READER
STRUCTURED PROGRAMMING BASICS_

Versie SP-B.1.1

M.C. van der Maas

15 augustus 2021

HAN_UNIVERSITY
OF APPLIED SCIENCES

VOORWOORD

Welkom bij de eerste course programmeren in het propedeusejaar van AIM. Deze course bestaat uit twee delen: Structured Programming Basics (SP-B) en Structured Programming Application Development (SP-AD) Tijdens SP-B leer je eerste basis van programmeren. In SP-AD gaan we dieper in op codekwaliteit en gaan we een grotere applicatie schrijven.

In de studiehandleiding kun je lezen wat je moet doen om deze course te halen. Deze reader is daar onderdeel van en vormt samen met de opgaven het basislesmateriaal voor deze course. Ze helpen je de eerste beginselen van programmeren eigen te maken. Het werkschema is op Onderwijs Online uitgewerkt.

Het programma wordt gecompleteerd door de colleges. In de colleges wordt ingegaan op de problemen die jij en je medestudenten aandragen, worden opdrachten gedaan en worden uitgevoerde opdrachten besproken. De theorie wordt uitgelegd aan de hand van de gemaakte opdrachten. Het is dus **belangrijk dat je voor elke les de bijbehorende theorie hebt bestudeerd en opdrachten doet.**

Veel succes!

Met vriendelijke groet,

Mark van der Maas
Coördinator Gestructureerd Programmeren

INHOUDSOPGAVE

INLEIDING

Wat is een computerprogramma

We staan er weinig bij stil, maar je bent constant in contact met allerlei computerprogramma's. Het bedienen van je mobiel gaat via een computerprogramma, maar ook het bedienen van een magnetron gaat niet zonder een computerprogramma. Bijna alle elektronica bevat een computer(tje) met daarop één of meerdere computerprogramma's.
Zo gebruiken we computerprogramma's om de temperatuur in ons huis te regelen, auto's (deels) te besturen, aandelen te verhandelen, tv te kijken, fabrieken te automatiseren, films te bewerken, drones te besturen, panden te beveiligen, het internet te browsen, documenten te lezen, te gamen en nog veel meer. Deze computerprogramma's zijn ontwikkeld door teams van mensen, zodat ze uit eindelijk door gebruikers, zoals wij allemaal, ingezet kunnen worden.
Het ontwikkelen van een computerprogramma (of kortweg: programma) gaat in verschillende stappen. Deze stappen gaan we ook leren in deze opleiding. De eerste stap is dat er wordt nagedacht over wat het programma allemaal moet kunnen en hoe het moet reageren op gebruikersacties. Dit wordt vastgelegd in de eisen van het programma. Vervolgens wordt er een ontwerp gemaakt om uit te zoeken wat, gegeven de eisen, de beste vorm is van het programma. Aan de hand van dit ontwerp wordt het programma in code uitgewerkt. Dit eindresultaat wordt vervolgens getest om vast te stellen of het aan alle vooraf gestelde eisen voldoet.

Het schrijven van een programma

Een computerprogramma bestaat uit een aantal *instructies* (commando's) die de computer één voor één uitvoert. Om een computerprogramma te laten werken, moet de computer dus de juiste instructie op het juiste moment uitvoeren. Dit betekent dat het schrijven van een programma nauwkeurig moet gebeuren. Vooral omdat een computer enkel exact doet wat je hem opdraagt.

Wanneer je aan een Nederlandssprekend persoon vraagt 'Hoeveel is 4 + 8?' zal de gemiddelde persoon '12' antwoorden. Wanneer je een computer de opdracht geeft om 4+8 uit te rekenen, zal deze dat doen, maar ook niet meer dan dat. Als gebruiker zie je dus niet wat er is gebeurd. Sterker nog, als je de computer niet vertelt dat hij de uitkomst moet onthouden, dan vergeet hij het ook direct weer. Als je de uitkomst op het scherm wil zien, dan moet je ook die instructie aan de computer geven.

Maar dat is niet het enige waar je rekening mee moet houden als je een programma schrijft. Wanneer je een mens de opdracht geeft om een stap te zetten, dan zal deze controleren of dit wel kan en zet dan pas die stap. Als jouw karakter in een computerspelletje een stap moet zetten, dan moet je alle *controles* heel precies uitschrijven om te voorkomen dat de stap in een muur (of in een ravijn) eindigt. Ook moet je nog duidelijk maken in welke richting de stap moet worden gezet.

Deze voorbeelden laten zien dat een computer eigenlijk heel dom is. Een computer lijkt enkel slim, omdat het heel snel de door mensen geschreven instructies kan uitvoeren.

Processing en Java

In deze course maken we gebruik van de programmeertaal '*Processing*'. Er bestaan echter veel meer programmeertalen. Zomaar een greep:

- C
- C++
- PHP
- Ruby
- C#
- Perl
- Python
- Visual Basic
- Haskell
- Prolog
- Java

Processing is een programmeertaal en ontwikkelomgeving die is ontwikkeld 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.
Java is een programmeertaal die bedacht is door een grote computerfabrikant: Sun. Inmiddels is Sun overgenomen door Oracle (vooral bekend van zijn databasesystemen).
De gedachte achter Java is simpel: *Write once, run everywhere.* Tijdens het ontwikkelen van de taal is er veel rekening gehouden dat een geschreven Java-programma op allerlei type computers moet kunnen draaien. Zo is er voor Java ook nog een *micro edition*. Deze maakt het mogelijk Java-programma's te draaien op telefoons, robotstofzuigers, wasmachines en thermostaten en andere apparaten om ons heen die steeds *intelligenter* worden.
Java wordt gebruikt in de programmeercourse OOPD in de propedeuse en ook in het semester *OOSE* wordt in Java geprogrammeerd.
Nog een kleine opmerking: Java wordt vaak verward met JavaScript. Hoewel de naam anders doet vermoeden hebben beide talen niets met elkaar te maken.

Van geschreven naar werkend programma

Wanneer je zelf een programma schrijft, zijn de instructies die je hebt geschreven voor jou wel leesbaar. Voor de computer is dit echter nog abracadabra. Deze begrijpt immers alleen enen en nullen.

