebruiken moet je de syntax kunnen begrijpen. Doe onderzoek naar de volgende termen en leg uit wat ze (voor Java) betekenen:

* Object
* Class
* Package
* Constructor
* Instance
* Method

**Opdracht 2 - Variables**

Ook *variables* werken in Java net iets anders dan je hiervoor misschien gewend bent. Java maakt onderscheid tussen 3 soorten variables:

* **Local** variables
* **Instance** variables
* **Class** variables (ook wel **Static** variables genoemd)

Wat voor soort variabele je gebruikt hangt onder andere af van de plek waar je hem declareert .

1. Waar wordt een local variable gedeclareerd?

Class/Static variables en Instance variables worden op dezelfde plek gedeclareerd, en houden ook allebei de gegevens van een Object bij.

1. Wat voor soort informatie sla je op in een instance variable?
2. Wat voor soort informatie sla je op in een class variable?

De waarde van een class variable kan je in principe ook opslaan in een instance variable.

1. Wat is het voordeel van het gebruiken van een class variable?

**Opdracht 3 - Access Control Modifiers**

Java gebruikt zogenaamde ***modifiers***om bepaalde eigenschappen van stukjes code te veranderen. Deze zijn te verdelen in ***Access Control Modifiers*** en ***Non-Access Modifiers***

De Access Control Modifiers zijn heel belangrijk om te begrijpen - die geven namelijk aan welke onderdelen van je applicatie van het betreffende stukje code gebruik mogen maken.

Noem de vier verschillende Access Control Modifiers en geef bij elke een korte beschrijving.

**Opdracht 4 - Non-Access Modifiers**

Zoals bij de vorige opdracht vermeld stond, zijn er ook enkele ***Non-Access Modifiers***. Het grootste deel hiervan hoef je nog niet te kennen, maar de volgende twee wel:

* Static
* Final

Static ben je eerder al tegengekomen. Kan je ook uitleggen waar je Final voor zou gebruiken?

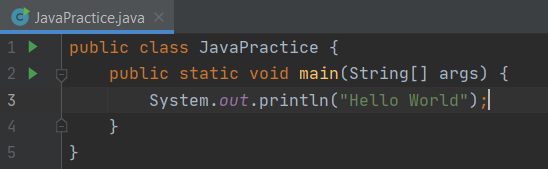
**Opdracht 5 - Strictly Typed**

Er wordt in IT onderscheid gemaakt tussen programmeertalen die **Strictly/Strongly Typed** zijn, en talen die **Loosely/Weakly Typed** zijn. Java is Strictly Typed - dit betekent dat je bij elke variabele in Java een datatype moet declareren. Bij Loosely Typed programmeertalen zoals JavaScript hoeft dit niet.

1. Wat is een voordeel van Strict Typing?
2. Wat is een voordeel van Weak Typing?

**Opdracht 6 - main & Arrays**

De main method van een Java class is de functie die wordt uitgevoerd als je een programma draait. Dit zal er altijd zo uit zien:



*Public* & *static* heb je hiervoor gehad; die geven aan dat je main method toegankelijk is voor iedereen, en dat deze voor elke Object van je class precies hetzelfde is.

*void* geeft aan dat er geen return value wordt verwacht. Bij de main method is dit verplicht!



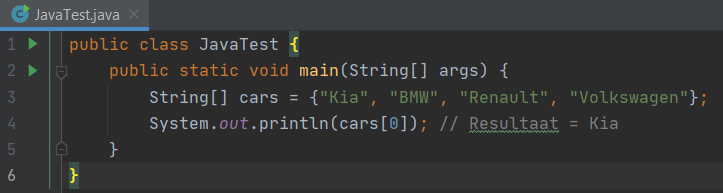
Tenslotte nog Dit slaat op de naam en parameters.

Met *String[] args* geef je dus aan dat de parameters van de *main* method in een array zitten. Dit array heet *args* (kort voor *arguments*) en bestaat enkel uit strings.

1. Kan je een reden bedenken waarom deze parameter verplicht is?

System.out.println() wordt gebruikt om tekst weer te geven in de console. Voor debugging doeleinden kan je dit net zo gebruiken als de console.log() command van JavaScript.

