

# Reader Structured Program Development

**Versie:** 3.5 (29 Augustus 2017)  
**Auteurs:** C. Köppe en B.J.P. Nieuwhof  
Leo van den Berge (3.3)  
J.Visch (3.4, 3.5)  
**Screencasts:** L. Tijsma  
J. Visch (2017)

## Inhoud

I. Inleiding: Over deze Course en deze Reader .....	4
I.I De course .....	4
I.II. De Reader .....	4
II Programmeren .....	5
II.I Wat is programmeren .....	5
II.II. Wat is een computerprogramma.....	5
II.III. Het schrijven van een programma .....	5
II.IV Processing en Java .....	6
II.V Van geschreven naar werkend programma .....	6
II.VI De programmeeromgeving.....	6
III Processing .....	8
III.I. Downloaden en Installeren Processing .....	8
Downloaden Processing .....	8
Installeren Processing (windows) .....	8
III.II Klaar voor de start.....	8
1 Programma's, rare typetjes en meer.....	10
1.1 Types om mee te programmeren.....	11
1.2 Make-up en andere opmaakmiddelen .....	11
1.3 Middeltjes tegen geheugenverlies .....	13
1.4 Rekenen met variabelen .....	14
1.5 Test wat je geleerd hebt.....	14
2 Methoden, parameters, veldnamen en fouten .....	14
2.1 Methoden en parameters .....	15
2.2 Variabelen en Naamgeving.....	15
2.3 Rekenen met parameters en foutopsporing .....	15
3 Structuur, interactie, voorwaarden, events en testen .....	17
3.1 De GUI loop .....	17
3.2 Interactie, events en voorwaarden.....	20
3.2.1 Interactie .....	20
3.2.2 Voorwaarden.....	21
3.2.3 Kiezen uit een lijst van mogelijkheden .....	23
3.3 Testen .....	26
4 Methoden en Parameters, Scope .....	27
4.1 Scope van variabelen binnen je programma .....	27
4.2 Methoden met Parameters .....	30
5 Returnwaarden.....	33
5.1 De mindset.....	33
5.2 Het principe van returnwaarden.....	33
5.3 Returnwaarden toepassen op de driehoek .....	35
6 Herhalingen.....	37
6.1 Soorten herhalingen.....	37

6.2	De tellende herhaling .....	38
6.3	De voorwaardelijke herhaling 0 of meer keer .....	39
6.4	De voorwaardelijke herhaling 1 of meer keer .....	39
6.5	In het kort .....	42
7	Arrays.....	43
7.1	Enkelvoudige Arrays .....	43
7.2	2-dimensionale Arrays .....	44
7.3	Kubussen en nog meer dimensies.....	47
8	Van Probleem naar oplossing.....	48
8.1	De analyse, Wat moet het programma doen? .....	48
8.2	Het Ontwerp. Hoe ga ik het maken?.....	49
8.3	Van Ontwerp naar Realisatie .....	50
8.3.1	De velden declareren .....	51
8.3.2	Wanneer maak je een methode? .....	51
8.3.3	Conclusie .....	53

## I. Inleiding: Over deze Course en deze Reader

*screencast*

Welkom bij de eerste course programmeren in het propedeusejaar van ICA. In de studiehandleiding is je uitgelegd wat er van je verwacht wordt om deze course te kunnen halen.

Deze reader is onderdeel van het lesmateriaal voor deze course. Samen met een serie *screencasts*, de diverse opgaven en de college-uren vormt hij het totale lespakket waarmee je jezelf de basis van programmeren eigen kunt maken.

### I.I De course

De course is opgebouwd uit een aantal modules. Het werkschema voor deze modules is in de studiehandleiding uitgewerkt.

Elke module bestaat uit een hoofdstuk in deze reader, een aantal screencasts met uitleg, een serie vragen om te toetsen of je de behandelde theorie voldoende beheerst en een opdracht waarin je de aangeleerde kennis gaat toepassen. Het programma wordt gecompleteerd door de colleges. In deze colleges wordt ingegaan op de problemen die jij en je collega's aandragen. Hier wordt ook feedback gegeven op de uitgevoerde opdrachten.

Uitgangspunt is dat, wanneer je het volledige pakket wekelijks blijft volgen, de course goed haalbaar is.

### I.II. De Reader

Deze reader begeleidt het proces van de course. Allereerst zijn er enkele hoofdstukjes die genummerd zijn met Romeinse cijfers. Deze vertellen een algemeen inleidend verhaal (en natuurlijk dit hoofdstuk).

Vervolgens komt er per module een hoofdstuk. Elk hoofdstuk begint met een kort overzicht.

Vervolgens is er voor elke screencast die bij deze module hoort een paragraaf. In deze paragraaf staat soms nog wat aanvullende theorie. Ook staan hier steeds oefenopgaven bij die je kunt maken om zelf te oefenen met datgene wat je in de screencast hebt gezien.

*extra opgave*

Voor de gevorderde student is er soms een *extra opgave* om de geest verder aan te scherpen.

**Let OP!! Deze reader is dus niet voor in bed. Gebruik hem naast je PC of Laptop! Kijk de screencasts en maak de oefeningen!**

## II Programmeren

Allereerst, zoals aangekondigd wat inleidende beschietingen.

### II.I Wat is programmeren

Programmeren is het uitzetten van activiteiten in een bepaalde volgorde op een gepland moment. Op deze manier wordt een theateragenda voor een heel jaar volgepland. Ook wanneer je de TV-gids openslaat (dit is een papieren uitgave waarin je kunt lezen waar je al zappend langs schiet) zie je dat er een programma is samengesteld. Hier kun je precies zien in welke volgorde en op welk tijdstip een uitzending op een bepaalde zender wordt vertoond.

Een theater of een televisie zonder een programma heeft weinig toegevoegde waarde te bieden.

Als je het theater nu vergelijkt met een computer, kun je hetzelfde stellen. Ook een computer zonder programma staat alleen ruimte in te nemen. Je kunt er verder niets mee. Om die computer nuttig (of leuk) te gebruiken heb je programma's nodig. Zo'n programma kun je kopen of downloaden en vervolgens installeren. Daarna kun je de computer daadwerkelijk gebruiken. Als huis- tuin- en keukengebruiker heb je hier voldoende aan. Bedrijven kunnen hier vaak niet mee toe. Hierover gaat een deel van dit hoofdstuk. Wanneer we het hier verder over een programma hebben, bedoelen we een computerprogramma.

### II.II. Wat is een computerprogramma

Als computergebruiker kijk je tegen een computerprogramma aan als iets waarmee je kunt werken of spelen. We vinden het allemaal heel logisch dat we een tekstverwerker of een pdf-reader opstarten en hier een syllabus in kunnen lezen. Ook starten we een willekeurig spel op en spelen dit waarin we helemaal meegaan in de beleving van dit spel.

*teams*

Deze tekstverwerker en deze spellen zijn computerprogramma's die niet zo maar ontstaan zijn. Hier hebben *teams* van mensen aan geprogrammeerd om dit programma zo aan het werk te krijgen dat jij het uiteindelijk kunt gebruiken. Uiteraard is eerst bedacht wat het programma allemaal moet kunnen en hoe het op allerlei gebruikersacties moet reageren.

Vervolgens is er een ontwerp gemaakt om uit te zoeken hoe dat dan moet gebeuren. Aan de hand van dit ontwerp wordt het geprogrammeerd.

*vooraf gestelde eisen*

Het eindresultaat wordt vervolgens *getest* om vast te stellen of het aan alle *vooraf gestelde eisen* voldoet.

Resultaat van dit proces is een computerprogramma.

### II.III. Het schrijven van een programma

*instructies*

Een computerprogramma bestaat uit een aantal *instructies* die de computer in een bepaalde volgorde krijgt opgedragen. Om een computerprogramma te schrijven, moet je de computer dus de juiste instructie op het juiste moment laten uitvoeren. Wanneer je een beetje weet wat een computer is, begrijp je dat dit wel heel nauwkeurig moet gebeuren. Wanneer je aan een Nederlands sprekend persoon vraagt 'Wat is  $4 + 8$ ?' zal de gemiddelde persoon '12' antwoorden. Wanneer je een computer de opdracht geeft om  $4+8$  uit te rekenen, zal hij ook niet meer doen dan dat en als gebruiker zie je dan niet wat er gebeurd is. Sterker nog, wanneer je de computer niet vertelt dat hij de uitkomst moet onthouden, vergeet hij het ook direct weer. Als je de uitkomst op het scherm wil zien, zal je dat aan de computer kenbaar moeten maken.

*controles*

Wanneer je een persoon de opdracht geeft om een stap te zetten, zal deze persoon zich vergewissen van de uitvoerbaarheid van deze taak en dan eventueel een stap in een mogelijke richting zetten. In een computerspelletje waarin jouw avatar een stap moet zetten, zal je alle *controles* ter voorkoming dat de stap tegen een muur (of in een ravijn) eindigt moeten uitschrijven. Ook moet je nog vertellen in welke richting de stap gezet moet worden.

oplossingsalgoritme	<p>Dit alles moet ook nog gebeuren op een manier die maar voor één uitleg vatbaar is. Het resultaat van dit denk- en schrijfwerk noemen we een 'oplossingsalgoritme'. Dit is het moeilijkst deel van programmeren.</p> <p>Wanneer we het oplossingsalgoritme voor een probleem hebben bedacht, moeten we het omzetten in een programmeertaal. In deze course maken we gebruik van de programmeertaal 'Processing'. Er bestaan echter veel meer programmeertalen. Zomaar een greep:</p> <ul style="list-style-type: none"> <li>• C</li> <li>• C++</li> <li>• PHP</li> <li>• Ruby</li> <li>• C#</li> <li>• Perl</li> <li>• Python</li> <li>• Visual Basic</li> <li>• Java</li> </ul>
---------------------	------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------

## II.IV Processing en Java

Oracle	<p>Processing is een programmeertaal die ontwikkeld is door MIT (Massachusetts Institute of Technology). De taal is een doorontwikkeling op de programmeertaal Java. Een aantal complexiteiten zijn er echter uitgehaald waardoor het beter geschikt is als leeromgeving.</p> <p>Java is een programmeertaal die bedacht is door een grote computerfabrikant: Sun. Inmiddels is Sun niet meer zelfstandig en is het overgenomen door Oracle (vooral bekend van zijn databasesystemen). De gedachte achter Java is simpel: <i>Write once, run everywhere</i>. Normaal wanneer je een computerprogramma schrijft, kun je dit programma alleen op hetzelfde type computer gebruiken als waarop het geschreven is. Voor Java programma's geldt dat niet. Hoe dit in zijn werk gaat wordt later besproken. Voor nu hoeft je alleen te onthouden dat dit zo is.</p>
micro edition	<p>Een leuke bijkomstigheid is dat er voor Java ook nog een zogenaamde <i>micro edition</i> bestaat die gebruikt wordt voor mobiele telefoons, wasmachines, magnetrons en andere apparaten om ons heen die steeds <i>intelligenter</i> worden. Java wordt ook gebruikt in de programmeercourse OOPD in de propedeuse en ook in het semester DDOA wordt in Java geprogrammeerd.</p>
Javascript	<p>Verwar Java niet met <i>Javascript</i>. Hoewel de naam anders doet vermoeden hebben beide talen <u>niets</u> met elkaar te maken.</p>

## II.V Van geschreven naar werkend programma

	<p>Wanneer je zelf een programma schrijft, zijn de instructies die je hebt geschreven voor jou wel leesbaar. Voor de computer is dit echter abracadabra.</p>
instructieset	<p>De geschreven programmacode moet worden omgezet naar een <i>instructieset</i> die voor de computer begrijpelijk is. Dit noemen we <i>compileren</i>.</p>
compileren	<p>Het gecompileerde programma kan dan ook daadwerkelijk worden uitgevoerd op de computer. Een Apple begrijpt echter niet dezelfde taal als een Windows computer. Daarom zal je een programma voor een Apple ook op een Apple moeten compileren. Een programma voor een Windows computer moet dus ook op een Windows computer worden gecompileerd. Hier bestaan uitzonderingen op. Gecompileerde Java programma's kunnen in principe op alle computers draaien.</p>

## II.VI De programmeeromgeving

Processing	<p>Programmacode is feitelijk een stukje tekst dat met een bepaalde syntax in een bepaalde volgorde moet worden getypt. Dit kun je in principe met elke simpele tekstverwerker doen wanneer deze er maar toe in staat is om deze tekst zonder vreemde symbolen als platte tekst weg te schrijven. Comfortabel is echter anders. De fabrikanten stellen hiervoor luxe omgevingen ter beschikking. Zo'n omgeving noemen we een <i>geïntegreerde</i></p>
------------	-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------

*Eclipse. Netbeans* *ontwikkelomgeving*. De afkorting van het Engelse woord hiervoor is IDE (Integrated Development environment). In deze course wordt gebruikt gemaakt van *Processing*. Deze biedt al heel aardige opties. Laat je zelf niet verleiden tot het gebruiken van een luxe omgeving zoals *Eclipse* of *Netbeans*. Deze bevatten zoveel functionaliteit dat ze het zicht wegnemen op wat je hier moet leren.

## III Processing

Zoals gezegd programmeren we in deze course in- en met Processing. Processing is een programmeeromgeving met een programmeertaal die je van internet kunt downloaden. We zullen dit dus eerst moeten doen. Vervolgens installeren we Processing zodat we aan de slag kunnen.

### III.I. Downloaden en Installeren Processing

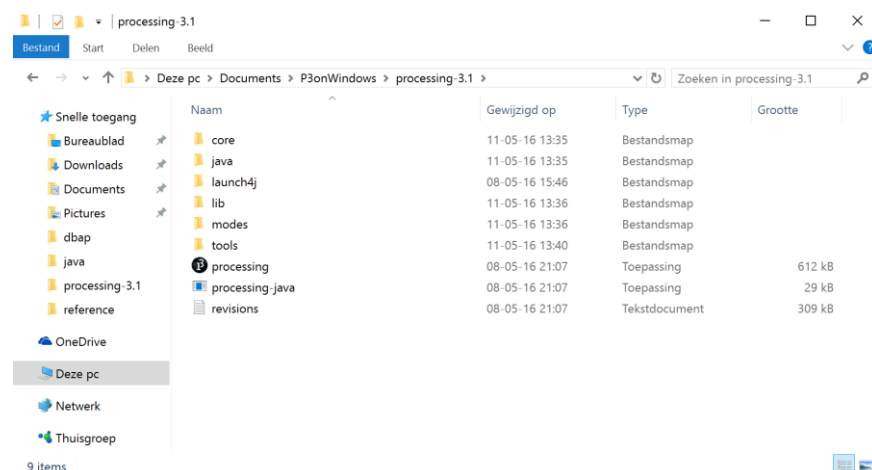
#### Downloaden Processing

Selecteer de link die hoort bij je besturingssysteem en download het bestand. Het kan zijn dat je docent deze bestanden ook anders beschikbaar stelt.

- Win 32: <http://download.processing.org/processing-3.1-windows32.zip>
- Win 64: <http://download.processing.org/processing-3.1-windows64.zip>
- Linux 32: <http://download.processing.org/processing-3.1-linux32.tgz>
- Linux 64: <http://download.processing.org/processing-3.1-linux64.tgz>
- Mac OS X: <http://download.processing.org/processing-3.1-macosx.zip>

#### Installeren Processing (windows)

Open de ZIP file en pak hem uit (Extract). Zoek weer een geschikte plek om alles neer te zetten (C:\school\java is misschien een aardige optie). Wanneer je de map opent, krijg je het volgende beeld:



*short-cut*

Om het voor je zelf gemakkelijk te maken kun je nu een *short-cut* (snelkoppeling) naar het programma Processing maken en deze op een handige plek zetten.

*Firewall*

Om te testen start je Processing een keer op. Het kan nu zijn dat je *Firewall* een melding geeft. Hoewel veiligheid een groot goed is, moet je nu het programma toch toestaan om op te starten. Een en ander moet tot het beeld op de volgende pagina leiden.

### III.II Klaar voor de start

Processing is nu geïnstalleerd en je bent klaar voor de eerste screencast. De lijst met screencasts krijg je aangeleverd via <http://onderwijsonline.han.nl>. In het vervolg van deze reader wordt je door de course heen geleid. De course is opgedeeld in modules. Elke module bevat een stukje begeleidende tekst in deze reader, één of meerdere screencasts en een aantal opgaven om voor je zelf vast te stellen of je de theorie voldoende kent. Er zijn oefenopgaven en moduleopgaven. De oefenopgaven betreffen steeds een



deel van de module. De moduleopgaven betreffen de gehele module. Tot slot is er een casusopdracht waarin je de (tot dan toe aangeleerde) theorie moet toepassen.