De geschreven programmacode moet daarom worden omgezet naar een *instructieset* die voor de computer wel begrijpelijk is. Dit noemen we *compileren.*

De programmeeromgeving

Programmacode (kortweg: code) is feitelijk een stukje tekst dat voldoet aan de regels van de programmeertaal (syntax). Dit kun je in principe met elke tekstverwerker doen, als deze maar in staat is om de tekst zonder vreemde symbolen als platte tekst op te slaan. De meeste tekstverwerkers zijn er echter niet op gemaakt, dus comfortabel is anders.
Gelukkig zijn er ook specifieke programma's die wel gemaakt zijn om code in te schrijven. Zo'n programma noemen we een *geïntegreerde ontwikkelomgeving*. De afkorting van het Engelse woord hiervoor is IDE (Integrated Development Environment).
In deze course wordt gebruikt gemaakt van de IDE van *Processing*. Deze biedt al heel opties die programeren makkelijker maken. Er zijn nog veel meer IDE's, maar l**aat jezelf (nog) niet verleiden tot het gebruiken van andere IDE's zoals *Eclipse* of *Netbeans*. Deze bevatten zoveel functionaliteit dat ze het zicht wegnemen op wat je hier moet leren.**

Processing

Zoals gezegd programmeren we in deze course in de IDE Processing en met de programmeertaal Processing. Processing is dus de naam van zowel de programmeeromgeving als de programmeertaal. Beide kun je in 1 pakket van het internet downloaden. Dit moet je doen voordat je de eerste opdracht maakt kunt maken. Op onderwijsonline onder het kopje "processing" staat waar je de juiste versie van Processing kunt vinden (de versie zelf staat in de studiehandleiding) en hoe je deze installeert.

1 PROGRAMMEREN EN CODEREN

1.1 Kennis maken met Processing

Een programma bestaat in feite uit niets anders dan een grote hoeveelheid instructies waarmee we gegevens gebruiken of aanpassen. Kleinere groepen instructies met een bepaalde samenhang groeperen we. Binnen Processing noemen we deze groepen instructies methoden[1]. Hieronder gaan we daar een aantal voorbeelden van zien.
De meeste moderne programmeertalen kennen al veel methoden waar je als programmeur gebruik van kan maken. Deze methoden worden via de zogeheten API aangeboden. API staat voor Application Programming Interface. In de API staat hoe je de aangeboden methoden op de juiste manier in jouw programma gebruikt.

De API van Processing is te vinden onder de volgende URL:

http://processing.org/reference/

In Processing heb je ook direct toegang tot de API:

Kies via het menu Help voor Reference (Handleiding).

Een voorbeeld van een methode uit de API is de rect() methode. Zoals je in de API kunt lezen tekent deze methode een rechthoek op het grafische scherm. Hiervoor *heeft deze methode echter wel meer informatie van jou nodig*: de x-positie, y-positie, de breedte en de hoogte van de rechthoek. Deze waarden geef je mee tussen de haakjes en wel in exact dezelfde volgorde als de API aangeeft:

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

Schrijf deze regel maar eens in Processing. Nadat je op de run-knop drukt zie je een klein schermpje met een witte rechthoek. Merk ook de puntkomma aan het einde van de regel op. Als je deze vergeet te typen dan krijg je een foutmelding en zal het programma niet runnen. Deze puntkomma kom je onder andere na elke methodeaanroep tegen en geeft aan dat dit het einde van het commando is. Niet vergeten dus.

[1] Behalve de term methode wordt ook vaak de term *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 per se onjuist is, maar Processing is gebaseerd op de programmeertaal Java. In deze programmeertaal is “methode” de officiële naam. Tijdens de vervolgcourse OOPD leren jullie het verschil tussen beide termen.

Teken nu eens een rechthoek op verschillende coördinaten en met diverse afmetingen en kijk wat er gebeurt.
De rechthoeken kunnen nu (deels) buiten het kleine scherm terecht komen. De grootte van dat grafische scherm kunnen we gelukkig aanpassen door als eerste de methode size() te gebruiken:

```
size(600, 400);
```

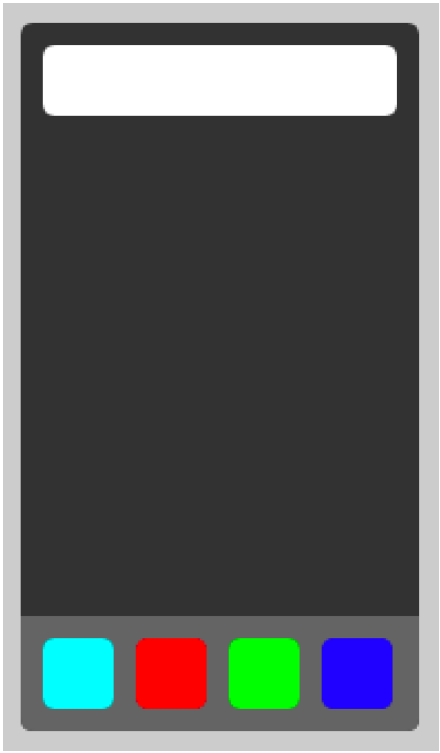
Met bovenstaande code wordt het scherm nu 600 pixels breed en 400 pixels hoog.
Het is ook mogelijk om de kleur van de achtergrond en de kleur van de vierkanten aan te passen. Probeer het maar eens (tip: onderzoek de methoden `background()` en `fill()`)in de reference van Processing?

De informatie die we steeds tussen de haakjes meegeven noemen we argumenten. Die argumenten van een methode bevatten dus steeds de informatie die een methode nodig heeft zijn werk te kunnen doen.
Je zult gaan merken dat naast de term argument ook de term parameter gebruikt wordt en dat deze regelmatig door elkaar heen worden gebruikt (zo vaak dat zelfs de Java specificatie er niet helemaal consistent in is).
Later, wanneer we onze eigen methoden gaan maken, komen we terug op de term parameter. We gaan eerst het verschil bekijken tussen programmeren en coderen.

1.2 Programmeren versus coderen

Beginnende programmeurs denken bij programmeren louter aan het geven van deze instructies. Het schrijven van instructies noemen we echter coderen, of soms gekscherend “code kloppen”. Het is echter heel belangrijk om je te realiseren dat programmeren meer is dan coderen.
Programmeren is het bedenken van stappen om een probleem op te lossen of een doel te behalen. We noemen deze stappen het “oplossingsalgoritme”. Coderen is de fase die daarop volgt: het omzetten van de algoritmen naar code die hoort bij de programmeertaal.

1.2.1 Oplossingsalgoritmen bedenken



Om een werkend programma te schrijven is het belangrijk goed na te denken voor dat je begint met het schrijven van code. Het is zonde dat je veel tijd hebt besteed aan het schrijven van code en het vervolgens niet werkt. Dat goed nadenken zit hem vooral in het bedenken van het oplossingsalgoritme. Hieronder werken we een voorbeeld uit hoe je het nadenken over het oplossingsalgoritme kan aanpakken.

Stel je krijgt de opdracht om een programma te schrijven die een besturingssysteem van een mobiele telefoon simuleert. In de figuur hiernaast staat een vereenvoudigde weergave van een startscherm met vier snelkoppelingen naar applicaties in een onderbalk en een widget bovenin het scherm. Het is nogal een uitdaging om dit in een keer in code op te schrijven. De typische beginner pakt zijn laptop en begint te typen. De professional echter denkt eerst na over hoe hij dit aan gaat pakken.