Arrays declareer je zo:

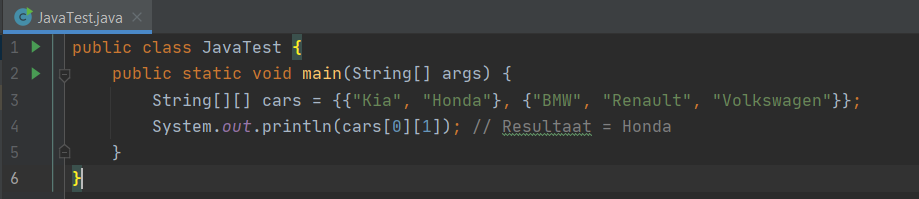


1. Kan je elementen van een Java Array veranderen?
2. Kan je elementen aan een Java Array toevoegen?

Net als bij JavaScript worden Arrays vaak gebruikt in combinatie met Loops. In de volgende opdracht gaan we hier verder op in.

**Theorie - Arrays in Arrays & ArrayLists**

Met Java kan je gebruik maken van arrays in arrays. Java noemt deze 2-Dimensional Arrays. Dit ziet er als volgt uit:



Bij de vorige opdracht heb je geleerd dat je de grootte van een Java Array niet aan kan passen. Als je dit wel wil doen, moet je gebruik maken van een ***ArrayList***.

**Belangrijk!** Voordat je een ArrayList kan gebruiken, moet je dat eerst in je eigen code aangeven. Dit doe je door de ArrayList code te ***importeren***.

Voorbeeld van een ArrayList:

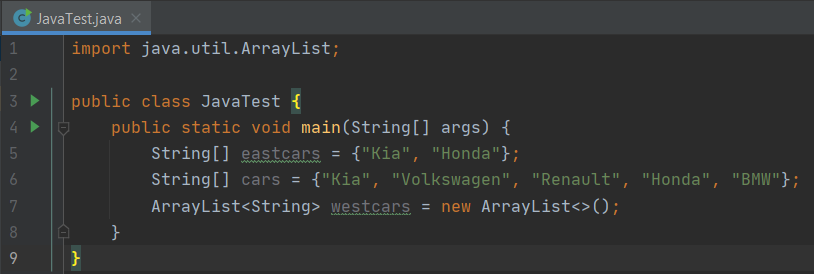


Je importeert de code van een ArrayList dus uit de *java.util* package.

**Opdracht 7 - Loops & Conditions**

Loops werken in Java net zoals je dat in JavaScript gewend bent. Ook conditions zijn heel erg vergelijkbaar. Er zijn hier en daar enkele kleine verschillen in de syntax, maar in gebruik zijn ze vrijwel identiek.

Gegeven de volgende code:



1. Maak gebruik van een for loop om de eerste drie items uit het cars Array te printen.
2. Maak gebruik van een for-each loop om alle items uit het cars Array te printen.

Zoals je ziet is de ArrayList westcars nog leeg - nu gaan we die vullen.

1. Loop door het cars array en vergelijk deze met de items in het eastcars array. Als je item niet in het eastcars array voor komt, voeg je hem toe aan het westcars array.

Vervolgens willen we onze Arrays en ArrayList mooi alfabetisch sorteren. Net als ArrayLists is hier een class voor die je moet importeren uit de java.util package.

* Als je een Array wil sorteren, moet je de Arrays class importeren
* Als je een ArrayList wil sorteren, moet je de Collections class importeren.

1. Pas dit nu toe op de Arrays en ArrayLists die je hebt gemaakt

**Opdracht 8 - Scanner en Date**

De Scanner en Date Objects zijn veel gebruikt; hiermee regel je de input van gebruikers en het bijhouden van tijd in je applicatie. Lees je over beiden in en probeer dan de volgende opdracht te maken:

Maak een java programma dat de gebruiker vraagt om hun naam en geboortedatum in te vullen. Vervolgens berekent je applicatie de leeftijd van die persoon en geeft dit met de naam weer.

*(tip: datum inlezen op basis van 3 getallen, dag maand en jaar)*

*(tip 2: huidige tijd opvragen kan je doen door volgende code regel “Date now = new Date();”)*

**Opdracht 9 - Subclasses en Superclasses**

Stel: je maakt een computersysteem voor een dierentuin. Je moet voor elk dier het volgende bijhouden:

* Dier is gevoerd.
* Dier is bij de dierenarts geweest.
* Naam en leeftijd

1. Maak een class Animal die de methods *feed()* en *visitVet()* heeft. De precieze uitwerking van de methods is nu nog niet belangrijk - als ze een berichtje in de console printen is dat al goed.
2. Geef je Animal class 2 variabelen - naam en leeftijd.
3. Maak ook een constructor die de naam en leeftijd van het dier als parameter neemt.
4. Maak tenslotte een static variable “type” en geef deze de waarde “Animal”. Waarom dit belangrijk is volgt in een latere vraag.