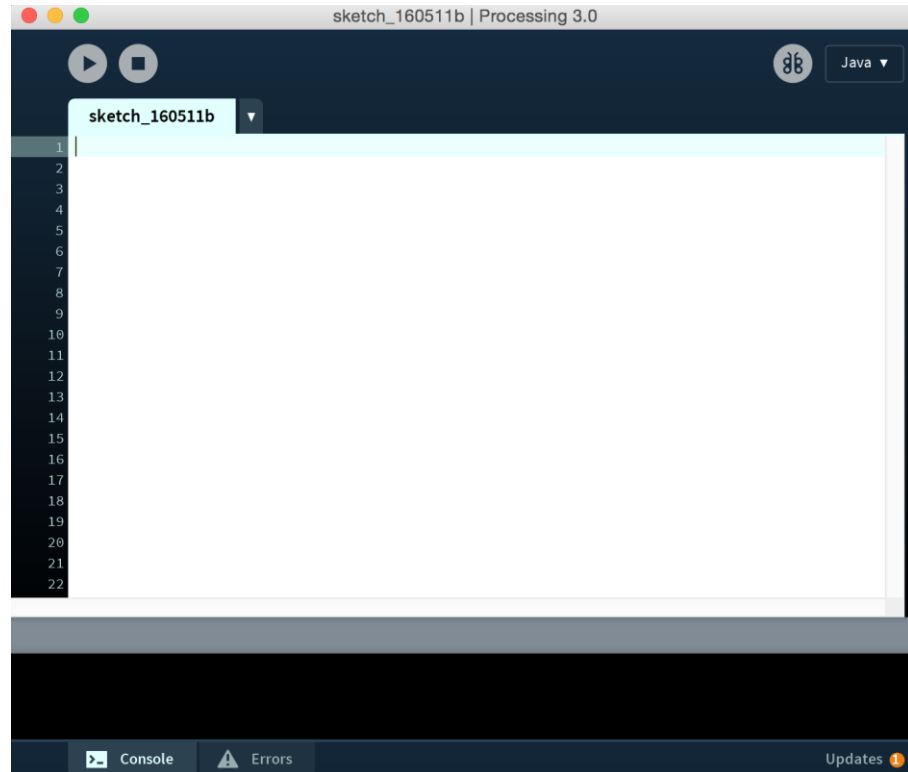
Je moet zelf weten of je eerst de tekst gaat lezen of eerst de screencast gaat maken. Je mag zelfs eerst de opgaven of de casus maken. De volgorde van werken kies je zelf. Sla echter geen delen over. Alle stappen zijn nodig. Ook wanneer je de casus hebt opgelost, is het belangrijk om voor je zelf op een rijtje te zetten waarom de gekozen oplossing werkt.

# 1 Programma's, rare typetjes en meer

In de eerste module wordt direct een eerste programmaatje gemaakt. Je maakt kennis met *Processing*. Je leert hierin dat je verschillende schermen hebt waarin je zaken kunt weergeven.

*geheugenvariabelen* Je leert over het gebruik van *geheugenvariabelen* en die bestaan in verschillende soorten. Je leert dat je met sommige variabelen kunt rekenen en je leert een zinvolle toepassing voor deze berekening.

Voordat je de eerste screencast bekijkt, kan het geen kwaad om het programma Processing alvast op te starten. In de screencast gebeurt dit ook al snel. Wanneer je Processing opstart krijg je het volgende beeld:



*Message Area*  
*Console*  
*Foutbalk*

Het witte vlak is het gedeelte waar je straks je programma gaat schrijven.

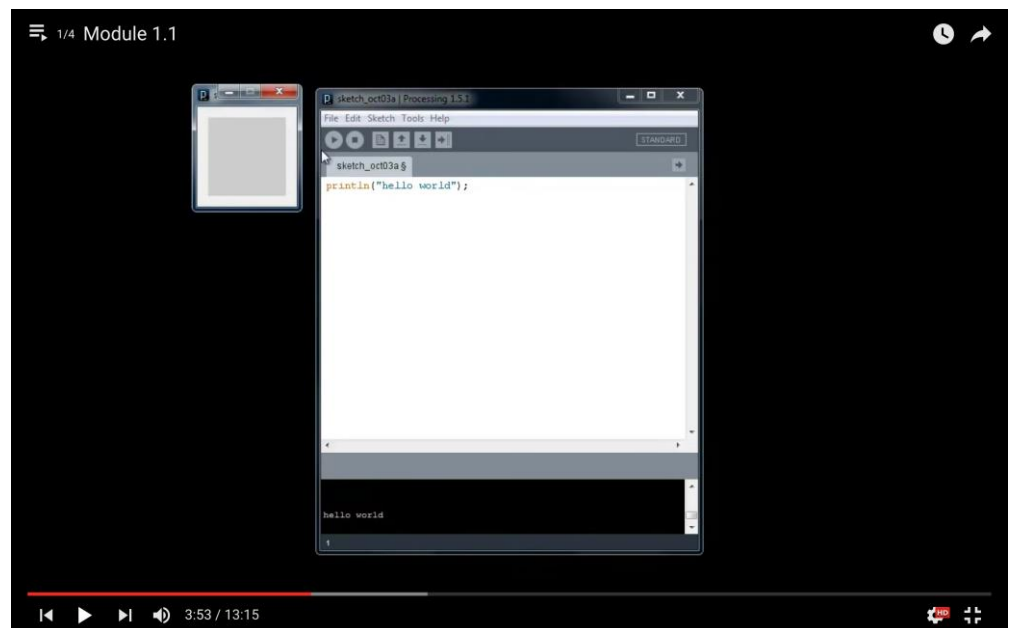
Daar onder zie je een klein zwart vlak. Dit wordt in Processing de *console* of *message area* genoemd.

Tussen het witte en zwarte vlak is een grijze balk. Deze balk heet de foutbalk en Processing gebruikt deze om fouten tijdens het programmeren te laten zien. Die balk wordt dan rood. Door op Errors te klikken krijg je dan uitgebreidere informatie over die fout.

Hoe je in deze omgeving programmeert kun je nu bekijken in je eerste screencast. LET OP: in de screencasts wordt een verouderde versie van Processing gebruikt (1.5). Dit maakt echter voor de meeste voorbeeld programma's niet uit.

## 1.1 Types om mee te programmeren

De eerste screencast laat je zien hoe Processing werkt. Al heel snel wordt het eerste mini-programmaatje voor je gemaakt. (Hello World):



*Pauzeren filmpje*

Als je naar de screencast kijkt en met je muis in dit scherm klikt, krijg je een pauze teken. Wanneer je daarop klikt, stopt de screencast en kun je het zelf ook een keer proberen in jouw eigen installatie van Processing.

*foutmelding*

Wanneer je eigen poging tot programmeren een keer fout gegaan is, zie je in de *foutbalk* een *foutmelding* die jou moet helpen om de fout te vinden. In de eerste screencast wordt hier iets over verteld.

De tekst *Hello World* is in de screencast tussen dubbele quotes (" ") gezet. Dat is niet zomaar. Dat heeft een reden. Wanneer je stukjes tekst in je programmacode opneemt die moeten worden weergegeven of bewaard, moet je altijd deze quotes er omheen zetten.

*String*

In de screencast wordt verteld dat een stukje tekst wordt weergegeven als een *String*. We kennen echter nog meer types. De belangrijkste voor dit moment worden hieronder op een rijtje gezet:

int	Een int staat voor een geheel getal
String	Een String staat voor een stukje tekst
float	Een float staat voor een getal met decimalen
boolean	Een boolean staat voor een Ja/Nee veld (true/false)

Later komen hier nog meerdere typen bij.

**Uitproberen!**

De screencast laat zien dat je stukjes tekst aan elkaar kunt plakken: Bijvoorbeeld: "Hallo " + "Wereld". Het resultaat hiervan is "Hallo Wereld". Met een int of een float kun je rekenen:  $9 * 8$  geeft als resultaat 72. En  $4.5 * 2.5$  geeft als resultaat 11.25. Wanneer je een String aan iets anders vastplakt, is dit weer een String. Dus: **"9 \* 8 = " + 9 \* 8; leidt tot "9 \* 8 = 72"**.

Maak Opgave 1.1 om de stof te oefenen

## 1.2 Make-up en andere opmaakmiddelen

*window*

In de tweede screencast wordt de message area (console) verlaten en gaan

we proberen alles in een *window* te tonen.. Behalve dat we tekst gaan tonen, worden ook vormpjes weergegeven zoals vierkantjes, en rondjes. Later krijgen dit soort vormen ook een betekenis. Een rechthoek is natuurlijk leuk om op scherm te tekenen. Wanneer je er ook nog een betekenis aan kunt geven, door hem bijvoorbeeld te gebruiken in een schema van een verkiezingsuitslag, gaat de rechthoek echt spreken.

### Het window

Op je PC of MAC kun je zien dat windows vaak verschillende afmetingen hebben. Kennelijk ligt dit niet vast. Het is de taak van de programmeur om te bepalen hoe groot een window in eerste instantie wordt.

Alles wat we weergeven in het window moet er dan natuurlijk wel inpassen. Zo zijn er een aantal zaken die je voor je zelf op een rijtje moet hebben:

- Hoe breed wordt mijn window?
- Hoe hoog wordt mijn window?
- Wat voor kleur krijgt mijn window?
- Wat ga ik voor teksten en figuren in mijn window zetten?
- Waar ga ik deze teksten en figuren neerzetten?
- Wat voor kleur krijgen deze teksten en figuren?
- Hoe groot worden de teksten en figuren?

Screencast 2 van module 1 helpt je op weg om antwoorden te vinden op deze vragen. Probeer in eerste instantie te variëren op de zaken die daar worden getoond.

*API*

De screencast laat vervolgens de documentatie van de taal Processing zien. Zo'n documentatie noemen we een *API*. Dat staat ook boven het eerste scherm dat opent wanneer je in Processing via het menu *Help* kiest voor *Reference (Handleiding)*.

API is de afkorting voor Application Programming Interface. In een moderne programmeertaal zijn hele stukken programma al voor jou geschreven. Dat hoeft je dus niet zelf meer te doen. In de API staat vervolgens hoe je deze stukjes programma op de juiste manier gebruikt. In de screencasts wordt dit aan de hand van een rechthoek uitgelegd.

Volg deze screencast in het geheel.

Je kunt nu de opgaven 1.2 t/m 1.4b maken.

### 1.3 Middeltjes tegen geheugenverlies

#### Variabelen

De screencasts van module 1 gaan in op het gebruik van variabelen. *Variabelen* zijn een soort van veldjes waaraan je een naam geeft en waar je een waarde in stopt die daarin bewaard moet blijven. Uit de wiskunde kennen we allemaal het principe van X de grote onbekende die we moesten oplossen. Wanneer we X een waarde toekennen, blijft X deze waarde houden tot we deze expliciet veranderen. Zo is dat met variabelen ook. Stel we hebben een variabele *naam*. Hier kennen we als volgt een waarde aan toe:

```
naam = "Meriintie Giisen";
```

De opdracht: `println("Hallo " + naam);`  
Toont dan *Hallo Merijntje Giisen* in de message area.

De screencast laat dit zien.

Voordeel van die variabele is dat je er één keer een waarde instopt. Vervolgens hoef je die naam niet meer te onthouden maar gebruik je gewoon de variabele op allerlei verschillende plekken in je programma. Wanneer de inhoud van de variabele dan verandert, is hij automatisch daardoor ook overal veranderd waar deze wordt gebruikt.

Maar..... Zoals het hier boven staat mag niet!  
Probeer dit maar uit en Processing zal een foutmelding geven.

#### declareren

Het is zo dat je een variabele pas mag gebruiken nadat je hem *gedeclareerd* hebt.

#### Wat is Declareren?

De screencasts gaan uitgebreid in op variabelen en het declareren hiervan. In een programma moet je kenbaar maken dat je een variabele gaat gebruiken. In paragraaf 1.1 is je iets verteld over Types. Je moet ook nog tegen je programma vertellen wat voor type waarde je in deze variabele gaat bewaren.

Voordat we bovenstaande code dus kunnen uitvoeren moeten we eerst de variabele naam declareren.

Dat doe je als volgt:

```
String naam;
```

Wanneer je dit heb gedaan, mag je vervolgens alleen nog maar tekst in het veld naam zetten.

*naam = 0.6*; leidt dus tot een foutmelding.

#### quotes

Echter: *naam = "0.6"*; is wel goed! Omdat de symbolen tussen *quotes* gezet zijn, worden ze als een String gezien.

Maak nu opdracht 1.5, geniet van het resultaat, vergeet niet om het op te slaan en ga verder met de stof.

## 1.4 Rekenen met variabelen

De screencasts laten zien hoe je met variabelen kunt werken. Of liever nog: Het laat zien hoe je variabelen voor jou kunt laten werken. Zo komen we al weer wat dichterbij datgene waarvoor de computer bedoeld is. De computer moet voor jou werken en niet andersom tenslotte.

We hebben gezien hoe je *Strings* door middel van 'optellen' aan elkaar vast kunt plakken. Dat dit ook met String variabelen werkt, is aangetoond in paragraaf 1.3.

Wanneer je echter 2 int variabelen bij elkaar optelt, vindt hier op de zelfde manier een optelling plaats als in je rekenboekje van de basisschool.

Voorbeeld:

```
int x = 25;
int y = 40;
println(x + y);
```

 drukt 65 af in het console.

Ook wanneer je float variabelen gebruikt, kun je hier mee rekenen met dien verstande dat je dan ook met cijfers achter de komma kunt werken. Er zitten hier echter toch enkele addertjes onder het gras:

Wanneer je met de bovenstaande variabelen het volgende commando zou geven:

*println*                      `println(y/x);`

zou je misschien wel als uitvoer 1,6 verwachten. Dat is echter niet zo. Probeer het maar eens uit. Kijk wat het resultaat is en probeer te verklaren waarom. Probeer zo allerlei berekeningen. Probeer in ieder geval de volgende bewerkingen. Te weten: optellen (+), aftrekken(-), vermenigvuldigen(\*), delen(/) en modulo(%).

Probeer ook het resultaat van een berekening met een float uit te laten komen in een int (b.v. `int x = 40.0 / 25`). Wat gebeurt er dan?

Opgave 1.6 en 1.7 zouden nu gemaakt moeten kunnen worden. Voor de liefhebbers is er nog een extra opgave.

## 1.5 Test wat je geleerd hebt

Casus

Op [onderwijsonline.han.nl](http://onderwijsonline.han.nl) wordt een moduleopgave aangeboden bij deze module. Probeer deze opgave tot een goed einde te brengen en stuur het resultaat in naar je docent. Deze moduleopgaven en de vragen die je hebt bij de oefenopgaven vormen het uitgangspunt voor het volgend college. Wanneer je deze zaken niet gedaan hebt, heeft het college voor jou ook weinig te bieden.

## 2 Methoden, parameters, veldnamen en fouten

*functie*

Module 2 gaat in op het gebruik van methoden en parameters. Behalve de term methode wordt ook vaak *functie* gebruikt. Ga er voorlopig van uit dat hiermee hetzelfde wordt bedoeld. In deze reader wordt vastgehouden aan de term *methode*. Dit is niet omdat de andere term onjuist is, maar Processing is gebaseerd op de programmeertaal Java. In deze programmeertaal is dit de officiële naam.

Veel methoden worden al via de API aangeboden. In Module 1 heb je kunnen leren hoe je in de API zaken kunt opzoeken. De API biedt methoden aan. Vaak moet je tussen haakjes dan waarden meegeven, bijv.

```
rect(10,23,45,67);
```

De waarden tussen haakjes noemen we parameters.

In deze module wordt een aantal handige methoden behandeld. Ook worden er tips gegeven om fouten in je programma te voorkomen en op te sporen.

## 2.1 Methoden en parameters

### *methode aanroep*

In de eerste module is al gebruik gemaakt van een API. Het tekenen van vierkantjes en cirkels deed je tenslotte niet zelf maar je gaf Processing opdracht om deze vormen te tekenen en er een bepaalde kleur aan te geven. Een dergelijke opdracht noemen we een *methode aanroep*.