De stappen van de professional kunnen er als volgt uitzien:

1. teken achtergrond
2. teken widget
3. teken onderbalk
4. teken knoppen(snelkoppelingen)[1]
5. ALS op lichtblauwe knop gedrukt DAN {
   voer actie uit van lichtblauwe knop
   }
6. ALS op rode knop gedrukt DAN {
   voer actie uit van rode knop
   }

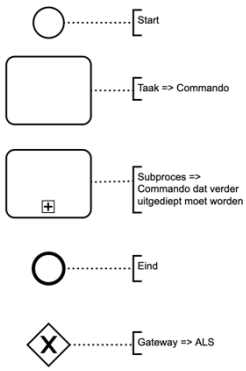
etc…

Je hebt nu een algoritme bedacht met 6 onderdelen. Nu kan de programmeur zich gaan richten op die individuele onderdelen. We bekijken de eerste vier:

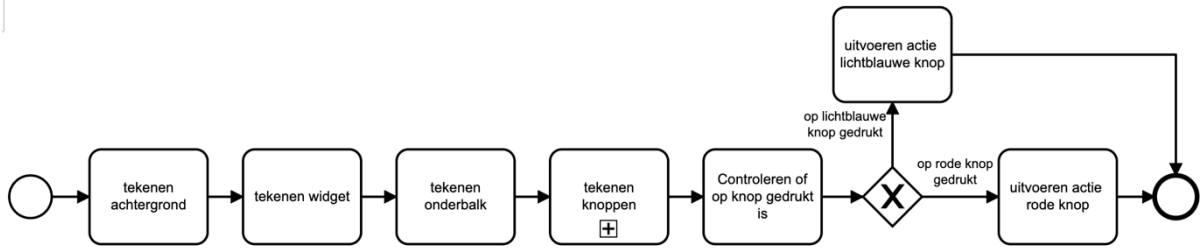
tekenAchtergrond: teken grote rechthoek (donkergrijs, afgeronde hoeken)

tekenWidget: teken een witte rechthoek met afgeronde hoeken op coördinaat (xWidget, yWidget) met breedte breedteWidget en hoogte hoogteWidget
tekenOnderbalk: teken een grijze rechthoek op coördinaat (xOnderbalk, yOnderbalk) met breedte breedteOnderbalk en hoogte hoogteOnderbalk
tekenKnoppen: VOOR alle knoppen doe: tekenKnop met bijbehorende eigenschappen (x, y, breedte, hoogte)

1.2.2 Oplossingsalgoritme in de vorm van een procesmodel



De stappen van een oplossingsalgoritme vormen samen feitelijk een proces. Oplossingsalgoritmen zijn daarom te vatten in een procesmodel. Tijdens FAT leren jullie processchema's lezen die de zogenaamde BPMN-notatie hanteren. BPMN staat voor Business Process Model and Notation en is een notatiewijze die je later in de studie nog tegenkomt. BIM studenten komen deze notatie veel tegen tijdens het modelleren en analyseren van bedrijfsprocessen. De notatie gaan we niet toetsen tijdens SPB. Het tekenen van een dergelijk processchema kan je echter wel helpen bij het analyseren van problemen en het lezen ervan bij het begrijpen van de stof. Er zijn namelijk enorm veel overeenkomsten tussen het modelleren van processen en het opstellen van oplossingsalgoritmen. Wanneer we de 6 stappen uit het algoritme in een procesmodel zouden zetten dan zou dat er als volgt uitzien:



1.2.3 Code: realisatie van de oplossingsalgoritmen

Nadat we hebben nagedacht over hoe ons programma opgebouwd gaat worden kunnen we ons oplossingsalgoritme vertalen naar code. Ter illustratie staat hieronder een voorbeeld. We gaan er nog niet vanuit dat nu al alle code uit het voorbeeld begrijpt, maar misschien kan je een eind komen met de hulp van de API van Processing. Om het voorbeeld duidelijker te maken staat er ook *commentaar* in. Het commentaar kan je herkennen aan de twee schuine strepen (//) die ervoor staan. Tijdens het uitvoeren van de code negeert het programma alle regels die beginnen met twee schuine strepen. OPMERKING: deze code is nog niet helemaal compleet. Hoe dat zit zien we in het volgende hoofdstuk.

```
// tekenAchtergrond
{
  fill(DONKERGRIJS);
  rect(xAchtergrond, yAchtergrond, bAchtergrond, hAchtergrond, radiusAchtergrond);
}

// tekenWidget
{
  fill(WIT);
  rect(xWidget, yWidget, bWidget, hWidget, radiusWidget);
}

// tekenOnderbalk
{
  fill(GRIJS);
  rect(xOnderbalk, yOnderbalk, bOnderbalk, hOnderbalk, 0, 0,
       radiusOnderbalk, radiusOnderbalk);
}

// tekenKnoppen:
// tekenKnop(1)
{
  fill(CYAAN);
  rect(xKnop1, yKnop1, bKnop1, hKnop1, radiusKnop);
}
// tekenKnop(2)
{
  fill(ROOD);
  rect(xKnop2, yKnop2, bKnop2, hKnop2, radiusKnop);
}
// tekenKnop(3)
{
  fill(GROEN);
  rect(xKnop3, yKnop3, bKnop3, hKnop3, radiusKnop);
}
// tekenKnop(4)
```

```
{
  fill(BLAUW);
  rect(xKnop4, yKnop4, bKnop4, hKnop4, radiusKnop);
}
```

Zijn de accolades (in het Engels: *curly braces*) { en } je opgevallen? Hoewel in dit voorbeeld (nog) niet echt nodig of gebruikelijk, op allerlei plekken in programma's kom je deze tegen. "{" en "}" geven het begin resp. het eind van een blok code aan. Met de accolades geef je dus aan dat deze code bij elkaar hoort.

1.3 Kwaliteitsafspraak

We maken de volgende kwaliteitsafspraak voor het vervolg van deze course:

Voor complexere problemen bedenken we eerst een oplossingsalgoritme.

De code uit 1.2.2 is allesbehalve ideaal. En wellicht begrijp je ook nog niet alle details uit het procesmodel. Dat is prima voor nu. Om dat beter te doen hebben we nog een en ander te leren. In het volgende hoofdstuk gaan we het hebben over hoe wij gegevens in het geheugen opslaan.

2 GEGEVENS OPSLAAN IN HET GEHEUGEN

Zoals je eerder kon lezen bestaat een programma in feite uit niets anders dan een set instructies waarmee we gegevens gebruiken of manipuleren. In dit hoofdstuk gaan we zien hoe we het geheugen gebruiken om gegevens in te bewaren en op te halen.