Vervolgens merk je: dit is niet het enige wat je voor je dieren moet bijhouden. Er zijn in jouw dierentuin ook een stel beren die naar een speciale beren-arts moeten. Daarvoor ga je een ***Subclass*** gebruiken. Syntax:

Hiermee geef je dus aan dat je class Bear een *subclass* is van je class Animal. Andersom heet dit een ***superclass*** - Animal is hier dus een *superclass* van Bear. Dit is vergelijkbaar met de parent-child naamgeving in HTML.

Net zoals bij HTML de child elementen bepaalde properties overnemen van de parent, nemen de subclasses bepaalde properties over van hun superclass. Hoe dit precies in zijn werk gaat gaan we in de volgende opdracht verder op in.

**Opdracht 10 - Ik zag 2 beren instantiëren...**

We moeten de code van de Bear class eerst wat uitbreiden:

1. Geef je Bear class een instance variabele “type”.

Nu heeft je Bear class ook een constructor nodig. Je kan dit net zo uitschrijven als je bij de vorige Animal constructor hebt gedaan. Je kan ook ***super()*** gebruiken. Super verwijst naar de superclass - super() verwijst dan ook altijd naar de constructor van de superclass.

1. Maak een constructor voor je class Bear. Deze moet de variabelen name, age en type als parameter gebruiken.

Vervolgens gaan we onze beren instantiëren. Dat doen we in de main method van de class Bear.

1. Maak 2 Bear Objects aan. De naam en leeftijd mag je zelf kiezen - zorg er wel voor dat de ene de type “Brown Bear” heeft en de andere de type “Polar Bear”.

Zoals je waarschijnlijk al hebt opgemerkt kan je dus een variabele die in je superclass zit ook gebruiken in de subclass. **Toch hebben je beren de informatie van beide variabelen “type”!** Dit gaan we nu laten zien.

1. Maak in je Bear class een method showType(). Print in deze method de type van je class Bear EN die van de superclass Animal. Roep deze vervolgens in je main method voor beide beren aan. *Tip: Gebruik het keyword super. Let ook op je access modifiers!*

Je Bear Objects zijn dus niet alleen een Bear met een property “type”, maar tegelijkertijd ook een Animal met een property “type”. Dat een Object tegelijkertijd als meerdere soorten Object behandeld kan worden kan je ook testen:

1. Maak in de main method van je Bear class naast de 2 beren nu ook een Animal object aan. Zet vervolgens alle beren in een Bear[] array EN alle drie je dieren in een Animal[] array.

Het concept dat Objecten tegelijkertijd als meerdere verschillende soorten Object behandeld kunnen worden wordt in Java ***Polymorphism*** genoemd.

**Opdracht 11 - Inheritance, Overriding & Overloading**

De methods *feed()* en *visitVet()* van Animal zijn te gebruiken door je beren, ook al heeft je Bear class die nog zelf niet.

1. Maak in je main een for-each loop die door je Animal array gaat en bij elk Animal de method *feed()* aanroept.
2. Doe nu hetzelfde voor je Bear array

Dit is een voorbeeld van ***Inheritance***.

Je kan *inherited methods* ook aanpassen om ze wat specifieker te maken.

1. Maak nu in je Bear class de methods *feed()* en *visitVet()*. Roep bij beiden eerst de method van de superclass aan, en voeg er vervolgens aan toe dat de beer de dierenarts heeft opgegeten.

Dit heet ***Method Overriding***.

Als je niet wil dat een method door je subclass veranderd wordt, kan je de method **final** maken. Je kan ook de hele class final maken, dan kan deze niet als superclass worden gebruikt.

Stel dat je nu een method wil maken die doorgeeft dat je beer iets speciaals heeft gegeten. Wat voor speciaals wil je als parameter aan je method doorgeven. Dit kan je doen door een nieuwe method aan te maken, maar dit kan ook door de bestaande method *feed()* te **overloaden**.

***Method Overloading*** houdt in dat je dezelfde naam voor verschillende methods gebruikt, met als verschil welke/hoeveel parameters worden gebruikt. Als de method wordt opgeroepen, kijkt je programma naar de parameters en selecteert aan de hand daarvan welke hij moet gebruiken.

1. Maak een 2e *feed()* method die een String als parameter neemt en dat in de output weergeeft.

Waar dit vooral veel bij wordt toegepast is bij constructors.