De methode zoals tot nu toe gebruikt, deed wat voor je maar had verder geen invloed op het verdere verloop van jouw programma.

Er zijn echter methoden die wel invloed hebben. Zij zijn er namelijk toe in staat om de waarden van variabelen te veranderen. In de API kunnen we er zo een aantal vinden:

Wanneer we het hoofdstukje *math* opzoeken staan daar allerlei methodes die dat kunnen doen.

Een voorbeeld:

### *Machtsverheffen*

Klik in de reference van *Processing* op *pow()*.

Hier staat uitgelegd wat deze methode doet. Wanneer ik in mijn programma een vierkant heb, weet je dat de oppervlakte gelijk is aan de lengte van een zijde in het kwadraat:

```
int lengte = 20;  
float oppervlakte = 0;
```

Na:

```
oppervlakte = pow(lengte, 2);
```

is de waarde van oppervlakte in 400.0 gewijzigd. (Let op de decimaal want oppervlakte is een float.)

In de screencast wordt het voorbeeld aangehaald met absolute waarde.

Na:

```
int absoluteWaarde = abs(-400);
```

is de waarde van het veld *absoluteWaarde* 400 (Zonder de -)  
Bekijk de screencasts en zie nog meer toepassingen.

Hierna is het tijd voor opgave 2.1 t/m 2.3

## 2.2 Variabelen en Naamgeving

### *Betekenis veldnaam*

Even een opmerking over veldnamen. Er is een afspraak om velden een naam te geven die overeenkomt met de werkelijke betekenis. (Dus geen x1, x2, y3 etc..) De werkelijke betekenis kun je echter niet altijd in één woord vangen. (Zoals *Absolute Waarde*.) Hiervoor is de volgende afspraak gemaakt:

### *Hoofdletters?*

Wanneer je meer woorden nodig hebt, schrijf je deze gewoon aan elkaar alsof het één woord is. Om het leesbaar te maken, laat je ieder woord beginnen met een Hoofdletter. **Behalve:** Het totale woord begint *altijd* met een kleine letter! (Dus geen *AbsoluteWaarde* maar *absoluteWaarde*)

## 2.3 Rekenen met parameters en foutopsporing

In de screencast wordt dieper ingegaan op het gebruik van parameters. Het is handig om methodes te gebruiken voor klusjes die steeds weer terugkomen. Alleen ze worden steeds net een beetje anders gebruikt. Dat "beetje anders" wordt dus veroorzaakt door die parameters.

Het aardige is nu dat de parameters die je gebruikt, vaak ook weer het gevolg zijn van een berekening. Als parameter mag je dus ook deze berekening zelf

opgeven.

Dus:

```
float macht = pow(9,3);
```

is gelijk aan:

```
float macht = pow(3*3, 12/4);
```

#### *verkeerde volgorde*

Soms doet een programma heel andere dingen dan je verwacht. Meestal gebeurt dit, doordat je ergens een foutje hebt gemaakt in een berekening of dat je een aantal opdrachten net in de *verkeerde volgorde* hebt gezet. Ook kan het gebeuren dat een methode, die je hebt gebruikt, net een ander resultaat opleverde dan je verwachtte.

Als dit gebeurt, komen we bij een moeilijker onderdeel van programmeren: foutopsporing.

#### *Debuggen*

Een beproefde manier van foutopsporing is *debuggen*. Debuggen is het doorlopen van je programma waarbij je het verloop binnen jouw geschreven code kunt volgen. Op elk moment kun je dan kijken welke waarde een bepaalde variabele heeft. In Processing 3 kun je dit doen met de Debugger



tool. Het gebruik hiervan wordt apart behandeld tijdens college.

Wanneer je geen debugger beschikbaar hebt of niet weet hoe hij werkt, moet je een list verzinnen om te kunnen debuggen. Deze list is even simpel als geniaal:

#### *Print veld*

Voeg op de belangrijke plekken in je programma regels toe om de inhoud van velden af te drukken:

```
println("De waarde van nulpuntX = " + nulpuntX);
```

```
println("De waarde van nulpuntY = " + nulpuntY);
```

Wanneer je deze regels in je programma zet vlak voordat je de velden nulpuntX en nulpuntY gebruikt, wordt de waarde in de console afgedrukt, en kun je controleren of deze velden de waarde hebben die je verwacht.



### 3 Structuur, interactie, voorwaarden, events en testen

In deze module gaan we een begin maken met het structureren van een programma. We gaan wat dieper in op de interactie tussen de gebruiker en het programma. Dit gebeurt aan de hand van gebeurtenissen oftewel *events*. Verder wordt het gebruik van condities behandeld en het gebruik van accolades.

#### 3.1 De GUI loop

GUI

De screencasts van module 3 laten zien hoe een GUI in Processing werkt. Er wordt gereageerd op mousekliks waardoor de kleur van een figuur verandert. In deze paragraaf wordt het principe hiervan uitgelegd.

Events

Het window waarin jullie tot op heden de resultaten konden zien noemen we ook wel een *GUI (Graphical User Interface)*. De message area of de console is meer een uitlaatklep voor teksten.

In een GUI kun je op willekeurig plekken klikken met je muis en wanneer er invulvelden instaan, kun je zelf kiezen wanneer je welk veld invult. In de screencast beweeg je met je muis over een window. Zo'n klik of wijziging noemen we een *event*. Een event moet eerst worden gedetecteerd. Wanneer een event wordt gedetecteerd koppel je daar (al dan niet) een consequentie aan. (Klikken op een OK Button leidt er meestal toe dan de ingevulde velden in een window worden verwerkt.

GUI-loop

Zaken die in zo'n window staan kunnen veranderen. Doordat er iets gebeurt, kan iets bijvoorbeeld van kleur veranderen. Het is allemaal heel leuk wanneer je als programmeur over een vierkant vertelt dat het voortaan rood is in plaats van blauw. Echter... De gebruiker kan dit pas zien wanneer het vierkant opnieuw getekend is in het window.

Eigenlijk is het dus zo dat dit window een lijstje heeft met een hele serie gebeurtenissen. Van elke gebeurtenis checkt hij steeds of het is voorgekomen. Zo ja, dan worden passende maatregelen genomen. Als alle gebeurtenissen afgewerkt zijn, wordt het window opnieuw getekend zodat de gebruiker zich ook kan vergewissen van de nieuwe situatie. Om het een naam te geven, noemen we het hier de *GUI loop*.

Probeer het je een beetje zo voor te stellen:

Start:

```
mouseMoved();  
mouseClicked();  
/  
.  
.  
/  
draw();  
→ en nu echt tekenen!  
ga terug naar Start:
```

Eind:

draw()

Deze loop (in het Nederlands *lus*) wordt dus in principe eindeloos herhaald. De methode die het scherm steeds opnieuw tekent doet dit op de achtergrond. Als programmeur in Processing hoeft je daar niets voor te doen. Voorafgaand aan dit daadwerkelijk tekenen wordt altijd de methode *draw()* uitgevoerd. Zoals je in bovenstaand fragment kunt zien wordt deze methode steeds opnieuw uitgevoerd in de GUI loop. Veranderingen die optreden kun je dus in deze methode zetten waardoor het window met deze veranderingen opnieuw wordt getekend. Om een beeld te geven: in standaard instelling wordt deze lus 60 x per seconde uitgevoerd. Dit gaat dus aanzienlijk sneller dan de waarnemingssnelheid van het menselijk oog. Hoewel dit feitelijk leidt tot een knipperend beeld, vernemen wij daar helemaal niets van.

*setup()*

In de screencast zie je dat behalve deze *draw()* methode nog een andere methode moet worden gemaakt. Deze methode heet *setup()*. In deze methode kun je al je variabelen, voor zover als mogelijk, de juiste waarde geven. Deze variabelen moet je dan wel gedeclareerd hebben. Het declareren van deze variabelen doe je dus niet in de *setup()* en ook niet in de *draw()* maar helemaal bovenaan. Hieronder staat een voorbeeld van de structuur van een programma weergegeven. Deze wordt vervolgens verklaard in het vervolg van deze paragraaf. Wanneer je het programma overneemt en uitvoert, laat dat wel zien dat de methode *draw()* steeds opnieuw wordt uitgevoerd.

**Dit voorbeeld kun je in Processing overnemen en uitvoeren..**

**LET OP!**

**Mocht je snel last krijgen van draaierigheid bij flikkerende beelden, zet dat de waarde in *delay(500)*; niet te laag!**

```
int windowX = 600,
    windowY,
    marge = 10,
    rechthoekX,
    rechthoekY,
    cirkelDiameter = 600,
    cirkelX,
    cirkelY,
    seq = 0,
    RED = #FF0000,
    YELLOW = #FFFF00,
    GREEN = #00FF00,
    BLUE = #0000FF;

void setup() {
    size(600, 600);
    windowY = (windowX) ;
    rechthoekX = windowX - (2*marge);
    rechthoekY = windowY - (2 * marge);
    cirkelX = windowX / 2;
    cirkelY = windowY / 2;
}

void draw() {
    switch (seq%4) {
        case 0:
            fill(RED);
            break;
        case 1:
            fill(YELLOW);
            break;
        case 2:
            fill(GREEN);
            break;
        default:
            fill(BLUE);
            break;
    }
    seq++;
    if (rechthoekY > 0) {
        rect(marge,marge, rechthoekX, rechthoekY);
        marge = marge + 10;
        rechthoekX = rechthoekX - 20;
    }
}
```

```

        rechthoekY = rechthoekY - 20;
    } else if (cirkelDiameter > 10 ){
        ellipse(cirkelX, cirkelY, cirkelDiameter, cirkelDiameter);
        cirkelDiameter = cirkelDiameter - 20;
    }
    delay(500);
}

```

Uitleg bij het programma:

De eerste serie regels zijn declaraties van velden die worden gebruikt. Hier gebeurt wat raars: Alleen bij de eerste variabele staat aangegeven dat het een *int* betreft. Het zijn echter allemaal *int*'s.

Als je goed kijkt, zie je dat alle declaratieregels (behalve de laatste) met een *komma* in plaats van een *punt komma* worden afgesloten.

DUS:

```
int  lengte, breedte, hoogte;
```

betekent het zelfde als:

```
int  lengte;
int  breedte;
int  hoogte;
```

#### Hexadecimaal

Verder zie je dat de velden RED, YELLOW etc. een heel rare waarde hebben. Dit noemen we een *hexadecimale* waarde. Dit is een ander talstelsel dan dat we gewend zijn. We zijn gewend aan een talstelsel waarin de cijfers lopen van 0 tot 9. In het *hexadecimale* stelsel lopen de waarden van 0 tot F. (Waarbij de F gelijk is aan de decimale waarde 15.) Dit stelsel wordt ook wel het 16-tallig stelsel genoemd

#### Constanten

Ook zie je dat de velden voor deze kleuren geheel in hoofdletters staan. Dit is ook weer zo'n afspraak. We spreken af dat velden met een *constante* waarde (dat wil zeggen: een waarde die we in de loop van het programma nooit veranderen) in *Hoofdletters* weergeven. Zo zijn de velden als zodanig direct herkenbaar.

Na de declaraties komt de methode *setup()*. Deze methode wordt één keer uitgevoerd. Hier worden dus berekeningen en bewerkingen uitgevoerd die initieel één keer moeten plaatsvinden. LET OP: Indien je de *size(...)* methode gebruikt moet deze als eerste commando binnen de *setup()* staan. Je mag in Processing 3 geen variabelen meegeven aan *size(..)*, maar alleen harde waardes, zoals in dit voorbeeld: *size(600,600);*.

Vervolgens staat er de methode *draw()*. Deze methode wordt aan het eind van de *GUI loop* steeds opnieuw aangeroepen. Deze methode maakt gebruik van allerlei tekenhulpmiddelen die in de *API* terug te vinden zijn. Vervolgens staat er nog een constructie met *if(.....)* en *else*. In de screencast wordt hier al aandacht aan geschonken. In de volgende paragraaf wordt hier dieper op ingegaan.

Helemaal aan het eind wordt de methode *delay(500);* aangeroepen. Dit is een methode die het programma even stilzet. De waarde 500 staat voor het aantal milliseconden. Wanneer je dit getal kleiner maakt, zal het programma dus sneller zijn werk doen.

*Nu is een goed moment om aan opgave 3.1 te werken.*

## 3.2 Interactie, events en voorwaarden

De screencasts van module 3 tonen hoe een programma kan reageren op acties van een gebruiker. Wanneer een programma reageert op prikkels van buitenaf, noemen we het programma *interactief*. Er is dus sprake van interactie.

Soorten interactie

Interactie kan op verschillende manieren plaatsvinden. Met name in oude computerprogramma's schrijft een programma precies voor in welke volgorde een gebruiker veldjes mag invullen in een scherm. Het verloop van een dergelijk programma verloopt dus precies volgens een uitgestippeld proces. We noemen dit *process driven*.

### 3.2.1 Interactie

Event driven

Wanneer een gebruiker allerlei verschillende dingen kan doen, is het verloop van het programma al een stuk minder voorspelbaar. Wanneer een programma met een muis wordt bestuurd kun je als programmeur niet van te voren weten waar het pijltje van de muis zich bevindt op het moment dat de gebruiker het een goed idee vindt om te *klikken*. Zo'n muisklik noemen we een *gebeurtenis* of in het Engels een *event*. Afhankelijk waar de muis op dit moment staat moet het programma het juiste doen. Een dergelijke benadering noemen we *event driven*. Het programma reageert dus op gebeurtenissen. In paragraaf 3.1 wordt verteld over de *GUI loop*:

```
Start:
    mouseMoved();
    mouseClicked();
    /.
    .
    .
    .
    ./
    draw();
    ga terug naar Start:
Eind:
```

Er is in de vorige paragraaf vooral ingegaan op de methode *draw()* die 60 keer per seconde wordt uitgevoerd. In deze loop gebeuren echter meer dingen. Een belangrijk onderdeel hiervan is het *uitluisteren* van *events*. In dit voorbeeld worden er al 2 genoemd. Gelukkig hebben deze *events* logische namen waaraan je meteen kunt zien wat ze betekenen.

Als eerste staat genoemd *mouseMoved()*; Dit is een methode die (wanneer je deze hebt beschreven) wordt aangeroepen nadat de muis verplaatst is. De actuele Xpositie van de muis wordt weergegeven in het veld *mouseX*. De actuele Ypositie van de muis wordt weergegeven in het veld *mouseY*. Om nu iets te doen met het gegeven dat je met de muis hebt bewogen, moet je in je eigen programma een methode maken met de naam *mouseMoved()*:

```
void mouseMoved() {
    // zet hier je programma code
}
```

Wanneer je programma wordt uitgevoerd en je met de muis beweegt, wordt deze methode automatisch uitgevoerd.

*Neem even de tijd om je te verdiepen in opgave 3.2 en 3.3.*

### 3.2.2 Voorwaarden

In deze module heb je tot nu toe kennis gemaakt met:

- De opbouw van een Processing programma;
- Declareren van variabelen;
- De methode `setup()`;
- De methode `draw()`;

Events:

- `mouseMoved`;
- `mouseClicked`;

- de methode `mouseMoved()`;
- de methode `mouseClicked()`;

#### Conditie

Voordat je echter een programma kunt maken dat een beetje ergens op lijkt is er nog een belangrijk iets om te weten: In het Nederlands noemen we dat de *voorwaarde* in het Engels de *condition*. Sommige dingetjes wil je alleen laten gebeuren wanneer er aan een bepaalde voorwaarde is voldaan.

Voorbeeld:

**Als resultaat Groter of gelijk aan 5,5  
aantalStudiepunten wordt vermeerderd met 7,5**

*if*

Feitelijk neem je dit bijna letterlijk over wanneer je dit in een programmeertaal wil zetten. In Processing ziet het er als volgt uit:

```
float resultaat, aantalStudiepunten;
//
//      Hier moet een stukje programma komen dat het resultaat
//      voor een eindtoets inleest en het aantal tot nu toe
//      behaalde studiepunten
//
if      (resultaat >= 5.5)
    aantalStudiepunten = aantalStudiepunten + 7.5;
```

In dit geval wordt bij het aantalStudiepunten  $7\frac{1}{2}$  opgeteld wanneer het resultaat  $5\frac{1}{2}$  of groter is.

#### Let op!

De voorwaarde achter het *if*-statement dient tussen haakjes te staan! Ook zie je een *operator* die 2 mogelijkheden aangeeft: ' $>$ ' groter dan en ' $=$ ' gelijk aan.