2.1 Datatypen

Een computerprogramma slaat gegevens op in het geheugen. Dat moet jij de computer echter wel zelf vertellen en doe je door variabelen aan te maken en te manipuleren. Variabelen zijn in feite opslaglocaties in het geheugen van de computer. Elke variabele heeft een naam, een datatype (kortweg: type) en een waarde:

- de naam helpt ons om de computer te vertellen over welke opslaglocatie we het hebben,
- het datatype vertelt de computer hoe hij, wat wij erin stoppen, moet opslaan en
- de waarde bevat informatie die we willen opslaan.

Computers zijn heel streng over datatypes. Als we in ons programma zeggen dat een variabele getallen bevat, zal de computer gaan klagen als je in die variabele ineens tekst willen stoppen. Er moet dus goed worden nagedacht over welk datatype een variabele krijgt. Er zijn heel veel verschillende datatypen. In deze paragraaf worden de basistypen beschreven die het meest worden gebruikt.

2.1.1 Integers

Integers zijn hele getallen. Heb je in jouw programma hele getallen nodig dan moet je variabelen aanmaken van het datatype integer. Dit aanmaken van een variabele heet declareren. Het declareren gaat als volgt:

```
int aantalGetallen;
```

De variabele `aantalGetallen` is nu aangemaakt, maar bevat nog geen waarde. *Een variabele die is gedeclareerd als een integer kan alleen gehele getallen bevatten*. De eerste keer dat een variabele een waarde krijgt noemen we initialiseren:

```
aantalGetallen = 3;
```

Declareren en initialiseren kan ook achter elkaar op een regel gebeuren:

```
int aantalGetallen = 3;
```

LET OP: In dit hele hoofdstuk laten we steeds zien hoe we het declareren en initialiseren apart kunnen doen. Dat doen we zodat je de terminologie helder krijgt. **Echter voorkom dat er situaties kunnen ontstaan waarin variabelen niet geinitialiseerd zijn, maar wel gebruikt worden**. Dit zorgt voor grote problemen (In Processing krijg je een foutmelding, maar in andere talen kunnen hele raketten neerstorten). Het beste is dus om, als het kan, het declareren en initialiseren op dezelfde regel te doen.

2.1.2 Floating point getallen

Floating point getallen zijn getallen met cijfers achter de komma en worden daarom ook wel kommagetallen genoemd. Het declareren van een kommagetal gaat als volgt:

```
float bedrag;
```

En het initialiseren van een floating point variabele:

```
bedrag = 3.25;
```

Let op dat getallen de Engelse notatie volgen. We gebruiken dus een punt in plaats van een komma voor de scheiding met de decimalen.
Hoewel we in een variabele alleen data van het bijbehorende datatype kunnen stoppen, mogen we in floating point variabelen toch een heel getal stoppen:

```
float totaal = 5;
```

Hoewel in de variabele `totaal` een geheel getal wordt gestopt zal je zien dat er toch een kommagetal in zit. De integer wordt namelijk automatisch omgezet in een kommagetal. Immers het getal 5 is gelijk aan 5.0.
Let op: Andersom kan dit niet. In een variabele van het type integer kun je geen kommagetal stoppen.

2.1.3 Strings

In de programmeerwereld noemen we een stukje tekst een string. Een declaratie van een variabele van het type String ziet er als volgt uit:

```
String naam;
```

Let op dat String tijdens de declaratie met een hoofdletter geschreven wordt. Waarom dat zo is leer je bij de vervolgcourse OOPD. Voor nu is het voldoende om dit gewoon te onthouden. Nu we de variabele

naam als String hebben gedeclareerd mogen we er alleen teksten in stoppen. Dit wordt in Processing aangegeven met dubbele quotes:

```
naam = “Gestructureerd Programmeren”;
```

2.1.4 De boolean

Een boolean bevat of de waarde *true* of de waarde *false*. Meer mogelijkheden zijn er niet. Het boolean type gebruik je daarom om bij te houden of iets waar is of niet waar. De declaratie en initialisatie ziet er als volgt uit:

```
boolean isHond = true;
```

Of als je een boolean op false wil zetten:

```
boolean isKat = false
```

2.1.5 Variabelen en datatypen gekoppeld aan Processing

De datatypen zoals genoemd in 2.1.1 - 2.1.4 zijn basic datatypen die ook in andere talen dan Processing worden gebruikt. Processing heeft echter ook een aantal eigen ingebouwde variabelen. Een aantal veelgebruikte Processing specifieke variabelen zijn: width, height, mouseX, mouseY, mousePressed. Ook zijn er Processing specifieke datatypen, zoals PImage. Bestudeer deze veelgebruikte variabelen en het PImage datatype in de API en leer over het gebruik ervan.

2.2 Reeksen

Soms (of eigenlijk best vaak) heb je meerdere variabelen nodig van hetzelfde datatype om meerdere zaken samen in te bewaren. Denk hierbij bijvoorbeeld aan een serie getallen of een rij woorden. Wanneer je honderd getallen wil bewaren zou het natuurlijk erg onhandig zijn wanneer je hiervoor honderd variabelen moet declareren: getal1, getal2, getal3, etc”. Hier hebben we gelukkig een andere oplossing voor. We kunnen deze variabelen dan in een reeks zetten. Het Engelse woord voor reeks is *‘array’*.

2.2.1 1-dimensionale array

De eenvoudigste array is de zogenaamde enkelvoudige of ééndimensionale array.

Je kunt dit zien als een rij gegevens die in een la met vakjes liggen, waarbij elk vakje voorzien is van een identificatienummer. Dit identificatienummer noemen we de *index* van de array. In onderstaande afbeelding zie je een array met tien integers.

[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]
3	2	11	87	115	1	38	95	42	65

Merk op dat de indices altijd beginnen bij 0 en oplopen met stappen van 1. In deze array hebben we dus de indices 0 t/m 9.

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

```
int[] getallen;
```

Hiermee hebben we aangegeven dat we een array van integers gaan gebruiken met de naam *getallen*. We hebben hier nog niet opgegeven hoeveel getallen we erin willen zetten. Dat doen we als volgt:

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

Hiermee hebben we de array geïnitialiseerd en hebben we aangegeven dat de array tien getallen kan bevatten van het type integer. Dit kan uiteraard ook weer op 1 regel:

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

De array ziet er nu als volgt uit:

[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]

De getallen zitten er echter nog niet in. We gebruiken de index van de array om waarden in de array te zetten, bijvoorbeeld:

```
getallen[0] = 3;
getallen[1] = 2;
```

De array ziet er nu als volgt uit:

[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]
3	2								

Je kunt een array ook in één keer declareren, initialiseren en vullen. Dit doe je als volgt:

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

Hoe haal je de gegevens weer uit de array? Ook daar heb je de eerdergenoemde index voor nodig. Wanneer je uit de array `getallen` hierboven het getal 87 wil uitlezen doe je dat met `getallen[3]` en 65 haal je op met `getallen[9]`. Uiteraard kunnen we in plaats van een getal (3 of 9) ook een integer variabele gebruiken als index. Bijvoorbeeld:

```
int positie = 2;
println(getallen[positie]); //In het console komt het getal 11 te staan.
```

LET OP: wanneer je een ongeldige index gebruikt krijg je een foutmelding te zien.