1. Overload je *Bear()* constructor met een extra constructor die naast name+age ook weight als parameter gebruikt.

**Bonus Opdracht - Dierentuin**

Nu gaan we een realistisch dierentuin-systeem nabouwen. Dit systeem houdt het volgende bij:

* Een lijst met alle dieren in de dierentuin
* De naam van elk dier
* De leeftijd van elk dier
* De geboortedag van elk dier
* De dag dat een dier voor het laatst gevoerd is.
* Per dag een nieuwe lijst met dieren die gevoerd moeten worden.
  + Als het de verjaardag van een dier is, ook een bericht dat dit meldt.

Gebruik de volgende classes:

* Zoo - je Zoo class heeft de volgende methods
  + *add(...)* - voegt een dier aan de dierentuin toe
  + *remove(...)* - haalt een dier uit de dierentuin weg
  + *update(...)* - Deze geeft terug:
    - Welke dieren vandaag jarig zijn
    - Welke dieren vandaag gevoerd moeten worden
    - Gebruik een Date Object als parameter, zodat je dit ook op bijvoorbeeld de dag van morgen kan testen.
* Animal
* Bird, subclass van Animal
  + Vogels worden 1x per 2 dagen gevoerd.
* Reptile, subclass van Animal
  + Reptielen worden 1x per week gevoerd.
* Cat, subclass van Animal
  + Katten worden dagelijks gevoerd.

**Theorie - Four Pillars of OOP**

Eerder heb je gelezen dat Java een Object Oriented Programmeertaal is. Maar wat houdt dat nou eigenlijk in?

Simpel gezegd gaat het over de coding structuur en concepten die een programmeertaal gebruikt om de applicatie vorm te geven. Het makkelijkste voorbeeld is het gebruik van Objects om data bij te houden, maar zo zijn er nog wel wat meer.

De belangrijkste concepten die elke OOP-taal gebruikt worden ook wel de **Four Pillars of Object Oriented Programming** genoemd. Het gaat dan over de volgende principes:

* **Inheritance**
* **Polymorphism**
* **Encapsulation**
* **Abstraction**

Hoe Java Inheritance en Polymorphism gebruikt heb je in de opdrachten hiervoor geleerd. In de volgende opdrachten gaan we de ***Encapsulation*** en ***Abstraction*** behandelen.

**Opdracht 12 - Encapsulation // Set yourself before you get yourself**

**Encapsulation** is het idee dat elk Object verantwoordelijk is voor zijn eigen data. Dit soort scheiding van verantwoordelijkheid maakt het makkelijker om na te gaan wanneer er iets aan een Object veranderd wordt.

Java gebruikt Encapsulation door middel van **private variables** en **Getters/Setters**. Dit houdt in dat de variabelen van een Object alleen gelezen of veranderd kunnen worden door methods van datzelfde Object. Deze methods worden dus ***getters*** en ***setters*** genoemd.

1. Maak een class School en een class Student. Geef beiden een private variable name, een method getName die deze weergeeft en een method setName die deze aanpast naar de gegeven parameter.

Je hebt nu je eerste eigen Getter en Setter gemaakt. Het is zoals je misschien merkt veel getyp voor een enkele variable. Gelukkig hebben meeste IDE’s tegenwoordig shortcuts om dit makkelijker te maken - dan hoef je alleen de private variable uit te typen. Daarna kan je IDE er automatisch de bijbehorende getters en setters voor aanmaken.

1. Onderzoek wat deze shortcuts zijn voor jouw IDE.

**Opdracht 13 - Abstraction**

**Abstraction** is het opzettelijk weglaten van concrete details. Net zoals je in een korte ochtendvergadering niet alles van je collega’s wil weten, is het voor een computer ook handig om de details soms te laten totdat ze nodig zijn. ***Als je code abstract maakt, beschrijf je dus alleen WAT er gebeurt, en niet HOE het gebeurt.***

Java gebruikt hiervoor ook het *keyword* ***abstract.*** Dit wordt gebruikt gebruikt om aan te geven dat een method of class abstract is.

1. Wanneer is een class abstract?
2. Maak in je Student class een abstract method *passYear().* Kan je al bedenken op wat voor plek je de code voor *passYear()* zou moeten implementeren?

Een abstract class kan je niet instantiëren. Omdat de method *passYear()* in je Student class abstract is gemaakt, is het niet mogelijk om een Student object te creëren.

1. Maak 4 subclasses van Student aan. Noem deze *FirstYear, SecondYear, ThirdYear* en *FourthYear*.
2. Implementeer de abstract method *passYear()* bij elk van deze! *(alleen een bericht in de console printen is voorlopig al genoeg).*
3. Geef elk van deze classes daarna ook het volgende:

* String Array met lessen die ze volgen (Engels, Nederlands, Wiskunde etc.)
* Numbers Array met cijfers die ze voor elk van deze vakken hebben
* String Arraylist met notities van slecht gedrag die over deze student gemaakt zijn als ze bijv. te laat zijn gekomen.
* Methods om bovenstaande Arrays te wijzigen
* Constructor

**Theorie - Interfaces**

**Abstraction** kan je dus zien als een soort contract tussen een abstract superclass en een subclass. De superclass daarbij geeft aan welke details de subclass moet beschrijven.