*Tijd voor opgave 3.4.*

#### AND operator &&

In de screencast wordt ook een samengestelde voorwaarde getoond. Dat betekent dat iets aan twee voorwaarden moet voldoen alvorens het wordt uitgevoerd. Deze combinatie wordt uitgevraagd met behulp van de AND operator `&&`.

Het is dus zo dat een conditie die je formuleert altijd evalueert in *true* of in *false*. In het geval van het gebruik van de AND operator zal de totale conditie alleen tot *true* evalueren, wanneer aan *beide* voorwaarden is voldaan.

Stel dat je voor 2 toetsen minimaal een 5,5 had moeten hebben voor je studiepunten, dan zou het algoritme als volgt zijn:

**Als resultaat1 groter of gelijk aan 5,5  
En resultaat2 groter of gelijk aan 5,5  
aantalStudiepunten wordt vermeerderd met 7,5**

De vertaling naar Processing ziet er dan als volgt uit:

```
float resultaat1, resultaat2, aantalStudiepunten;  
//  
//      Hier moet een stukje programma komen dat het resultaat  
//      voor beide toetsen inleest en het aantal tot nu toe  
//      behaalde studiepunten  
//  
  
if      (resultaat1 >= 5.5 && resultaat2 >= 5.5)  
        aantalStudiepunten = aantalStudiepunten + 7.5;
```

*Ruimte voor opgave 3.5.*

OR

Behalve deze AND operator kennen we ook een OR operator. Deze combineert twee voorwaarden. In tegenstelling tot de AND operator evalueert de OR al tot *true* wanneer één van beide condities waar is. Hiervoor gebruiken we in Processing de OR operator `||`. In het voorbeeld van de studiepunten zou je kunnen zeggen dat studiepunten worden toegekend op basis van het toets resultaat of op basis van een EVC (vrijstelling).

Dit gaat er als volgt uitzien:

**Als resultaat groter of gelijk aan 5,5  
of evc is waar  
aantalStudiepunten wordt vermeerderd met 7,5**

De vertaling naar Processing ziet er dan als volgt uit:

```
float resultaat, aantalStudiepunten;  
boolean evc;  
//  
//      Hier moet een stukje programma komen dat het resultaat  
//      voor en de eventuele evc inleest en het aantal tot nu toe  
//      behaalde studiepunten  
//  
  
if      (resultaat1 >= 5.5 || evc)  
        aantalStudiepunten = aantalStudiepunten + 7.5;
```

Je kunt deze operatoren natuurlijk ook combineren. Gebruik dan alleen, net als bij rekenen, voor de zekerheid haakjes zodat je weet dat de conditie op de juiste manier geëvalueerd wordt.

Want:

**If (resultaat1 >= 5.5 AND (resultaat2 >= 5.5 OR evc))**

Is niet hetzelfde als:

**If ((resultaat1 >= 5.5 AND resultaat2 >= 5.5) OR evc))**

In het eerste geval kan alleen resultaat2 door een evc worden gecompenseerd terwijl in het tweede geval de evc voor de gehele course geldt.

#### *De andere gevallen*

Stel dat de student in bovenstaande situatie bij een onvoldoende resultaat een punt aftrek zou krijgen, dan zou dat ook moeten worden verwerkt. Je krijgt dan het volgend algoritme:

```
Als resultaat Groter of gelijk aan 5,5  
    aantalStudiepunten wordt vermeerderd met 7,5  
anders  
    aantalStudiepunten wordt verminderd met 1
```

*else*

Voor dit *anders* wordt in Processing het sleutelwoord *else* gebruikt. In Processing ziet het er dan zo uit:

```
float resultaat, aantalStudiepunten;  
//  
//    Hier moet een stukje programma komen dat het resultaat  
//    voor een eindtoets inleest en het aantal tot nu toe  
//    behaalde studiepunten  
//  
  
if    (resultaat >= 5.5)  
    aantalStudiepunten = aantalStudiepunten + 7.5;  
else  
    aantalStudiepunten = aantalStudiepunten - 1;
```

*Bijpassend bij de deze stof zijn de opgaven 3.6 en 3.7.*

### **3.2.3 Kiezen uit een lijst van mogelijkheden**

Soms heb je te maken met een ingewikkelder keuze. Stel we hebben de regel dat je maximale toetscijfer kleiner wordt bij elke herkansing.

Wanneer je een toets voor de eerste keer maakt, kun je maximaal een 10 geregistreerd krijgen op je certificaat. Na 2 keer kan dit nog maximaal een 8 zijn, na 3 keer een 7 en na 4 keer een 6. Wanneer je de toets na 4 keer nog niet hebt gehaald, kun je geen voldoende meer krijgen en wordt je maximale cijfer een 5. Voor sommige toetsen kan dit betekenen dat het automatisch leidt tot definitieve uitsluiting maar daar hoeven we ons hier niet druk over te maken.

De laatste screencast van hoofdstuk 3 laat hier een oplossing voor zien. Het algoritme dat we oplossen is het volgende:

**Start**

```
Wanneer pogingnummer is 1:  
    Certificaatcijfer = de kleinste waarde van cijfer en 10.0  
    Ga naar Einde
```

```
Wanneer pogingnummer is 2:  
    Certificaatcijfer = de kleinste waarde van cijfer en 8.0  
    Ga naar Einde
```

```
Wanneer pogingnummer is 3:  
    Certificaatcijfer = de kleinste waarde van cijfer en 7.0  
    Ga naar Einde
```

```
Wanneer pogingnummer is 4:  
    Certificaatcijfer = de kleinste waarde van cijfer en 6.0  
    Ga naar Einde
```

```
In alle andere gevallen:  
    Certificaatcijfer = de kleinste waarde van cijfer en 5.0
```

**Einde**

*switch*

In Processing kennen we hier de *switch* constructie voor. Het ziet er dan als volgt uit:

```
float cijfer = 4;
int poging = 10;
float definitief = 0;

switch (poging) {    // de keuze wordt bepaald door poging
  case 1:            // als poging is 1
    definitief = constrain(cijfer, cijfer, 10);
    break;           // ga naar }
  case 2:
    definitief = constrain(cijfer, cijfer, 8);
    break;
  case 3:
    definitief = constrain(cijfer, cijfer, 7);
    break;
  case 4:
    definitief = constrain(cijfer, cijfer, 6);
    break;
  default:
    definitief = constrain(cijfer, cijfer, 5);
    break;
}
```

*break*

De term *break* is een sleutelwoord dat opdracht geeft om deze *structuur* uit te springen. De werking van de methode *constrain(...)* is in de eerste serie screencasts uitgelegd en is ook terug te vinden in de *reference* van Processing.

*Accolades*

Op allerlei plekken in programma's zie je steeds *accolades* (in het Engels: *curling braces*) verschijnen. Hoewel ze soms triviaal lijken is het tegendeel waar. Zij geven het begin ( { ) en het eind ( } ) van een blok aan. De term *blok* moet je ruim zien. Op de volgende pagina zie je enkele voorbeelden. De commando's die je hebt geschreven binnen de methodes *setup()*, *draw()*, *mouseClicked()* etc. begonnen allemaal met een accolade openen ( { ) en eindigden allemaal met een sluit-accolade ( } ). Binnen een methode kun je zelf weer opnieuw zo'n blok definiëren. Stel dat je, bij een voldoende toetsresultaat niet alleen de studiepunten wil aanpassen maar ook wil aanvinken dat een student voor een course geslaagd is. In dat geval moeten er 2 commando's worden uitgevoerd. Maar hoe weet de computer nu hoeveel commandoregels hij moet uitvoeren na het evalueren van een conditie? JUIST! Daar gebruiken we deze accolades weer voor.

Dus:

```
float resultaat, aantalStudiepunten;
boolean evc, geslaagd;
//
//      Hier moet een stukje programma komen dat het resultaat
//      voor en de eventuele evc inleest en het aantal tot nu toe
//      behaalde studiepunten
//
if      (resultaat1 >= 5.5 || evc) {
    aantalStudiepunten = aantalStudiepunten + 7.5;
    geslaagd = true;
}
```



Zonder de accolades is alleen de eerste opdracht (tot aan de ;) afhankelijk van de conditie. Wanneer we de accolades er niet omheen zouden zetten, zou elke student dus geslaagd zijn ongeacht zijn cijfer. Hoewel dit mogelijk een aantrekkelijk perspectief is, is dit toch echt niet de bedoeling.

LET OP!

Wanneer je programma groter wordt, zal je zien dat je steeds meer gebruik maakt van accolades. Om te voorkomen dat je door de bomen straks het bos niet meer ziet bestaat er een afspraak over het lay-outen van je programma's. In het geval van accolades is het de gewoonte om de openingsaccolade achter aan de regel te plaatsen die het blok inleidt. Vervolgens spring je één tab in voor elke programmaregel om vervolgens de sluitaccolade weer een tab terug te laten springen. In het vorige fragment zie je dat de sluitaccolade precies onder de *i* van de bijbehorende *if-conditie* staat. Hieraan kun je zien hoever de commando's doorlopen die bij deze conditie horen.

Schematisch ziet je programma er als volgt uit:

```

declaratie1;
declaratie2
.
.
//
void setup() {                                // openen methode setup()
    commando1;
    commando2;
    .
    .
    //
}                                                // sluiten methode setup()

void mouseClicked() {                        // openen methode mouseClicked()
    commando3;
    conditie1 {                                // openen blok conditie1 = true
        commando1.1;
        commando1.2;
        .
        .
        //
    }                                            // sluiten blok conditie1
    commando4;
    .
    .
    //
}                                                // sluiten methode mouseClicked()
                                                // Zie verder volgende pagina

void draw() {                                // openen method draw()
    conditie2 {                                // openen blok conditie2 = true
        commando2.1;
        commando2.2;
        .
        .
        //
    }                                            // sluiten blok conditie2 = true
    else {                                    // openen blok conditie2 = false
        commando2.3;
        commando2.4;
    }
    }                                            // sluiten blok conditie2 = false
                                                // sluiten methode draw()

```

### 3.3 Testen

#### Testen

Een belangrijk onderdeel bij het programmeren is dat je ook aan kan tonen dat je programma werkt. Je moet dus kunnen laten zien dat het doet wat het moet doen en niets anders. Dit bereik je door je programma te *testen*. Testen is meer dan je programma uitvoeren, wat acties doen en gewoon kijken of het werkt. Want als je alleen dit doet zijn er een aantal problemen. Ten eerste weet je zo niet zeker of je alle belangrijke acties hebt gedaan. Verder hebben programmeurs nogal de neiging om alleen de “happy path” te testen (de acties die wel goed gaan omdat de programmeur dit net heeft ontwikkeld). Maar gebruikers doen soms heel vreemde dingen en daar moet het programma ook mee om kunnen gaan. Denk daarbij aan foute waardes (voorbeelden), klikken ergens op het scherm, een actie middenin afbreken etc. Zeker als je werkt met condities kan dit dus gebeuren. Daarom gaan we bij het testen meer gestructureerd aan de slag. We bedenken hierbij welke verschillende acties (of soms ook waardes) mogelijk zijn en hoe het programma daarop zou reageren. Als iemand bijv. in een programma voor een school als cijfer een -1 wil invoeren, dan zou het programma hier dus een foutmelding moeten geven dat dit geen geldige waarde is (dit is dan het verwachte resultaat na de actie). Een cijfer 7 zou uiteraard geaccepteerd worden. Opgeschreven zouden dit dus twee testcases kunnen zijn:

Testcase	Actie	Verwacht resultaat
1	-1 als cijfer invoeren	Systeem toont foutmelding: “cijfer is niet geldig, moet 1 t/m 10 zijn”
2	7 als cijfer invoeren	Cijfer wordt geaccepteerd en in systeem opgeslagen

Op die manier kan je dus alle mogelijke acties bedenken en hoe het systeem daarop zou moeten reageren. Al deze acties en verwachte resultaten beschrijf je dan in je *testplan*. Dit kan je trouwens al doen zonder dat er ook maar een regel code is geschreven, want het gaat vooral om de gevraagde functionaliteit, en die is al bekend.

Nadat je de software (deels) hebt ontwikkeld ga je de tests dan uitvoeren om te kijken hoe goed je software werkt. Je gaat hierbij dan observeren wat het daadwerkelijke resultaat is. Als die overeenkomt met het verwachte, dan is die test geslaagd. De samenvatting van alle uitgevoerde tests (per testcase) zet je dan in een *testrapportage*. Hieronder twee voorbeelden (van alleen die ene testcase):

#### 1<sup>e</sup> uitvoer test

Testcase	Actie	Verwacht resultaat	Geslaagd
1	-1 als cijfer invoeren	Systeem toont foutmelding: “cijfer is niet geldig, moet 1 t/m 10 zijn”	Nee
2	7 als cijfer invoeren	Cijfer wordt geaccepteerd en in systeem opgeslagen	Ja

#### 2<sup>de</sup> uitvoer test (nadat controle op verkeerde cijfers is toegevoegd in programma)

Testcase	Actie	Verwacht resultaat	Geslaagd
1	-1 als cijfer invoeren	Systeem toont foutmelding: “cijfer is niet geldig, moet 1 t/m 10 zijn”	Ja
2	7 als cijfer invoeren	Cijfer wordt geaccepteerd en in systeem opgeslagen	Ja

Ga op die manier aan de slag bij de volgende moduleopgaves en laat zo zien dat deze daadwerkelijk goed werken. Meer informatie over testen krijg je later ook nog in SAQ, maar je kan ook hier kijken:  
[https://nl.wikipedia.org/wiki/Testen\\_\(software\)](https://nl.wikipedia.org/wiki/Testen_(software)).

## 4 Methoden en Parameters, Scope

In module 2 en module 3 hebben jullie al wat met methoden gedaan. In module 4 wordt uit de doeken gedaan hoe je meer kunt doen met zelf geschreven methoden. Je krijgt ook uitgelegd hoe je methoden met parameters zelf schrijft en gebruikt.

In het verlengde hiervan wordt ook de scope, oftewel het bereik, van variabelen behandeld. De reader wijkt qua volgorde een beetje af van de screencasts. In de reader wordt eerst de scope behandeld en daarna de methodes met parameters.

Door deze techniek op een goede manier toe te passen vermijd je het schrijven van dubbele code en wordt je programma eenvoudiger aanpasbaar.

### 4.1 Scope van variabelen binnen je programma

Een variabele die je declareert hoeft niet binnen het gehele systeem zichtbaar te zijn. Wanneer je een klein programmaatje schrijft van 50 regels zou dit nog wel te overzien zijn. Grotere systemen bevatten echter duizenden variabelen. Wanneer die altijd overal zichtbaar en bereikbaar zouden zijn, zou elke variabele een unieke naam moeten hebben en zou je een grote bibliotheek moeten bijhouden met de betekenis van elke variabele.

*Maximale beperking*

Dit zou een gebrek aan overzicht en derhalve grote fouten tot gevolg hebben. Om deze reden is het belangrijk dat je variabelen alleen beschikbaar hebt op de plek waar dat echt nodig is. De meeste programmeertalen bieden dan ook de mogelijkheid om met het bereik (de scope) van variabelen te spelen.

*Declareren*

Voordat we variabelen mogen gebruiken moeten we ze *declareren*. Wanneer we een variabele declareren zeggen we tegen het systeem dat we hem gaan gebruiken. We hoeven de variabele op dat moment nog geen waarde te geven. Dat kan ook later. De plaats waar je een variabele **declareert** is bepalend voor het bereik binnen je programma.

Zo zijn in Processing de variabelen, die je bovenin in je programma declareert, binnen het gehele programma te gebruiken:

```
int hoogte, breedte;

void setup() {
    hoogte = 15;
    breedte = 20;
}

void draw() {
    rect(10, 10, hoogte, breedte);
}
```

In bovenstaand stukje zie je dat de velden hoogte en breedte in alle methoden binnen het programma te gebruiken zijn.

*Accolades*

Eigenlijk is de stelregel heel eenvoudig: Een variabele is bekend tussen de accolades waar hij is gedeclareerd. Hier bestaat één uitzondering op. Deze wordt besproken in de paragraaf over methoden met parameters.

In het geval van de variabelen die je bovenin processing declareert, denk je nu waarschijnlijk dat daar geen accolades omheen staan..... Mis! Deze accolades worden er door processing wel degelijk omheen gezet. Je ziet ze alleen niet.