Het komt regelmatig voor dat een programmeur de lengte van een array nodig heeft. Gelukkig wordt de lengte van een array automatisch voor ons opgeslagen op het moment dat we deze array aanmaken en staat opgeslagen in de variabele *arrayNaam.length* (hier dus `getallen.length`). Willen we bijvoorbeeld het laatste element uit de array afdrukken in de console dan doen we dat als volgt:

```
println(getallen[getallen.length - 1]);
```

2.2.2 2-dimensionale array

Wanneer je naar een programma kijkt als MS Excel, dan zie je op het eerste gezicht een tabel in twee 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).

Hieronder staat een voorbeeld van een tweedimensionale array, wat erg overeenkomt met een Excel sheet. In totaal heeft deze array tien rijen van tien getallen (tien reeksen van tien getallen, in totaal dus honderd getallen).

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. In Processing declareren we dit type array als volgt:

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

Er worden nu twee paar blokhaken neergezet. Eigenlijk betekent dit dat je nu geen array met integers declareert, maar een array met daarin arrays van integers. Een tweedimensionale array is dus eigenlijk een array met arrays.

Je kunt een tweedimensionale array, net als bij een één-dimensionale, 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 twee indices nodig. De eerste index bepaalt uit welke rij je het gegeven haalt en de tweede bepaalt welk element binnen die rij het wordt. Het getal 80 staat in 4de rij op de 5de positie. Dit benader je dus middels: **`getallen[3][4].`**

LET OP: De hier aangemaakte tweedimensionale tabel is mooi vierkant. Maar zoals vermeld is een tweedimensionale array eigenlijk gewoon een array van arrays. Nergens staat voorgeschreven dat die arrays allemaal evenveel elementen moeten hebben. Het volgende kan dus ook:

```
int[][] getallen = {
  {25, 26, 11, 87, 76, 1, 55, 91},
  {22, 38, 14, 85, 78, 4, 60, 95, 49, 67},
  {21, 33, 18, 18, 74}
};
```

Alle arrays hebben een eigen .length eigenschap. Wil je het aantal arrays(rijen) opvragen binnen de array? Dan staat deze opgeslagen in getallen.length.
Wil je het aantal elementen in 1 van de 3 rijen? Dan selecteren we eerst de juiste rij met hun index, hier dus `getallen[i]`. Deze array heeft zijn eigen .length eigenschap. `getallen[1].length` bevat de lengte van de array op de 2e rij.

2.2.3 Multi-dimensionale array

De wereld houdt niet op bij twee dimensies. Wanneer we de eigenschappen van een kubus bekijken heb je met lengte, breedte en hoogte te maken. Drie dimensies dus. Hieronder staat een voorbeeld van een onregelmatige driedimensionale array.

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

Hoewel wij als mens niet meer dan drie 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.

2.3 Codeblokken en scope van variabelen

Eerder hebben we het al even gehad over accolades. Met de accolades geef je aan dat deze code bij elkaar hoort. Maar accolades geven ook de zogenaamde scope van variabelen aan. Eenvoudig gezegd bedoelen we met de scope van variabelen de regels in de code waar betreffende variabelen bestaan.

Een voorbeeld:

```
1   int getal1; // deze variabele is overal beschikbaar.
2   {
3     int getal2 = 10; // deze is alleen beschikbaar binnen de accolades
4   }
5
6   {
7     int getal2 = 6; // andere getal2 dan die op r3, getal2 op r3 bestaat
8     //hier niet meer
9     getal1 = 4;  // dit is getal 1 van r1
10    {
11      int getal2 = 10; //dit mag niet en levert een foutmelding op
12    }
13  }
```

Wat gebeurt hier allemaal? Getal1 op regel 1 wordt daar globaal gedefinieerd. Deze staat bovenaan, buiten alle andere accolades, en bestaat daarom in het hele programma. Als een variabele bestaat in het gehele programma dan noemen we dat een globale variabele. De naam getal1 die we voor deze variabel hebben gekozen kunnen we hierdoor niet nog een keer gebruiken om een variabele te declareren! (Later, wanneer we eigen methoden gaan maken, zien we dat dat iets genuanceerder ligt).

De accolades op r2 en r4 geven een nieuw blok aan, met een eigen scope. Daar declareren we een variabele getal2. Deze bestaat alleen binnen de scope waarbinnen deze gedeclareerd is. Dus tussen regel 2 en 4. Op regel 5 bestaat er dus geen getal2 meer. Daarom kunnen we in regel 7 wel een nieuwe variabele met naam getal 2 declareren. Dat is dus een andere dan die op regel 3!

Binnen het codeblok r6 - r13 wordt nog een codeblok (r10 - r12) gedefinieerd. Getal2 uit regel 7 bestaat nog in regel 11. Daarom mogen we daar niet nog een getal2 aanmaken. Tot slot zien we dat getal1 op regel 9 een waarde 4 krijgt. Deze getal1 bestaat en is de globaal gedefinieerde variabele uit regel 1.

2.4 Constanten

Als de inhoud van een variabele niet mag veranderen dan moeten we dat afdwingen en in het programma duidelijk maken dat deze variabele niet veranderd mag worden. Zo'n waarde, die nooit meer van waarde verandert, noemen we een constante. Het aangeven ervan doen we met het woord *final*:

```
final int ROOD = #FF0000;
```

Constanten declareer je over het algemeen op globaal niveau zodat ze in het gehele programma bruikbaar zijn. Ook zie je dat de naam ROOD geheel in hoofdletters staat. We spreken af dat we constanten altijd in HOOFDLETTERS weergeven en dat we er het woord final bij zetten. Zo is overal direct helder dat we te maken hebben met een constante en dat we de inhoud dus niet mogen en kunnen veranderen. Bestaat de naam van een constante uit meerdere woorden? Scheid de woorden dan met een underscore om de leesbaarheid te verhogen.

Tot slot nog een opmerking over die rare waarde van ROOD met een hashtag. Dit noemen we een hexadecimale waarde. Het hexadecimale talstelsel is anders dan 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. Het hexadecimale talstelsel zie je regelmatig terug bij kleurdefinities zoals hierboven. Hoe zo'n kleurwaarde is opgebouwd, zien we later.