Soms wil je dat je class zich aan meerdere verschillende soorten regels kan houden. Met alleen de superclass is dit een probleem - Java laat maar 1 superclass per subclass toe, en daarmee dus eigenlijk 1 set regels per superclass.

Om dit op te lossen maakt Java gebruik van ***Interfaces***. Net zoals je het gebruik van een superclass aangeeft met het keyword *extends,* geef je het gebruik van een interface aan met het keyword *implements*. Dit ziet er zo uit:





Je kan ook meerdere interfaces tegelijkertijd gebruiken:



Een Interface is dus vergelijkbaar met een abstract class. Een belangrijk verschil is dat een Interface compleet abstract is. Dit houdt in dat ***ALLE*** methods van een Interface abstract moeten zijn.

**Opdracht 14 - Interfaces**

Nu gaan we onze Studenten uit de vorige opdrachten wat meer uitbreiden. De school wil namelijk dat bepaalde studenten extra vakken kunnen volgen als ze zich extra goed gedragen. De school noemt deze studenten *Honor Students*.

1. Maak een Interface *HonorStudent*, en geef deze de volgende abstract methods:

* *removeFromProgram()*;
* *checkForNotes();*

1. Maak voor je vier jaarlagen Studenten elk een nieuwe Honor Student subclass aan, dus voor *FirstYear* maak je *FirstYearHonor* etc. Implementeer bij al deze classes de *HonorStudent* interface.

Misschien vraag je je af waarom er geen informatie over bijvoorbeeld de extra vakken in de Interface staat. Deze kunnen nog verschillen per jaarlaag van de studenten - in de Interface staat alleen beschreven wat voor ELKE Honor Student verplicht is.

* Elke Honor Student **moet** een manier hebben om omgezet te worden naar een normale student - dus elke Honor Student **moet** een method *removeFromProgram()* **implementeren**.
* Elke Honor Student **moet** een manier hebben om te checken of ze nog voldoen aan de voorwaarden van de school. Als de school bijvoorbeeld de voorwaarde heeft dat een Honor Student niet meer dan 1 keer per maand een notitie van slecht gedrag. Elke Honor Student **moet** dit dus kunnen checken met de method *checkForNotes();*

**Theorie - File I/O**

File I/O staat voor file input/output - een manier om via je programma informatie uit bestanden te lezen of in permanente bestanden te schrijven. Het wordt ook wel eens een stroom genoemd - een stroom van data die uitgelezen of gemaakt wordt bij input/output van data. Om dit te faciliteren heeft Java een eigen API meegeleverd in de **java.io** package. De meest gebruikte classes uit deze package zijn **FileInputStream** en **FileOutputStream**

**FileInputStream**

De class FileInputStream van het pakket java.io kan worden gebruikt om gegevens (in bytes) uit bestanden te lezen. Het extend de abstract class InputStream.

Om informatie uit een file te lezen, moet je dus een instantie van de FileInputStream class aanmaken. Dit kan op meerdere manieren. Meestal zal je zoiets tegenkomen:

**FileInputStream input = new FileInputStream(String path);**

De path kan hier zowel een relative als absolute path zijn. Daarnaast kan je ook een FileInputStream aanmaken op basis van een File object:

**FileInputStream input = new FileInputStream(File fileObject);**

Meer leren over FileInputStream? [Hier](https://www.javatpoint.com/java-fileinputstream-class) staan wat voorbeelden.

**FileOutputStream**

De class FileOutputStream kan worden gebruikt om gegevens (in bytes) naar bestanden te schrijven. Ook een ​​FileOutputStream class kan op meerdere manieren gemaakt worden. Naast de String/File parameter is er bij de output nog een parameter: een boolean value.

**//Met een boolean parameter**

**FileOutputStream output = new FileOutputStream(String path, boolean value);**

**//Zonder boolean parameter**

**FileOutputStream output = new FileOutputStream(String path);**

De boolean value geeft (met true) aan of de output aan het einde van een bestand wordt toegevoegd. Als je deze weglaat of op false zet wordt de output aan het begin van het bestand geschreven.

Meer leren over FileOutputStream? [Hier](https://www.javatpoint.com/java-fileoutputstream-class) staan wat voorbeelden.