*Lokale variabelen*

Wanneer je een variabele declareert binnen een methode, is deze variabele

alleen lokaal binnen deze methode te gebruiken. Daarom heet dit ook een lokale variabele.

Dat heeft de volgende consequentie:

```
void setup() {  
    int hoogte = 15;  
    int breedte = 20;  
}  
  
void draw() {  
    rect(10, 10, hoogte, breedte);  
}
```

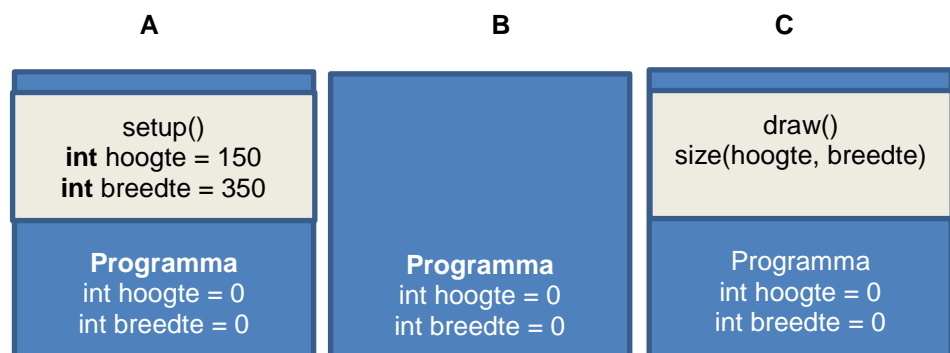
*Gelijke namen*

Bovenstaand stukje code zal een fout opleveren. De velden *hoogte* en *breedte* zijn alleen bekend binnen de accolades van de methode *setup()*. Daarbuiten zijn ze niet bekend en kunnen ze dus niet gebruikt worden. Nu nog iets raars. Variabelen kunnen gelijke namen hebben en toch niet hetzelfde zijn. Kijk maar naar volgend voorbeeld:

```
int hoogte = 0, breedte = 0;  
  
void setup() {  
    int hoogte = 150;  
    int breedte = 350;  
}  
  
void draw() {  
    rect(10, 10, hoogte, breedte);  
}
```

Dit zijn de lastiger gevallen. Het systeem zal geen fout geven. Alleen de velden *hoogte* en *breedte* boven in zijn niet het zelfde als in de methode *setup()*. Ze hebben dezelfde naam maar zijn totaal anders. Omdat de globale *hoogte* en *breedte* geïnitieerd worden met de waarde 0, wordt in *draw()* de methode *rect(10,10,hoogte, breedte)* aangeroepen met de waarden 0. De feitelijke aanroep is dus *rect(10,10, 0, 0)*;

Eigenlijk moet je het zien als in onderstaand plaatje. Het programma is een soort van container. De methode *setup()* is een werkje dat tijdelijk wordt ingelezen door het programma (Situatie A). Wanneer dat werkje zijn werk heeft gedaan wordt het er weer uit gekukeld (Situatie B). Dat betekent dat alle nieuw geïntroduceerde variabelen met het werkje mee weggegooid worden. Dus de nieuw gedeclareerde velden *hoogte* en *breedte* zijn dus weer weg. Vervolgens wordt er een werkje *draw()* boven op het programma gestapeld (Situatie C). De enige velden *hoogte* en *breedte* die *draw()* kent, zijn de velden in Programma. Hiermee zal *draw()* dan ook werken.



Het gaat nog verder dan dit. Stel je voor, dat je twee getallen hebt die je op volgorde van klein naar groot moet leggen. Wanneer je deze twee getallen van veld wil ruilen, heb je tijdelijk een derde veld nodig om een van beide getallen in op te slaan:

```
int getalA = 200, getalB = 150, getalC;  
getalC = getalA;  
getalA = getalB;  
getalB = getalC;
```

GetalC heeft nu verder geen nut meer en staat een beetje ruimte in te nemen.

Stel we doen dit in een methode met de naam `zetKleinstelnA()`. Dan zou dat als volgt kunnen worden opgelost:

```
int getalA = 300;  
int getalB = 200;  
  
void zetKleinstelnA() {  
    if (getalB < getalA) {  
        int wissel = getalA;  
        getalA = getalB;  
        getalB = wissel;  
    }  
    println(getalA + " < " + getalB);  
}
```

Wanneer ik in bovenstaand fragment bij de `println` ook het getal `wissel` zou gebruiken, leverde dit een fout op. `Wissel` wordt gedeclareerd in het blok achter de *if conditie* die met accolades wordt omringd. Alleen binnen deze accolades is de variabele `wissel` bekend. Na de sluitaccolade worden alle nieuw gedeclareerde variabelen weer weg gekukeld.

Dit lijkt ingewikkeld. Wanneer je echter wat beter nadenkt, is dit ook wel prettig. Die variabele `wissel` heb je verder ook niet meer nodig en wanneer je elders in je programma een soortgelijk kunstje moet uithalen, kun je vrolijk weer een variabele met de naam `wissel` erbij halen. Omdat jouw programma `wissel` vergeet, mag jij `wissel` als programmeur ook vergeten. Heerlijk toch!!! Kun je je tenminste concentreren op die zaken die echt belangrijk zijn.....

## 4.2 Methoden met Parameters

In de screencasts van module 4 kun je zien hoe een lamp wordt opgebouwd en hoe hier vervolgens meerdere exemplaren van kunnen worden getekend op verschillende plaatsen.

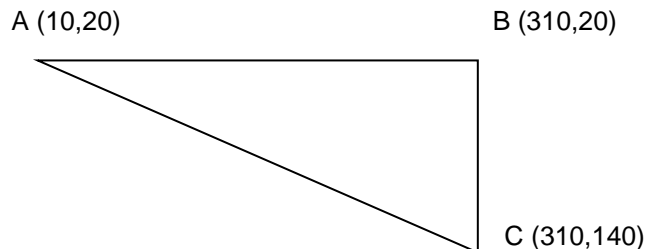
Om dit te realiseren wordt gebruik gemaakt van methoden met parameters. Het is een goede gewoonte om elke actie die mogelijk meer dan eens moet worden uitgevoerd in een methode te zetten. Dit heeft meerdere voordelen. Het eerste voordeel is, dat je dit deel van het programma maar één keer hoeft te schrijven. Wanneer je het niet apart in een methode zet, moet je het programmadeel steeds herhalen wanneer je het nodig hebt. Hoewel dit met 'kopiëren en plakken' heel snel kan gebeuren, wordt dit toch echt als 'dirty' beschouwd. Wanneer er iets verandert in je programma, moet je deze verandering namelijk meerdere keren doorvoeren met alle risico dat je het een keer vergeet. Dus... Gebruik methoden. Wijzigingen hoeven dan maar één keer te worden doorgevoerd en zijn dan ook voor elk gebruik doorgevoerd.

Een voorbeeld:

Stel je hebt de coördinaten van een driehoek. Dat wil zeggen, je kent van elke hoek de X-coördinaat en de Y-coördinaat. Jouw programma moet de oppervlakte van deze driehoek berekenen.

Dan zal je een aantal stappen moeten zetten. Om de oppervlakte te kunnen berekenen heb je de lengte van alle 3 de zijden nodig.

Die zal je moeten berekenen uit de coördinaten:



Stel hoek A ligt op de x coördinaat 10 en op y-coördinaat 20.

Hoek C ligt op x-coördinaat 310 en op y-coördinaat 140.

De lijn van A naar B is dan  $310 - 10 = 300$ .

De lijn van B naar C is dan  $140 - 20 = 120$ .

Om de lengte van de lijn van A naar C te bepalen maken we de volgende berekening (Stelling Pythagoras)

300 in het kwadraat =	90.000
120 in het kwadraat =	14.400+
	=====
Totaal	104.400

Vierkantswortel uit  $104.400 = 323,11$

We declareren een veld voor de uitkomst en maken voor deze berekening een Methode:

**double berekendeLengte;**

```
void berekenLengte(int aX, int aY, int bX, int bY) {  
    double aL = pow(bX - aX,2);  
    double bL = pow(bX - bY,2);  
    berekendeLengte = round( sqrt((float)(aL + bL)));  
}
```

Wanneer we nu de oppervlakte van een driehoek moeten berekenen op basis van de coördinaten declareren we een veld oppervlakte en voeren de volgende methode uit om deze te berekenen en ook te tekenen als een rode driehoek.

```
float oppervlakte = 0F;
```

```
void tekenEnBerekenOppervlakte(int aX, int aY, int bX, int bY, int  
                                cX, int cY) {  
    fill (255,0,0);  
    triangle(aX, aY, bX, bY, cX, cY);  
    berekenLengte(aX, aY, bX, bY);  
    double a = berekendeLengte;  
    berekenLengte(aX, aY, cX, cY);  
    double b = berekendeLengte;  
    berekenLengte(cX, cY, bX, bY);  
    double c = berekendeLengte;  
    double s = (a+b+c) / 2.0D;  
    double aD = s - a;  
    double bD = s - b;  
    double cD = s - c;  
    s = s * aD * bD * cD;  
    oppervlakte = sqrt( (float) s);  
}
```

Je ziet dat de berekening van de zijde gebeurt door middel van de methode *berekenLengte()* waarbij deze methode steeds wisselende parameters meekrijgt. Het resultaat wordt in een hulpveldje gezet en elke zijde krijgt de waarde hiervan vervolgens toegewezen.

Waarom moet het veld *berekendeLengte* buiten de methode worden gedeclareerd?

De nu opgedane kennis moet voldoende zijn om de opgaven bij hoofdstuk 4 te maken.



## 5 Returnwaarden

In hoofdstuk 4 heb je gezien hoe het gebruik van methoden met parameters, de mogelijkheid biedt om herhalende code te voorkomen. Ook heb je kunnen zien hoe het gebruik van het juiste bereik je programma overzichtelijk kan houden wanneer het ook groter wordt.

We eindigden het hoofdstuk echter met de vraag waarom we een hulpveld buiten de methode (dus voor het gehele programma) moesten declareren. Dit wordt beschouwd als vervuilend.

Om dit probleem op te lossen gaan we in dit hoofdstuk returnwaarden toevoegen aan de zelf gemaakte methoden. Deze helpen je nog beter om je uiteindelijke programma zo te structureren dat het ook nog overzichtelijk blijft wanneer het groter wordt.

Allereerst ga je werken aan je mindset om het belang van deze aanpak zichtbaar te maken. Dit is het eerste deel van deze course dat gaat over Analyse en Ontwerp.

### 5.1 De mindset

Even een stukje herhaling: In de eerste hoofdstukken heb je al meteen ervoor moeten kiezen om waarden die afleidbaar zijn ook daadwerkelijk af te leiden en niet zelf het rekenwerk voor de computer uit te voeren. Belangrijk was (en is) dat je bij een wijziging in je programma zo weinig mogelijk hoeft aan te passen.

Dit laatste geldt niet alleen voor variabelen maar zeker ook voor programma-instructies. In module 4 is er een methode gemaakt voor de berekening van de lengte van een zijde op basis van de x- en y-coördinaten van de uiteinden. Hierdoor voorkom je dat je dit stuk code 3 keer moet uitschrijven. Natuurlijk gaat dat 3 keer uitschrijven met Copy&Paste heel snel. Toch is het onwenselijk omdat je bij een programmawijziging in dat geval de code op 3 verschillende plaatsen moet aanpassen. Het is dus belangrijk om bij het analyseren van je probleem dit goed genoeg op te delen in steeds kleinere deelprobleempjes. Neem als stelregel aan dat, op het moment dat je code gaat kopiëren binnen je programma, je beter een stapje terug kunt zetten en even nadenkt over een betere oplossing.

In het voorbeeld van hoofdstuk 4 is er nog (minimaal) één storende factor aanwezig. Er is een veld, *berekendeLengte*, dat eigenlijk maar heel even onthouden moet worden steeds tijdens de uitvoering van ***tekenEnBerekenOppervlakte()***. Wanneer het programma wat groter wordt, heb je eigenlijk te maken met een veld dat het grootste deel van de tijd een beetje rond ligt te slingeren. Daarmee is het feitelijk betekenisloos en werkt het eerder verwarrend dan dat het ergens mee helpt.

### 5.2 Het principe van returnwaarden

Bij de beschrijving van velden heb je gezien dat je een veld van een bepaald type moet declareren. Gehele getallen zet je bijvoorbeeld in een int (of een long als ze heel groot kunnen worden). Zo heb je een float (of double) voor getallen met cijfers achter de komma, een String voor stukjes tekst, een boolean voor een ja/nee veld en nog vele andere mogelijkheden.

Deze type-declaratie geldt echter niet alleen voor velden. Ook methoden kunnen op deze manier van een bepaald type zijn.

Tot nu toe heb je alleen methoden gemaakt die begonnen met de term *void*. Void is het Engels woord voor “zonder waarde”. Tsja, wanneer het nodig is om dit ervoor te zetten, zullen er ook wel methodes moeten zijn die niet zonder waarde zijn. Logisch toch?

Dat is ook inderdaad zo. Een methode kan net als een veld heel veel verschillende soorten waarden “bevatten”.

Stel we hebben een groet-programma Afhankelijk van het tijdstip van de dag moet de juiste groet worden gegeven. De tekst moet vervolgens aan een andere tekst worden vastgeplakt.  
In hoofdstuk 4 zou dit nog zo gaan:

```
String groet;

void maakGroet() {
    if (hour() > 23 || hour() < 5) {
        groet = "Goede Nacht";
    } else if (hour() < 12) {
        groet = "Goede Morgen";
    } else if (hour() < 18) {
        groet = "Goede Middag";
    } else {
        groet = "Goede Avond";
    }
}

void groet(String naam) {
    maakGroet();
    groet = groet + " " + naam;
    text(groet, 100,100);
}
```

We hebben nu wel weer keurig 2 methoden gemaakt maar het veld *groet* slingert nog rond terwijl we in het geheel geen behoefte hebben aan het onthouden van deze *groet*. We kunnen dit stukje programma dan ook anders schrijven:

```
String maakGroet() {
    String groet;
    if (hour() > 23 || hour() < 5) {
        groet = "Goede Nacht";
    } else if (hour() < 12) {
        groet = "Goede Morgen";
    } else if (hour() < 18) {
        groet = "Goede Middag";
    } else {
        groet = "Goede Avond";
    }
    return groet;
}

void groet(String naam) {
    String groet = maakGroet();
    groet = groet + " " + naam;
    text(groet, 100,100);
}
```

Dit stukje programma doet precies hetzelfde. Alleen zie je dat de *String groet* in beide methoden lokaal gedeclareerd is. Na beëindiging van de methode *maakGroet()* is het veld *groet* in deze methode dus verdwenen. Het veld *groet* in de methode *groet()* is een ander veld. Hier kennen we de returnwaarde van de methode *maakGroet()* aan toe. Dit veld wordt verder gecombineerd. Het resultaat wordt getoond en het veld wordt vergeten en vervuult je programma ook niet meer.

Met een methode van het type *String* kun je precies het zelfde doen als met

een String veld. Daarom kunnen we de methode `groet()` ook zo schrijven:

```
void groet(String naam) {  
    String groet = maakGroet() + " " + naam;  
    text(groet, 100,100);  
}
```

En voor de echte luiakken doen we het zo:

```
void groet(String naam) {  
    text(maakGroet() + " " + naam, 100,100);  
}
```

Het wordt er een stuk korter van en de functionaliteit is niet veranderd. Er blijven geen onnodige variabelen meer rondslingeren. Dus als we elders in ons programma weer een keer een String met de naam *groet* willen gebruiken kan dit gewoon.

### 5.3 Returnwaarden toepassen op de driehoek

We komen nu bij de eindvraag van hoofdstuk 4. We gaan de oppervlakteberekening van de driehoek zodanig uitprogrammeren dat er geen onnodige variabelen rond blijven slingeren.

Hiervoor passen we het programma aan en gaan we werken met `returnvalues`.

We moesten een veld bewaren om het resultaat van de berekende lengte even vast te houden. Wanneer we de methode *berekenLengte()* het eindresultaat laten retourneren, komt deze waarde direct terug in de aanroepende methode.