2.5 Kwaliteitsafspraken

Als je goed kijkt naar de stukjes voorbeeldcode in deze reader dan zit er een bepaalde consistentie in de notatie. Voor wat betreft variabelen maken we de volgende kwaliteitsafspraken:

- De naam van een variabele geeft duidelijk weer waar de variabele betrekking op heeft. Code moet lezen als een roman. De keuze van de namen van variabelen heeft daar een flinke invloed op. Namen van variabelen zoals a, b of c zijn nietszeggend en gebruiken we dus niet!
- De naam van een variabele begint altijd met een kleine letter.
- De werkelijke betekenis kun je niet altijd in één woord vangen. Namen die uit twee of meer woorden bestaan worden volgens de lower camelCase notatie geschreven. Dat betekent dat je meerdere woorden gewoon aan elkaar schrijft alsof het één woord is. De eerste letter is een kleine letter, opvolgende woorden beginnen met een hoofdletter. Een variabele die bijhoudt of een vlieg op een plant zit zou bijvoorbeeld `zitVliegOpPlant` heten.
- Constante variabelen krijgen een naam volledig in hoofdletters en ze worden gedeclareerd als final. Meerdere woorden worden gescheiden door een underscore.
- Direct voor en na het "="-teken van de initialisatie staat een spatie.

3 ZELF METHODEN SCHRIJVEN

Als we goed naar de code uit 1.2.2 kijken dan zien we een probleem. De vier knoppen (en eigenlijk alle codeblokken) bestaan uit dezelfde twee commando's:

```
// tekenKnop(…)
fill(…);
rect(…, …, …, …, …);
```

Dit soort herhalende code moeten we altijd vermijden. Want wat als we alle knoppen willen veranderen omdat later blijkt dat dit beter of mooier kan? Of wat als we ook een logo of tekst op de knop willen weergeven? Dan moeten we op vier plekken de code aanpassen. Bovendien is het zonder commentaar niet direct duidelijk dat we hier vier knoppen aan het tekenen zijn. Dat druist tegen het principe dat code altijd goed leesbaar moet zijn. Dus dat moeten we oplossen.

Dat doen we door zelf een commando te definiëren. Vanaf nu noemen we dat een methode. Een methode is als een recept. Het recept schrijft voor wat een kok moet doen. Het enige wat jij hoeft te zeggen is 'maak volgens dit recept'.
Zo ook met methodes. In een methode staat wat de computer moet doen. Het enige dat jij hoeft te doen is een instructie geven 'voer deze methode uit'.

LET OP: Foutmelding in Processing
Wanneer je mee wilt doen met de reader en voor het eerst methoden definieert in Processing dan is de kans groot dat Processing de volgende foutmelding geeft:

It looks like you-re mixing "active" and "static" modes.

Dit is iets specifieks van Processing. Op het moment dat je namelijk zelf methoden schrijft verplicht Processing je om de aanroep te doen binnen de accolades van `void setup(){}` of `void draw(){}`.
Bekijk de Processing API en bestudeer deze methoden om te zien hoe het werkt en wat er gebeurt.

3.1 Opbouw van een methode

Wanneer je zelf een methode schrijft dan is de basisopbouw altijd hetzelfde:

```
returntype methodenaam (datatype parameter, datatype parameter, etc){
  // implementatie
}
```

returntype methodenaam(…) noemen we de methodeheader. Het codeblok tussen de accolades { en } noemen we de body van de methode. Hier staat de implementatie van de methode, de code die het werk uitvoert.

3.2 De methodeheader

De methodeheader bestaat, van links naar rechts, uit drie elementen:

1. Het datatype van de zogenaamde returnwaarde.
2. De methodenaam.
3. Haakjes met parameters

Hieronder gaan we het hebben over elk van deze elementen. We beginnen met de eenvoudigste, de methodenaam.

3.3 Methodenaam

We weten inmiddels dat wanneer we een methode aanroepen, dat we dan in feite een commando geven aan Processing. Elke methode voert een taak uit. De methodenaam is een zelfgekozen naam die zo nauwkeurig mogelijk moet aangeven wat die taak is, dus wat de methode doet. **Omdat een methode iets doet zit er meestal een werkwoord in die naam.**
Tekent de methode een knop? Dan heet de functie tekenKnop. Net als de naamgeving voor variabelen volgt de naam van een methode het lowerCamelCase format.

Belangrijk bij het schrijven van een methode is dat deze maar 1 verantwoordelijkheid krijgt. Een methode die tekenKnop() heet, tekent dus ook echt alleen maar een knop. Niets anders. Niet een tweede knop tekenen, niet bepalen waar de knop moet komen te staan, en ook niet bepalen of de gebruiker op de knop gedrukt heeft.
Dekt de naam van jouw methode de lading niet? Of lukt het je niet om een scherpe naam te verzinnen? Dan is de kans groot dat de implementatie die je voor ogen hebt meer dingen doet. Deze code hoor je in zo'n geval te splitsen en elk deel in een eigen methode te onder te brengen.

3.4 Parameters van een methode

Tussen de haakjes van de methodedefinitie staan de parameters van de methode. In feite zijn parameters bijzondere variabelen. Bijzondere variabelen die gedeclareerd worden in de methodeheader.
Via de parameters vertelt de ontwerper van de methode aan de buitenwereld (die de methode aanroept): dit is de informatie die ik nodig heb om mijn werk te kunnen doen. Oftewel in ons voorbeeld: Om een knop te tekenen heb ik de kleur, afmetingen en coördinaten nodig (en als ik er een logo op wil tekenen, ook nog het logo).
Ook moeten we van elke parameter bepalen van welk datatype deze moet zijn. Als we voor elk van de eigenschappen besluiten dat dit het beste integer kunnen zijn, dan volgt hieruit de volgende (gedeeltelijke) header:

```
… tekenKnop(int x, int y, int breedte, int hoogte, int kleur)
```

of met een array:

```
… tekenKnop(int[] eigenschappen)
```

3.4.1 Volgorde van parameters

We hebben gezien dat we bij het aanroepen van methoden moeten opletten in welke volgorde wij de informatie meegeven. De Processing methode `text()` vraagt bijvoorbeeld om 3 argumenten (de losse stukken informatie die je mee moet geven tijdens de aanroep). Het komt daarbij heel nauw in welke volgorde je die informatie meegeeft om jouw tekst op de juiste manier in het grafische scherm afgedrukt te krijgen: eerst de tekst zelf, daarna de x coördinaat en daarna pas de y coördinaat.