We hadden het volgende:

```
double berekendeLengte;  
  
void berekenLengte(int aX, int aY, int bX, int bY) {  
    double aL = pow(bX - aX,2);  
    double bL = pow(bX - bY,2);  
    berekendeLengte = round( sqrt( (float) (aL + bL) ) );  
}
```

Het veld *berekendeLengte* is straks niet meer nodig. Dat halen we weg. We laten de methode een double retourneren. We gebruiken een long omdat de maat gegeven is in aantal pixels. En halve pixels kunnen niet oplichten. Het resultaat is als volgt:

```
double berekenLengte(int aX, int aY, int bX, int bY) {  
    double aL = pow(bX - aX,2);  
    double bL = pow(bX - bY,2);  
    return round( sqrt ( (float) (aL + bL) ) );  
}
```

Wanneer we nu de oppervlakte van een driehoek moeten berekenen op basis van de coördinaten deden we in hoofdstuk 4 het volgende :

```
double oppervlakte = 0D;
```

```

void tekenEnBerekenOppervlakte(int aX, int aY, int bX, int bY, int
                                cX, int cY) {
    fill (255,0,0);
    triangle(aX, aY, bX, bY, cX, cY);
    berekenLengte(aX, aY, bX, bY);
    double a = berekendeLengte;
    berekenLengte(aX, aY, cX, cY);
    double b = berekendeLengte;
    berekenLengte(cX, cY, bX, bY);
    double c = berekendeLengte;
    double s = (a+b+c) / 2.0D;
    double aD = s - a;
    double bD = s - b;
    double cD = s - c;
    s = s * aD * bD * cD;
    oppervlakte = sqrt( (float) s);
}

```

We willen die oppervlakte eigenlijk alleen tonen en verder niet onthouden. Verder zie je overal het veld *berekendeLengte* nog staan. Dit veld kunnen we vervangen door de betreffende methode. Ook gaan we de berekende oppervlakte teruggeven aan de aanroepende methode.

De methode ziet er dan als volgt uit:

```

double tekenEnberekenOppervlakte(int aX, int aY, int bX, int bY,
                                int cX, int cY) {
    fill (255,0,0);
    triangle(aX, aY, bX, bY, cX, cY);
    double a = berekenLengte(aX, aY, bX, bY);
    double b = berekenLengte(aX, aY, cX, cY);
    double c = berekenLengte(cX, cY, bX, bY);
    double s = (a+b+c) / 2.0D;
    double aD = s - a;
    double bD = s - b;
    double cD = s - c;
    s = s * aD * bD * cD;
    return sqrt( (float) s);
}

```

Los van de meetkundige berekening (welke je uit je middelbare schooltijd natuurlijk zo uit je mouw schudt ☺) , zie je dat voor elke zijde van de driehoek de methode *berekenLengte()* wordt gebruikt.

Eerder is gesproken over de scope van variabelen. Je ziet dat voor deze ingewikkelde berekening best veel hulpvariabelen worden gebruikt terwijl de gebruiker van deze methode alleen de oppervlakte wil weten. Al die tussenresultaten zijn verder niet interessant. Daarom zijn deze lokaal gedeclareerd en dus weer vergeten wanneer deze methode zijn klus geklaard heeft.

Wanneer de oppervlakte is berekend mag de lengte van elke zijde ook worden vergeten. We hadden al bedacht dat dit 3 keer moet gebeuren, en dus in de aparte methode *berekenLengte()*. Alleen moet de uitkomst van deze berekening nu nog in de hiertoe gedeclareerde variabele terecht komen. Je ziet dat er een lokale variabele *a* wordt gedeclareerd. Hier wordt de uitkomst van de methode *berekenLengte()* aan toegekend. Dit is de zogenaamde *return value*.

Verder zie je dat de methode *berekenOppervlakte()* 6 parameters heeft. Dit zijn precies de coördinaten van de drie hoeken van een driehoek. Hierdoor is

*return value*

deze methode voor elke driehoek opnieuw te gebruiken. Dit voorkomt dat je voor verschillende driehoeken steeds hetzelfde kunstje moet uithalen.

Je kunt ook dit natuurlijk inkorten:

```
double tekenEnberekenOppervlakte(int aX, int aY, int bX, int bY,
                                  int cX, int cY)
{
    fill (255,0,0);
    triangle(aX, aY, bX, bY, cX, cY);
    double s = ( berekenLengte(aX, aY, bX, bY)
                + berekenLengte(aX, aY, cX, cY)
                + berekenLengte(cX, cY, bX, bY)) / 2.0D;
    double aD = s - berekenLengte(aX, aY, bX, bY);
    double bD = s - berekenLengte(aX, aY, cX, cY);
    double cD = s - berekenLengte(cX, cY, bX, bY);
    s = s * aD * bD * cD;
    return sqrt( (float) s);
}
```

Je moet je echter dan wel oprecht afvragen of je programma hierdoor niet inboet aan leesbaarheid.

Tijd voor oefeningen!

## 6 Herhalingen

Soms wil je zaken herhaald uitvoeren. Dat kan bijvoorbeeld zijn omdat het programma 10 keer een veld wil laten invoeren door een gebruiker. Ook kan het zijn dat je van tevoren niet weet hoeveel getallen de gebruiker gaat invoeren.

### 6.1 Soorten herhalingen

In programmering kennen we verschillende herhalingen. In bovenstaand stukje worden al twee verschillende soorten herhaling genoemd. Wanneer je iets een aantal keren uitvoert, zoals het uitvragen van 10 keer een getal, dan noemen we dat een *tellende herhaling*. Je kunt in dit geval tot 10 tellen en als je uitgeteld bent, ben je ook klaar:

```
teller = 1;
zolang teller <= 10 {
    doelets;
    teller = teller + 1;
}
```

Behalve deze soort heb je ook herhalingen die uitgevoerd worden als aan een bepaalde voorwaarde wordt voldaan. Dit noemen we een *voorwaardelijke herhaling*.

```
zolang totaal < 100 {
    getal = vraagGetal();
    totaal = totaal + getal;
}
```

We weten hier niet wat de waarde van **totaal** is wanneer voor het eerst de vraag wordt gesteld **zolang totaal < 100**. Wanneer de waarde al 100 of groter is, wordt het programmadeel binnen de herhaling geen enkele keer uitgevoerd. We zeggen dan ook wel dat deze herhaling 0 of meer keer wordt uitgevoerd.

Soms wil je echter dat een herhaling minimaal 1 keer wordt uitgevoerd en daarna zo vaak totdat een gewenste situatie is bereikt. Dit geldt bijvoorbeeld voor normale gebruikersinvoer. Gebruikers van computerprogramma's kunnen allerlei rare dingen invoeren waar jouw programma niets mee kan. Deze invoer ga je vervolgens valideren. Wanneer de invoer niet in orde is, vraag je opnieuw om invoer, eventueel vergezeld van een foutmelding. Je weet dus van tevoren niet hoeveel pogingen een gebruiker nodig heeft om tot een geldige invoer te komen maar je weet wel dat dit minimaal 1 zal zijn. Ook dit is een *voorwaardelijke herhaling* maar deze komt 1 of meer keer voor.

```
doe {  
    invoer = vraagInvoer();  
    invoerOK = valideerInvoer(invoer);  
} zolang als invoerOK = false
```

Zoals je ziet, vindt de voorwaardencontrole pas aan het eind plaats wanneer het stukje programma al een keer doorlopen is.

Dus samenvattend hebben we 3 soorten herhalingen:

- Tellende herhalingen;
- Voorwaardelijke herhalingen die 0 of meer keer voorkomen;
- Voorwaardelijke herhalingen die 1 of meer keer voorkomen;

## 6.2 De tellende herhaling

In Processing wordt de tellende herhaling geïmplementeerd middels een zogenaamde *for-lus*. Deze *for-lus* kent de volgende Syntax:

```
for(initiële situatie ; voorwaarde ; aanpassing situatie) {  
    instructie;  
}
```

Dit is wel heel algemeen geformuleerd. Hieronder wordt een voorbeeld gebruikt hoe dit er zou kunnen uitzien:

```
for(int i = 0; i < 10 ; i++) {  
    println((i+1) * 6);  
}
```

Uitleg bij het voorbeeld.

*for* is het sleutelwoord.

Tussen haakjes staan eigenlijk 3 parameters, gescheiden door een puntkomma.

De eerste parameter *int i = 0* declareert en initialiseert een int variabele met de naam *i* en zet de initiële waarde op 0. Deze variabele is de teller van tellende herhaling.

De tweede parameter, *i < 10*, beschrijft de voorwaarde waaronder de onderstaande lus moet worden uitgevoerd. In dit geval is het dus dat de lus moet worden uitgevoerd zolang *i* kleiner is dan 10.

De derde parameter *i++* beschrijft de actie die moet worden doorgevoerd nadat de lus is doorlopen. In dit voorbeeld wordt de teller *i* steeds met 1 opgehoogd.

Bovenstaande lus wordt dus in totaal 10 keer uitgevoerd.

Zoals je ziet wordt er met de *i* ook binnen de lus gerekend. Dat kan handig zijn om bijvoorbeeld de tafels af te drukken. Deze lus drukt dus de tafel van 6 af.

### 6.3 De voorwaardelijke herhaling 0 of meer keer

In Processing wordt de voorwaardelijke herhaling geïmplementeerd middels een *while-lus*. Deze lus kent de volgende syntax:

```
while (voorwaarde) {  
    instructie;  
}
```

Alles wat met een for-lus kan ook met een while-lus. Soms is het echter wel omslachtiger. Dat komt omdat deze lus voor iets anders bedoeld is. Wanneer ik bovenstaande for-lus wil omzetten naar een while-lus gaat dit als volgt:

```
int i = 0;  
while (i < 10) {  
    println((i+1) * 6);  
    i++;  
}
```

Zoals je ziet is het resultaat weliswaar bijna hetzelfde maar eigenlijk is dit niet handig. Bedenk zelf wat het verschil is nadat de lus doorlopen is.

De while-lus leent zich ook meer voor een situatie waarvan je de start niet van tevoren weet. Stel je wil een kompasnaald laten draaien naar het nul graden punt (het noorden). Voor deze naald bestaan 3 methoden: *void draaiGraadNaarLinks()*, *void draaiGraadNaarRechts()* en *int geefRichting()*. Op het moment dat je met de actie start, heb je geen flauw idee wat de richting is.

Een logische oplossing waarbij we rekening houden met de kortste route naar het 0-punt zou zijn:

```
while (getRichting() != 0) {  
    if (getRichting() < 180) {  
        draaiGraadNaarLinks();  
    } else {  
        draaiGraadNaarRechts();  
    }  
}  
println("De Naald wijst nu naar het noorden");
```

Uitleg:

Eerst wordt een voorwaarde gecontroleerd: Zolang de richting niet 0 is. Wanneer bij de eerste aanroep de richting al 0 is, wordt de beschreven lus niet uitgevoerd en wordt direct gesprongen naar de laatste regel van dit fragment.

Vervolgens checkt de lus of de naald overwegend naar het oosten wijst. Zo ja, dan linksom draaien en anders rechtsom. Dit is i.v.m. de kortste route.

Het gebruik van deze structuur leidt er dus toe dat de kompasnaald de juiste richting uitdraait tot hij de gewenste positie heeft bereikt.

### 6.4 De voorwaardelijke herhaling 1 of meer keer

Soms kun je ook te maken hebben met zaken die minimaal 1 keer moeten gebeuren en eventueel vaker. Dit kan bijvoorbeeld zijn wanneer je om gebruikersinvoer vraagt. Je vraagt minimaal 1 keer en zolang de invoer niet door de beugel kan, herhaal je de vraag.

Ook kun je je voorstellen dat je een klein spelletje opstart, uitspeelt en dat het programma vraagt of je nog een keer wil spelen. De vraag of je nog een keer wil spelen zou toch wel een beetje belachelijk overkomen wanneer je het spel net hebt opgestart.

Voor dit soort situaties hebben we de volgende structuur:

```
do {  
    instructie1;  
    instructie2;  
    instructie3;  
    instructie4;  
} while (voorwaarde)
```

De lus die wordt beschreven in de instructies wordt eerst eenmaal uitgevoerd. Daarna wordt gecheckt via de voorwaarde of het nog een keer moet gebeuren. In het spelletjesvoorbeeld wordt het spelen van het spel in de lus uitgevoerd. Wanneer het spel klaar is, wordt gecontroleerd of aan een voorwaarde is voldaan alvorens nog een keer te spelen. (De vraag 'nog eens' is met Ja beantwoord).

Voorbeeld: We hebben een spel balletje-balletje. Wanneer het spel opstart, toont het een beker met een balletje en twee lege bekertjes. Vervolgens wisselen de bekertjes gedurende enkele seconden steeds van plaats. Daarna vraagt het spel aan te klikken onder welke beker het balletje ligt. Na dit raden ben je je geld kwijt of heb je gewonnen. Het spel werkt je saldo bij en vraagt je of je nog een keer wil spelen (onder voorwaarde dat je nog saldo hebt.)

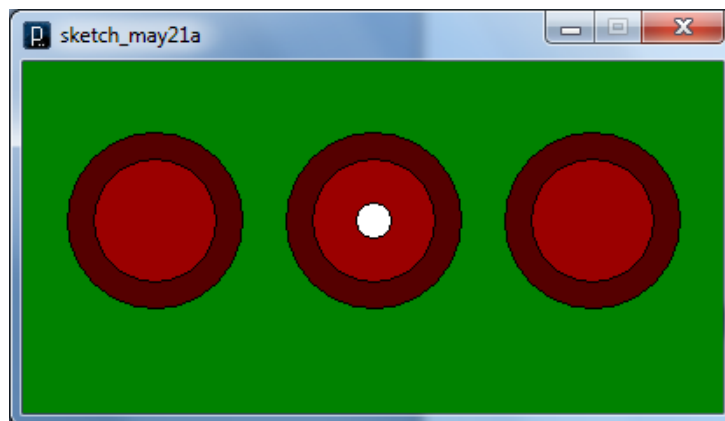


De loop van het spelletje is dan als volgt:

```
toonBekersMetBalletje();  
int beginSaldo = 100;  
int inzet = 25;  
do {  
    schuifBekers();  
    vraagBekerAanwijzen();  
    toonBekersMetBalletje();  
    verwerkResultaat(geradenBeker, juisteBeker);  
} while (saldo > 0 && nogEenKeer());
```

Uitleg:

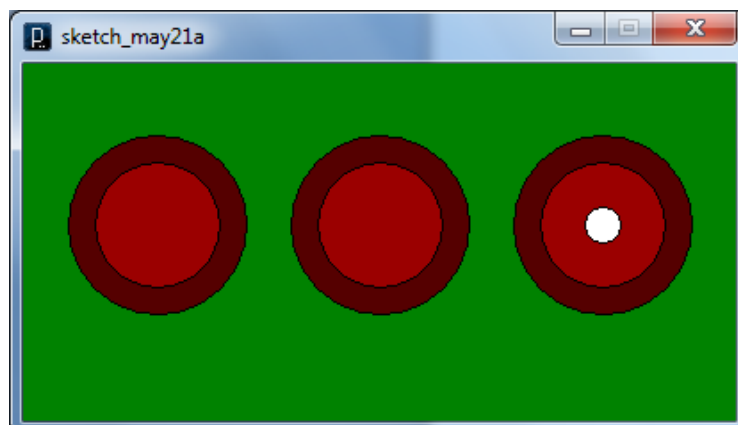
In de methode *toonBekersMetBalletje()* worden de 3 bekertjes getoond in helikopterview:



Je ziet dat onder de middelste beker het balletje ligt.

De methode *schuifBekers()* verbergt het witte bolletje, verwijdert steeds 2 bekertjes uit de view en toont ze weer. Dit betekent dat deze bekertjes van plaats verwisseld zijn en het balletje dus onder een andere beker ligt. Dit gebeurt natuurlijk snel en na enkele seconden is het balletje enkele 10-tallen keren van plaats verwisseld. Het programma reageert in die tijd niet op mouseklikken. In *vraagBekerAanwijzen()* reageert het programma wel en een mouseklik in één van de drie zones betekent een keuze voor 1, 2 of 3.

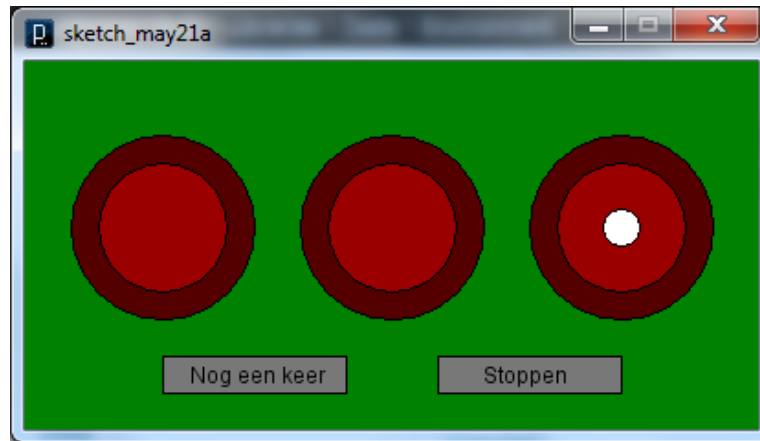
Hierna wordt het beginscherm getoond met de juiste plek:



Jammer wanneer je op een van de andere bekertjes had gedrukt.

Wanneer je goed gekozen had, krijg je er 25 bij en anders gaan er 25 af van je saldo.

Wanneer het saldo nu voldoende is krijg je het volgende scherm:



Dit is de methode *nogEenKeer()*. Wanneer je op de knop met de tekst “Nog een keer” klikt, herhaalt de lus zich.

## 6.5 In het kort

In dit hoofdstuk zijn er verschillende herhalingsstructuren (programmalussen) besproken. Welke structuur je kiest, is afhankelijk van het probleem dat je moet oplossen:

- Wanneer je een lus een vooraf bepaald aantal keren wil doorlopen, kies je de *for(...)*-lus;
- Wanneer je van te voren niet weet of een lus doorlopen moet worden en je ook niet weet hoe vaak dit moet, kies je voor een *while(...)*-lus;
- Wanneer een lus minimaal één keer maar verder een onbekend aantal keren moet worden doorlopen kies je voor de *do ..... while(.....)*-lus.

## 7 Arrays

Soms (of eigenlijk best vaak) heb je een serie variabelen nodig van hetzelfde soort om meerdere zaken in te bewaren. Denk hierbij bijvoorbeeld aan een serie getallen of een rij woorden.

Wanneer je 100 getallen wil bewaren zou het natuurlijk erg onhandig zijn wanneer je hier voor 100 variabelen moet declareren: `getal1`, `getal2`, `getal3`.....etc". Hier hebben we gelukkig een andere oplossing voor. We kunnen deze velden dan in een tabel zetten. Het Engelse woord hiervoor is Array, dus dat gebruiken we hier.

### 7.1 Enkelvoudige Arrays

De eenvoudigste Array is de zogenaamde enkelvoudige oftewel de eendimensionale Array.

Je kunt dit zien als rijtje gegevens. In onderstaand voorbeeld heb je een Array van 10 getallen.

25	36	11	87	115	1	38	95	42	65
----	----	----	----	-----	---	----	----	----	----

Wanneer we deze Array willen aanmaken declareren we deze Array eerst als volgt:

```
int[ ] getallen;
```

Hiermee hebben we gezegd dat we een Array van ints gaan gebruiken die we de naam *getallen* geven. We hebben nog niet opgegeven hoeveel getallen we erin willen zetten. Wanneer we dat wel willen doen moet dat als volgt:

```
int[ ] getallen = new int[10];
```

Hiermee hebben we de Array geïnitieerd en hebben we aangegeven dat de Array 10 getallen kan bevatten van het type int.

De getallen zitten er echter nog niet in.

Wanneer je bovenstaande Array in één keer wil declareren en initialiseren kan dit als volgt:

```
int[ ] getallen = { 25, 36, 11, 87, 115, 1, 38, 95, 42, 65 };
```

Nu heb je de Array gedeclareerd en gevuld. De volgende stap die je wil zetten is het benaderen van een element uit een Array. Dit gebeurt middels een index. Een index is niets anders dan een volgnummertje. Er is wel iets raars aan de hand met dit volgnummertje: Als mensen zijn we geneigd om bij bovenstaande Array van 10 elementen de volgnummers 1 t/m 10 te gebruiken. Dat gebeurt in Processing niet. In Processing beginnen we te tellen bij 0 (nul). In deze Array hebben we dus volgnummers 0 t/m 9. Wanneer we uit de opgegeven Array dus het getal 87 willen uitlezen kunnen we dit benaderen middels ***getallen[3]*** en 65 wordt benaderd via ***getallen[9]***. Uiteraard kunnen we in plaats van een getal (3 of 9) ook een variabele gebruiken als index. Onderstaand voorbeeld laat zien hoe je de Array met een *for-lus* doorloopt en van alle getallen die erin staan en kleiner zijn dan 50 wordt het 5-voud afgedrukt. Hier komen we gelijk een handige eigenschap tegen van een Array: Je kunt aan een Array vragen hoeveel elementen erin passen met behulp van het veldje *arrayNaam.length*. (Hier dus *getallen.length*)

Je kunt nu dus gewoon met alle getallen uit de Array werken. Zoals de screencasts bij deze module laten zien kun je de getallen gebruiken voor allerlei manipulaties. De API van Processing biedt dan ook volop ondersteuning om dit zo goed mogelijk te gebruiken. Zie hiervoor de screencasts.

```
sketch_160512g
1 int[] getallen = { 25,36,11,87,115,1,38,95,42,65};
2 for (int i = 0; i < 10; i++){
3   if (getallen[i] < 50)
4     println(getallen[i] * 5);
5 }
6
7
8
9
10
11
12
13
14
15
16
17
18
```

```
125
180
55
5
190
210
```

Hierboven zie je het beschreven programma en de bijpassende output in de console.

Behalve dat je velden kunt uitlezen uit een tabel kun je er net zo mee manipuleren als met lossen velden. Stel dat de velden op positie 5 en 6 van plaats omgewisseld moeten worden. Dat kan niet in één keer maar daar heb je een hulpveldje voor nodig:

```
int bewaarveld = getallen[5];
getallen[5] = getallen[6];
getallen[6] = bewaarveld;
```

Op deze manier kun je een tabel sorteren (oplopend of aflopend). Hiervoor wordt verwezen naar de opgaven.

## 7.2 2-dimensionale Arrays

De screencasts tonen alleen manipulaties met 1-dimensionale Arrays.

Wanneer je echter naar een programma kijkt als MS Excel zie je op het eerste gezicht al een tabel in 2 dimensies. Langs de X-as staan de letters A, B, C etc. Terwijl langs de Y-as de getallen van 1 tot heel veel staan. Je adresseert elke cel dan middels een letter en een getal (bv. veld C3).

Op de volgende pagina staat een voorbeeld van een 2-dimensionale Array. In totaal heeft deze Array 10 rijen van 10 getallen. (In totaal 100 dus.)

25	36	11	87	76	1	55	91	42	65
22	38	14	85	78	4	60	95	49	67
21	33	18	81	74	7	51	99	45	63
28	35	16	86	80	6	57	93	41	69
27	39	17	90	72	3	54	92	50	61
29	40	13	84	77	9	56	98	48	66
24	32	19	88	73	2	53	94	43	70
23	31	12	83	79	8	58	100	47	68
30	37	15	89	75	10	52	97	44	62
26	34	20	82	71	5	59	96	46	64

In deze Array staan de getallen van 1 t/m 100 in een (deels) willekeurige volgorde. We spreken van een Array waarbij sprake is van een X-as en een Y-as. In Processing declareren we dat als volgt:

```
int[ ][ ] getallen;
```

Er worden nu twee paar blokhaken neer gezet. Eigenlijk betekent dit dat je een Array met daarin Arrays van ints declareert. Probeer maar eens uit wanneer je het aantal elementen opvraagt: `getallen.length` levert als resultaat 10 op. Vervolgens kun je ook vragen naar `getallen[0].length`. Dit levert ook weer 10 op. Je kunt, net als bij een één-dimensionale Array deze ook weer direct initialiseren. Dit gaat als volgt:

```
int[ ][ ] getallen = {
    {25, 26, 11, 87, 76, 1, 55, 91, 42, 65},
    {22, 38, 14, 85, 78, 4, 60, 95, 49, 67},
    {21, 33, 18, 18, 74, 7, 51, 99, 45, 63},
    {28, 35, 16, 86, 80, 6, 57, 93, 41, 69},
    {27, 39, 17, 90, 72, 3, 54, 92, 50, 61},
    {29, 40, 13, 84, 77, 9, 56, 98, 49, 66},
    {24, 32, 19, 88, 73, 2, 53, 94, 43, 70},
    {23, 31, 12, 83, 79, 8, 56, 100, 47, 68},
    {30, 37, 15, 89, 75, 10, 52, 97, 44, 62},
    {26, 34, 20, 82, 71, 5, 59, 96, 46, 64}
};
```

Om getallen hierin te benaderen heb je 2 indexen nodig. De eerste index bepaalt uit welke rij je het gegeven haalt en de tweede bepaalt wel element binnen die rij het wordt. Het getal 80 staat in 4<sup>de</sup> rij op de 5<sup>de</sup> positie. Dit benader je dus middels: **`getallen[3][4]`**.

Hieronder volgt een stukje programmacode dat een tweedimensionale tabel initialiseert, vervolgens op het scherm afdrukt en daarna nog een keer element `[3][4]` afdrukt om aan te tonen dat bovenstaande juist is.

```

1
2 size(300,250);
3 fill(0,0,0);
4
5 int[ ][ ] getallen = {
6 {25, 26, 11, 87, 76, 1, 55, 91, 42, 65},
7 {22, 38, 14, 85, 78, 4, 60, 95, 49, 67},
8 {21, 33, 18, 18, 74, 7, 51, 99, 45, 63},
9 {28, 35, 16, 86, 80, 6, 57, 93, 41, 69},
10 {27, 39, 17, 90, 72, 3, 54, 92, 50, 61},
11 {29, 40, 13, 84, 77, 9, 56, 98, 49, 66},
12 {24, 32, 19, 88, 73, 2, 53, 94, 43, 70},
13 {23, 31, 12, 83, 79, 8, 56, 100, 47, 68},
14 {30, 37, 15, 89, 75, 10, 52, 97, 44, 62},
15 {26, 34, 20, 82, 71, 5, 59, 96, 46, 64}
16 };
17
18 for (int i = 0; i < getallen.length; i++){
19     for (int j = 0; j < getallen[i].length; j++) {
20         text(getallen[i][j], (j+1)*25, (i+1)*20);
21     }
22 }
23 text(getallen[3][4],150,230);
24

```

Eerst wordt de grootte van het window aangepast en de letterkleur op zwart gezet.

Daarna wordt conform de tekst hierboven een tweedimensionale array gedeclareerd en geïnitieerd.

Hierna wordt in een dubbele for-lus door de Array heengelopen. De index *i* wordt voor de rijen gebruikt, de index *j* voor de elementen in de rij.

Tot slot wordt onderaan nog een keer apart element [3][4] getoond.

Het resultaat van het programma wordt hieronder onder getoond.

```

sketch_160512g
25 26 11 87 76 1 55 91 42 65
22 38 14 85 78 4 60 95 49 67
21 33 18 18 74 7 51 99 45 63
28 35 16 86 80 6 57 93 41 69
27 39 17 90 72 3 54 92 50 61
29 40 13 84 77 9 56 98 49 66
24 32 19 88 73 2 53 94 43 70
23 31 12 83 79 8 56 100 47 68
30 37 15 89 75 10 52 97 44 62
26 34 20 82 71 5 59 96 46 64

80

```

In het programma zie je dat voor de buitenste lus gebruik gemaakt wordt van `getallen.length` terwijl in de binnenste lus `getallen[i].length` wordt gebruikt. Dit heeft een reden: De hier aangemaakte tweedimensionale tabel is mooi vierkant. Echter, zoals vermeld is een tweedimensionale Array eigenlijk gewoon een Array van Arrays. Nergens staat voorgeschreven dat die Arrays allemaal even veel elementen

moeten hebben. Mocht de lengte tussen twee Arrays verschillen, dan is dat door de keuze van `length[i]` ondervangen.

NB: Wanneer je een ongeldige index gebruikt ( < 0 of > aantal elementen-1) krijg je een foutmelding.

### 7.3 Kubussen en nog meer dimensies

De wereld houdt niet op bij 2 dimensies. Wanneer we de maat van een kubus bepalen heb je met lengte, breedte en hoogte te maken. Hiervoor heb je 3 dimensies nodig. Op de volgende bladzijde staat een stukje broncode dat een onregelmatige 3-dimensionale tabel vult en afdruckt. Er staan debug-regels tussen. Hierdoor kun je zien waarom een regel op dat moment wordt uitgevoerd. Laat Processing dit uitvoeren kijk wat er in de console wordt afgedrukt.

```
int[ ][ ][ ] drie = {
{
  {1,2,3},
  {4,5},
  {7,8,9}
},
{
  {11,12,13},
  {14,15,16},
  {17,18}
}
}

for (int i = 0; i < drie.length; i++) {
  println("i = " + i + " drie.length = " + drie.length);
  for (int j = 0; j < drie[i].length; j++) {
    println("j = " + j + " drie[i].length = " + drie[i].length);
    for (int k = 0; k < drie[i][j].length; k++) {
      println("k = " + k + " drie[i][j].length = " + drie[i][j].length);
      println(drie[i][j][k] );
    }
  }
}
```

Hoewel wij als mensen niet meer dan 3 dimensies kunnen waarnemen heeft Processing er geen enkel probleem mee om nog meer dimensies in Arrays toe te passen. Probeer maar eens hoe ver je kunt gaan.

## 8 Van Probleem naar oplossing

Vaak hoor je studenten zeggen dat Java of Processing of PHP of welke programmeertaal dan ook moeilijk vinden. In de eerste 7 hoofdstukken van deze reader hebben jullie kunnen vaststellen dat een programmeertaal in het geheel niet lastig is. Engels leren is vele malen ingewikkelder dan een programmeertaal.

In werkelijkheid is niet de programmeertaal lastig maar het vinden van een oplossing voor een probleem wordt lastig gevonden.

### 8.1 De analyse, Wat moet het programma doen?

We beginnen meteen met de moeilijkste stap. In dit hoofdstuk gaan we dit behandelen aan de hand van een voorbeeld.

We willen de volgende lijst genereren:

Vak	OSG	CSG	GSG	MSG	VSG	KSG	Gemiddeld
Wis	7,4	6,9	7,2	8,1	7,3	6,4	7,2
Nat	7,6	6,7	7,1	8,8	7,6	6,1	7,5
Ned	7,5	8,4	7,8	6,9	7,1	8,1	7,6
Eng	8,0	7,6	7,7	7,5	7,0	7,7	7,8
Bio	7,8	6,1	5,8	8,6	7,9	6,6	6,9
AK	6,5	7,1	7,3	7,9	7,5	7,1	7,3

In deze lijst staan voor 6 vakken de gemiddelde resultaten van 6 scholen vermeld. In de laatste kolom staat het gemiddelde van alle leerlingen van deze 6 scholen vermeld. De scholen zijn natuurlijk niet allemaal even groot. De aangeleverde gegevens zijn de individuele cijfers behaald per school en per vak.

De opdracht samengevat in één zin is dus:

**Toon een tabel met daarin per school per vak het gemiddelde resultaat en één kolom met het gemiddeld resultaat over alle scholen.**

Dit klinkt ingewikkeld. Dat is het ook. Daarom moeten we het vereenvoudigen. Eigenlijk bestaat deze opdracht uit een aantal deelopdrachten. Die moet je er eerst uit analyseren:

- **Toon een tabel**
- **Bereken gemiddelde per school en per vak**
- **Bereken gewogen gemiddelde per vak over alle scholen**

De opdracht is al wat overzichtelijker geworden. Nu moet je jezelf wel een vraag stellen: Waar komen al die gegevens vandaan? Die gegevens zullen ongetwijfeld ergens vastgelegd zijn (bijvoorbeeld in een Database). Hoe we die gegevens in gaan lezen straks is van latere zorg. We beschrijven eerst nog WAT we moeten doen.

Bovenstaande opdracht kun je toch nog verder uitrafelen. De tabel die we gaan tonen moeten we namelijk ook vullen. Hierna ziet de opdracht er zo uit:

- **Vul de bovenste rij vanaf de tweede kolom met de namen van de scholen. Voeg tekst "Gemiddeld" toe in de laatste kolom op de eerste rij.**
- **Vul de linker kolom vanaf de tweede rij met de namen van de vakken**
- **Bereken gemiddelde per school en per vak**
- **Bereken gewogen gemiddelde per vak over alle scholen**
- **Vul elk veld op het kruispunt tussen school en vak met de gemiddelde score van deze school voor dit vak**



- **Vul elk veld op het kruispunt tussen “Gemiddeld” en vak met de gemiddelde totale gemiddelde voor dit vak**
- **Toon de gegevens**

Het is al weer wat verder uiteengerafeld maar wanneer je nu programmacode gaat schrijven, stuur je jezelf met een kluitje in het riet. Je weet namelijk nog een hele hoop niet. Je moet bijvoorbeeld vaststellen waar al die gegevens op het scherm komen te staan. Je zult per kolom de x-Coördinaat moeten berekenen en per rij de Y-coördinaat. En hoe ga je de gemiddelden bepalen? Laten we het bepalen van de gemiddelden eens wat meer uiteenrafelen.