We moeten deze volgorde aanhouden doordat de ontwikkelaar van de methode die volgorde heeft bepaald en oplegt aan de buitenwereld. De volgorde van de parameters is de volgorde die tijdens de aanroep ook echt aangehouden moet worden.
Ben je daar dan ook helemaal vrij in? In principe wel, maar probeer een logische volgorde te kiezen. Het had heel onhandig geweest als we tijdens de aanroep van `text()` eerst de y, dan de tekst en daarna pas de x hadden moeten meegeven.
Hou het ook consequent. Het niet handig om bij de ene methode eerst x en dan y te vragen en bij een andere eerst y en dan x. In de ideale situatie weten gebruikers van methoden op een gegeven moment heel intuïtief welke informatie er meegegeven moet worden en in welke volgorde dat moet gebeuren.

3.4.2 Naamgeving van parameters

In tegenstelling tot wat we eerder zagen mogen we voor de parameters namen kiezen die ook al elders in de globale scope gedeclareerd zijn. Er kunnen in deze situatie dus prima twee variabelen met dezelfde naam bestaan (LET WEL: dit zijn dus niet dezelfde!). Als je echter een goede methodenaam kiest zal dit echter vaak niet het geval zijn. Dat heef alles te maken met context. Bekijk onderstaand voorbeeld maar eens.

```
//ergens in de code
int xKnop = 200;
int xScherm = 250;
// …
tekenKnop (xKnop, …);


//verderop
void tekenKnop(int x, …){
  // …
}
```

Bovenin wordt een variabele voor de x-coördinaat van een knop gedeclareerd en geinitialiseerd. Hier is xKnop een goede naam. De toevoeging van Knop is noodzakelijk om aan te geven welke coördinaat we bedoelen: die van de knop en niet van het scherm bijvoorbeeld. Naast de x moeten we dus ook de context opnemen in de naam.
Definieer je echter een methode tekenKnop() dan geeft deze al aan dat het over een knop gaat. Een parameter in de definitie die de x-coördinaat doorgeeft hoeft in de naam niet nog eens het woord knop te bevatten. De naam x is dan voldoende om duidelijk te maken waar dit over gaat.

Samengevat. Nadat je bepaald hebt wat de methode moet doen, moet je jezelf dus altijd de vraag stellen welke informatie de methode nodig heeft om zijn werk te kunnen doen en vraag je via de parameters om deze informatie aan de buitenwereld.
De volgorde van de parameters binnen de header bepaalt in welke volgorde de informatie meegegeven moet worden tijdens de aanroep.

Het injecteren van informatie via de parameters en het idee dat een methode maar 1 verantwoordelijkheid mag hebben zijn heel belangrijke principes die ervoor zorgen dat jouw methoden, onderhoudbaar en herbruikbaar worden. Daarnaast hebben we hiermee stappen gemaakt om jouw

code beter te kunnen testen. Om dat laatste te begrijpen moeten we nog extra kennis krijgen over returnwaarden.

3.5 Returnwaarden

Naast dat een methode informatie kan eisen, kan deze ook informatie geven aan de buitenwereld. Dit gebeurt via de returnwaarde, het resultaat van de methode.
De methodeheader geeft aan van welk type die returnwaarde is. Stel we willen een methode schrijven die het gemiddelde uitrekent van drie getallen. Deze zal er als volgt uit kunnen zien:

```
float berekenGemiddelde(float getal1, float getal2, float getal3){
  int aantalGetallen = 3;
  float totaal = getal1 + getal2 + getal3;
  return totaal / aantalGetallen;
}
```

De methode berekenGemiddelde definieert de drie paramaters getal1, getal2 en getal3. Deze zijn gedeclareerd als floating point getal (kommagetal). Als je aan iemand vraagt om het gemiddelde te berekenen, dan verwacht je een antwoord. De methode moet dus ook een antwoord geven. Dat antwoord wordt aan het hoofdprogramma gegeven via de returnwaarde. Een gemiddelde bevat meestal getallen achter de komma. Daarom is het type van de returnwaarde een floating point getal.
Kijken we vervolgens naar de implementatie in de body dan zien we dat eerst het totaal wordt uitgerekend en daarna het gemiddelde wordt uitgerekend en geretourneerd. Dit retourneren geven we aan met het sleutelwoord `return`.

Opmerking. Deze methode is helaas alleen te gebruiken om het gemiddelde van drie getallen uit te rekenen. Wil je het gemiddelde van 10 getallen uitrekenen dan kan dat niet. We hebben de kennis nu nog niet om dit helemaal goed te realiseren, maar de eerste stap kunnen we al wel maken:

```
float berekenGemiddelde(float[] getallen){
  float totaal = berekenTotaal(getallen);
  return totaal / getallen.length;
}
```

Merk hierbij op dat we heel eenvoudig twee dingen los van elkaar kunnen testen. Namelijk of het berekenen van het totaal wel goed gaat en of het gemiddelde goed wordt uitgerekend. En hoewel het hier wellicht overbodig of te veel moeite lijkt, zal je later merken dat dit van onschatbare waarde is.

3.5.1 Type void als returnwaarde

Naast de bekende datatypen zoals int, float, String en boolean, kennen we bij returnwaarden ook het void type. Void is het Engelse woord voor leegte. Een header die begint met het woord void heeft geen returnwaarde. Kijk eens naar onderstaande code voor het tekenen van knoppen.

```
// tekenKnop(1)
fill(CYAAN);
rect(xKnop1, yKnop1, bKnop1, hKnop1, radiusKnop);

// tekenKnop(2)
fill(ROOD);
rect(xKnop2, yKnop2, bKnop2, hKnop2, radiusKnop);

// tekenKnop(3)
fill(GROEN);
rect(xKnop3, yKnop3, bKnop3, hKnop3, radiusKnop);

// tekenKnop(4)
fill(BLAUW);
rect(xKnop4, yKnop4, bKnop4, hKnop4, radiusKnop);
```

We zien hier herhalende code, dus schrijven we hiervoor een methode:

```
void tekenKnop(int x, int y, int breedte, int hoogte, int radius, int kleur){
  fill(kleur);
  rect(x, y, breedte, hoogte, radius);
}
```

Deze methode tekent iets op het scherm en hoeft dus niets te berekenen of zo. Daarom hoeft deze methode niets te retourneren en gebruiken we het woordje `void` in de header op de plaats waar we de het returntype aangeven.

Wat je verder misschien ook opvalt is dat we het sleutelwoord `return` niet terugvinden in de body van de methode. Dat is geen fout! Wanneer we niets terug hoeven te geven, mogen we het sleutelwoord `return` gewoon achterwege laten. Je zult zelfs zien dat dit heel gebruikelijk is. Daarom spreken we voor deze course af het sleutelwoord achterwege te laten als de methode niets teruggeeft.