- **Lees Cijfers per Vak per School**
- **Voor elk vak**
  - **Verwerk Scholen**
    - **Voor elke school**
      - **Verwerk cijfers**
      - **Voor elk cijfer tel dit op bij het totaal per vak per school**
    - **Deel totaal door het aantal cijfers van vak bij school**
    - **Tel totaal school op bij totaal vak**
  - **Tel totaal aantal cijfers school op bij totaal aantal cijfers per vak**
- **Deel totaal door totaal aantal cijfers**

Zonder dat we nu nog bepaald hebben hoe we dit gaan verwerken hebben we nu tamelijk gedetailleerd voorwerk gedaan door uiteindelijk een eenduidig proces neer te zetten. We zijn klaar voor de volgende stap!

## 8.2 Het Ontwerp. Hoe ga ik het maken?

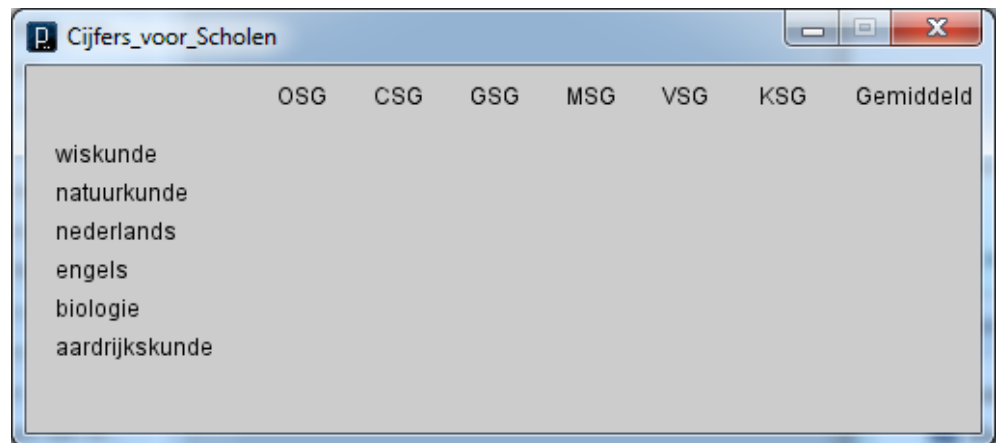
In hoofdstuk 1 t/m 7 zijn verschillende concepten aan de orde geweest. Het is goed om nu erover na te denken welke je hiervan gaat toepassen. In de opdracht is sprake van 6 scholen, 6 vakken en een aantal cijfers per vak per school. Hier kun je een driedimensionale tabel in herkennen. Uiteindelijk lever je wel een tweedimensionale tabel op maar de derde dimensie bestaat uit de individuele cijfers die leerlingen voor elk vak hebben behaald. Analooq aan het op te leveren rapport wordt de tabel als volgt geïndexeerd:

**int cijfers[vakvolgnr][schoolvolgnr][cijfervolgnr]**

Mmm.... Hier staan allemaal volgnummers. De gehele tabel moet worden door geakkerd. Een lus ligt voor de hand. Voordat je een lus in gaat weet je hoe vaak je hem moet doorlopen (lengte Array). Het schoolvoorbeeld van een tellende herhaling, dus een for-lus. Wanneer je nu toch met volgnummers gaat werken kun je voor het gemak nog 2 Arrays aanmaken:

**String[] scholen**  
**String[] vakken**

Wanneer je de naam in dezelfde volgorde zet als je het toepast in de cijfers Array kun je deze met de zelfde index benaderen. Als je het heel goed beheerst, zou je nu haast kunnen beginnen, ware het niet dat er nog een paar kleinigheidjes moeten worden opgelost. We moeten een beslissing nemen met betrekking tot de grootte van het window en de positionering van de labels en de velden. We doen dit bij voorkeur op een intelligente wijze. Intelligent wil zeggen dat we dit zo veel mogelijk door het programma laten berekenen. Waarom? Omdat je het niet opnieuw wil bedenken wanneer er een zevende vak of school bij komt. Je kent waarschijnlijk de pixels niet uit je hoofd, dus je mag hier eerst wat experimenteren. Je wil een scherm krijgen dat er ongeveer als volgt uit ziet:



Wat blijkt nu? We moeten weer even een stapje terugzetten naar de analyse! Wanneer we goed kijken dan kunnen we de volgende variabelen onderscheiden:

- Marges (links, rechts, boven en onder)
- Vaklabelbreedte (De naam van het vak heeft meer breedte nodig dan een cijfer)
- Schoollabelhoogte (Dit is een keuze voor de opmaak. Je kunt ook kiezen voor rijhoogte + extra marge)
- Kolombreedte
- Rijhoogte

Met behulp van deze variabelen kun je de positie van elk veld bepalen en ook de hoogte en breedte van het window.

Dus even op een rijtje: Wat moeten we doen om te beginnen?

- Bereken Schermbreedte
- Bereken Schermhoogte
- Plaats Scherm
- Teken Vak Labels
  - Voor elk vak
    - Bereken Y positie
    - Teken op linkermarge en Y positie
- Teken School Labels
  - Voor elke school
    - Bereken X positie
    - Teken op bovenmarge en X positie
  - Voor Gemiddelde kolom
    - Bereken X positie
    - Teken op bovenmarge en X positie

Uiteraard moeten we nog de achtergrondkleur en de letterkleur bepalen. Zoals je ziet hebben we het probleem vrijwel geheel opgelost zonder een letter te programmeren.

In ieder geval moet je bij je ontwerp ernaar streven dat je uiteindelijk alleen nog maar eenduidige opdrachten overhoudt. Op die manier creëer je uiteindelijk de meeste structuur in je programma.

### 8.3 Van Ontwerp naar Realisatie

Al die zaken die we tot nu toe bedacht hebben moeten natuurlijk ook in een programma passen. Dat begint met de declaratie van de velden en gaat verder met de implementatie van de bedachte methodes.

### 8.3.1 De velden declareren

Bij de velden moet je even goed nadenken welke velden je waar declareert.

```
final int vakLayoutBreedte = 115;
final int schoolLayoutHoogte = 30;
final int linkermarge = 15;
final int rechtermarge = 15;
final int bovenmarge = 20;
final int ondermarge = 20;
final int veldBreedte = 50;
final int veldHoogte = 20;
```

Bovenstaande velden zijn zogenaamde constanten. Het is niet de bedoeling dat je daar iets aan gaat wijzigen. Het is een goede gewoonte om dat af te dwingen (en ook duidelijk te maken) met het woord *final*. Deze velden declareer je in het algemeen ook buiten de methodes zodat ze in het gehele programma bruikbaar zijn.

Velden die je slechts op enkele plekken nodig hebt, houd je ook alleen daar beschikbaar. Hoe minder variabelen je tot je beschikking hebt, hoe minder verwarring.

### 8.3.2 Wanneer maak je een methode?

Als je nu terugkijkt naar het ontwerp, zie je dat er vaak een opdracht staat die iets verder moet worden uitgewerkt. Eigenlijk kun je stellen dat elke regel die nog om uitleg vraagt de aanroep van een methode betekent. Voor de `setup()` methode van je programma betekent dit dan:

```
void settings(){
    int breedte = berekenSchermbreedte(scholen.length);
    int hoogte = berekenSchermhoogte(vakken.length);
    size(breedte, hoogte);
}

void setup(){
    float [ ][ ] cijfers;
    String[ ] vakken = {"wiskunde", "natuurkunde",
                       "nederlands", "engels",
                       "biologie", "aardrijkskunde" };
    String[ ] scholen =
        {"OSG", "CSG", "GSG", "MSG", "VSG", "KSG"};
    fill(0,0,0);
    tekenScholen(scholen);
    tekenVakken(vakken);
}
```

Je ziet dat de Arrays pas hier gedeclareerd zijn. Van hier uit worden de andere methoden aangeroepen met precies die informatie die ze nodig hebben.

Het berekenen van de schermbreedte en –hoogte is weliswaar slechts 1 regel code. Echter omdat je er even over moet nadenken hoe dat moet, kun je dat beter in een aparte methode doen. Dan is er niets dat je afleidt. Om die reden is dus ook alleen de lengte van de scholen-Array als parameter meegegeven en niet de hele Array.

Al deze methoden die je schrijft zijn dan lekker kort en hebben in principe maar één taak. Als ze meerdere taken hebben, delegeer je deze weer aan andere methoden.

```
int berekenSchermbreedte(int aantal) {
    return linkermarge + vakLayoutBreedte + ((aantal + 1) * veldBreedte) +
    rechtermarge;
}

int berekenSchermhoogte(int aantal) {
```

```

        return bovenmarge + schoolLayoutHoogte + (aantal * veldHoogte) +
        ondermarge;
    }

```

```

void tekenScholen(String[ ] scholen) {
    for (int i = 0 ; i < scholen.length; i++) {
        text(scholen[i], getX(i), bovenmarge );
    }
    text("Gemiddeld", getX(scholen.length), bovenmarge);
}

```

```

void tekenVakken(String[ ] vakken) {
    for (int i = 0 ; i < vakken.length; i++) {
        text(vakken[i], linkermarge, getY(i));
    }
}

```

In de methode tekenScholen() wordt de X-coördinaat berekend op basis van de index. In het ontwerp stond: bereken X positie → Daar moet je even over nadenken dus daar maak je weer een aparte methode van. Het zelfde kun je zeggen over de Y-coördinaat in tekenVakken(); We hebben het window en labels nu staan. Nu de inhoud nog. Volgens analyse en ontwerp geven we de opdracht om de vakken te verwerken:

```

verwerkVakken(cijfers);

```

In verwerk vakken hoeft niet veel te gebeuren. Voor elk vak moeten de scholen verwerkt worden:

```

void verwerkVakken(int[][][] vakken) {
    for (int i = 0; i < vakken.length; i++) {
        verwerkScholen(vakken[i], i);
    }
}

```

Hier zie je iets handigs terugkomen. De driedimensionale Array vakken wordt in de eerste dimensie doorlopen. Door vervolgens vakken[i] (met één index) door te geven geef je aan de methode verwerkScholen een tweedimensionale Array mee als parameter. De index moet ook worden meegegeven omdat de methode moet weten op welke regel de gegevens moeten worden afgedrukt. In de analyse hebben we kunnen zien dat er in verwerkScholen wat meer moet gebeuren. Dat zien we hier onder:

```

void verwerkScholen(int[][] scholen, int vaknummer){
    int aantalLeerlingen = 0;
    int totaalScore = 0;
    for (int i = 0; i < scholen.length; i++) {
        int resultaat = verwerkCijfers(scholen[i]);
        float gemiddeld = (float)resultaat / scholen[i].length;
        text(opmaak(gemiddeld), getX(i), getY(vaknummer));
        aantalLeerlingen += scholen[i].length;
        totaalScore += resultaat;
    }
    float gemiddeld = (float)totaalScore / aantalLeerlingen;
    text(opmaak(gemiddeld), getX(scholen.length), getY(vaknummer));
}

```

Per vak moet je het gemiddelde berekenen over alle leerlingen van alle scholen samen. Dat totaal moet je dus bewaren. We lopen dus alle scholen door. Het totaal van alle cijfers is verwerkCijfers(scholen[i]). De lengte van scholen[i] is het aantal cijfers. Dit tonen we op de juiste plek. Alleen moet de

float gemiddeld nog netjes worden opgemaakt. De totalen worden bijgewerkt. Tot slot wordt hiervan het gemiddelde berekend en getoond. Dit was het meest complexe deel. Let wel op dat het scoreveld voor de deling naar een float moet casten. Anders doe je int/int en dat resulteert in een int!

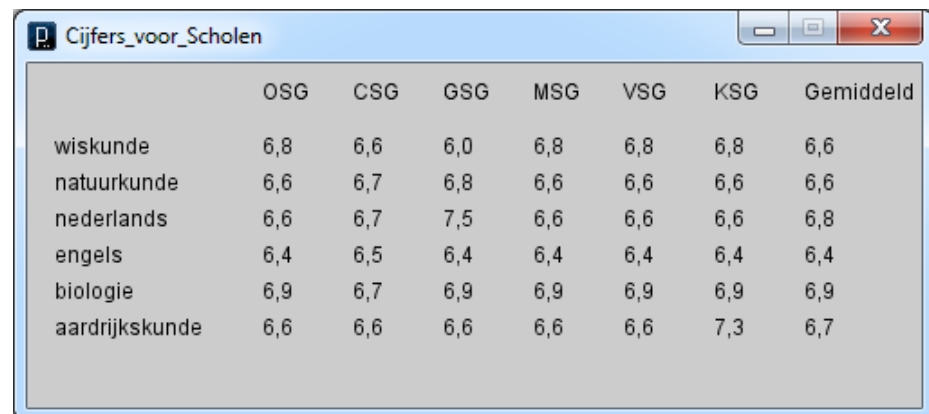
```
int verwerkCijfers(int[] cijfers) {  
    int totaal = 0;  
    for (int i = 0; i < cijfers.length; i++) {  
        totaal += cijfers[i];  
    }  
    return totaal;  
}
```

Zoals gezegd: verwerkCijfers is ook eenvoudig. Dan blijft de methode *opmaak()* over:

```
String opmaak(float getal) {  
    getal = getal * 10;  
    int rond = round(getal);  
    return(rond/10 + "," + rond%10);  
}
```

Hier is een eenvoudig trucje toegepast om eerst alle extra decimalen weg te werken. Omdat de int waarde nu het 10-voudige is van de werkelijke waarde kun je met een deling en een modulo bewerking keurig het getal voor- en na de komma bepalen.

Wanneer je er nu ook nog voor gezorgd hebt dat er wat cijfers in de tabel zijn gekomen, ziet het er als volgt uit:



	OSG	CSG	GSG	MSG	VSG	KSG	Gemiddeld
wiskunde	6,8	6,6	6,0	6,8	6,8	6,8	6,6
natuurkunde	6,6	6,7	6,8	6,6	6,6	6,6	6,6
nederlands	6,6	6,7	7,5	6,6	6,6	6,6	6,8
engels	6,4	6,5	6,4	6,4	6,4	6,4	6,4
biologie	6,9	6,7	6,9	6,9	6,9	6,9	6,9
aardrijkskunde	6,6	6,6	6,6	6,6	6,6	7,3	6,7

### 8.3.3 Conclusie

Tijdens het programmeren kom je nog genoeg dingen tegen die om een oplossing vragen. Wanneer je op dat moment de aangedragen problematiek nog moet oplossen. Maak je het je zelf moeilijk. Je vervalt dan snel in een trial-and-error methode die mogelijk tot een werkende oplossing leidt maar met meer inspanning en veel minder structuur. En wat nog belangrijker is: wanneer je op deze manier problemen structureert en oplost is het leereffect veel groter.

Daar komt nog iets bij. Stel dat ik nu een vak meer op het overzicht wil, hoef ik nu dit vak alleen toe te voegen aan de Array met vakken en ik moet resultaten opvoeren in de Array met cijfers. Met het volgend resultaat:

	OSG	CSG	GSG	MSG	VSG	KSG	Gemiddeld
wiskunde	6,8	6,6	6,0	6,8	6,8	6,8	6,6
natuurkunde	6,6	6,7	6,8	6,6	6,6	6,6	6,6
nederlands	6,6	6,7	7,5	6,6	6,6	6,6	6,8
engels	6,4	6,5	6,4	6,4	6,4	6,4	6,4
biologie	6,9	6,7	6,9	6,9	6,9	6,9	6,9
aardrijkskunde	6,6	6,6	6,6	6,6	6,6	7,3	6,7
geschiedenis	6,9	7,1	6,9	6,9	6,9	6,9	6,9

Het vullen van de gegevens was geen onderdeel van deze opdracht maar de echte liefhebber kan in de uitwerking wel bekijken hoe dat gedaan is.