Met de hierboven gedefinieerde methode `tekenKnop()` wordt de aanroep voor vier knoppen:

```
tekenKnop(xKnop1, yKnop1, bKnop1, hKnop1, radiusKnop1, kleurKnop1);
tekenKnop(xKnop2, yKnop2, bKnop2, hKnop2, radiusKnop2, kleurKnop2);
tekenKnop(xKnop3, yKnop3, bKnop3, hKnop3, radiusKnop3, kleurKnop3);
tekenKnop(xKnop4, yKnop4, bKnop4, hKnop4, radiusKnop4, kleurKnop4);
```

Het is hier direct duidelijk dat we vier knoppen tekenen. Dit is heel anders dan wanneer we 4x fill() en rect() hadden geschreven. We zien nog wel dubbele code, maar dat kunnen we later pas wegwerken.

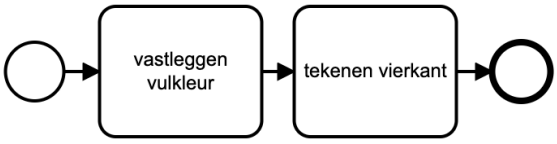
3.6 Methoden benadert via een procesmodel

Hiernaast zien we het tekenen van een knop zoals we dat in een procesmodel zouden kunnen terugzien. Doordat de details zijn weggelaten is nog goed leesbaar wat er gebeurd. Ook als dit onderdeel is van een veel groter proces.

Als we direct de details in een groter proces zouden modelleren zien we niet meer dat het gaat over het tekenen van een knop.

Je ziet verder dat het tekenen van de knop een input heeft en een output. Activiteiten hebben bepaalde gegevens nodig om uitgevoerd te kunnen worden.
In ons voorbeeld heeft de knop die getekend moet gaan worden de eigenschappen kleur, de coördinaat en de afmetingen. Om de “tekenen knop” activiteit uit te kunnen voeren zijn deze gegevens dus nodig. Daarmee vormen deze eigenschappen de input van “tekenen knop”.

Kijken we nu naar de details van “tekenen knop” (het subproces van “tekenen knop”) dan zien we dat de input van “vastleggen vulkeur” de kleur is (zoek de Processing methode `fill()` nog maar eens op). Voor “tekenen vierkant” zijn dat de x + y-coördinaat en afmetingen (zie `rect()`). Samen komt dat dus overeen met de input van “tekenen knop”.



In code vertalen we nu de activiteit “tekenen knop” naar de methode `tekenKnop`, waarbij de input gedefinieerd wordt via de parameters. De details van het subproces staan in de body. Deze methode heeft dus “vastleggen kleur” en “tekenen vierkant” in de body staan. Deze methoden bestaan al in processing, dus daar hoef je niet zelf een methode voor de definiëren.

Tot slot de returnwaarde. Het proces tekent naar het grafische scherm van Processing en geeft verder niets door naar een ander punt in het proces. Dat uit zich in code door `void` als returntype te definiëren.

Opmerking: In complexere situaties ga je zien dat er vaak subprocessen binnen subprocessen ontstaan, oftewel aanroepen van zelf gedefinieerde methoden binnen andere zelf gedefinieerde methoden.

3.7 Globale variabelen: een slechte gewoonte

We weten uit de eerste lessen dat we variabelen globaal kunnen declareren, zodat ze overal in het programma te gebruiken zijn. **Het is echter een slechte gewoonte om voor alles globale variabelen te declareren**. Deze variabelen blijven in het geheugen staan, zelfs als het programma deze tijdelijk of zelfs helemaal niet meer nodig heeft. Bovendien zorgt het voor een koppeling tussen hoofdprogramma en methoden die wel of niet in andere modulen staan (hier komen we in het hoofdstuk over software architectuur in SPAD op terug). Een goede programmeur doet zijn uiterste best om zo veel mogelijk onderdelen zelfstandig te kunnen testen en te kunnen laten functioneren.

Met de kennis die je in dit hoofdstuk hebt opgedaan kun je jouw programma structureren en leesbaar maken, maar kun je ook de hoeveelheid globale variabelen verminderen. Hierdoor blijft het overzichtelijk en onderhoudbaar, ook als jouw programma groeit.

3.8 Kwaliteitsafspraken

We weten dat we ons bij het definiëren van een eigen methode moeten afvragen wat de methode moet doen, welke informatie de methode nodig heeft om zijn werk te kunnen doen en welke informatie de methode teruggeeft.

We vullen de kwaliteitsafspraken uit voorgaande hoofdstukken nu aan met:

1. Code bevindt zich in methoden. Niet alleen voor code waarvan je nu ziet dat het herhalende code is, of waarvan je weet dat je het nog een keer nodig hebt, maak je een methode. Nee, voor elk codeblok dat een bepaalde omkaderde taak uitvoert schrijven we een methode.
2. Wanneer we zelf een methode schrijven moeten we er altijd voor zorgen dat deze methode maar één ding doet, één verantwoordelijkheid heeft. Een methode die tekenHuis() heet tekent alleen een huis. Een methode die isVingerOpKnop () heet bepaalt alleen of de vinger zich op de knop bevindt. Meer niet. Dit principe noemen we het *scheiden van verantwoordlijkheden*.
3. Informatie die een methode nodig heeft wordt gegeven (“injecteren we”) via parameters.
4. Een methode geeft informatie terug aan het hoofdprogramma via de returnwaarde. Wanneer er geen informatie wordt teruggegeven gebruiken we het sleutelwoord `void` in de header en laten we het sleutelwoord `return` in de body achterwege.
5. De naam van de methode wordt geschreven in lowerCamelCase en beschrijft zo nauwkeurig mogelijk wat de methode doet.
6. De naam van een methode bevat een werkwoord: tekenHuis(), berekenGemiddelde(), isGroot().

Voor wat betreft accolades maken we de volgende algemene afspraken:

- In het geval van accolades is het de gewoonte om de openingsaccolade achteraan de regel te plaatsen die het blok inluidt. Bij een methodedefinitie is dat aan het einde van de header.
- Vervolgens spring je één tab (standaard voegt Processing dan 2 spaties in) in voor elke programmaregel die volgt, om vervolgens
- de sluit-accolade weer een tab terug te laten springen. In het vorige fragmenten zie je dat de sluit-accolade precies onder de v van void staat.

4 OPERATOREN

4.1 Wiskundige operatoren

Met integers en kommagetallen kun je rekenen. Hiervoor heb je zogenaamde numerieke operatoren nodig. De belangrijkste wiskundige operatoren zijn optellen (+), aftrekken (-), vermenigvuldigen (*), delen (/) en modulo (%). Bekijk de volgende voorbeelden:

```
int getal1 = 25;
int getal2 = 40;
println (getal1 + getal2);  //drukt 65 af in de console.
println (getal1 - getal2);  //drukt -15 af in de console.
println (getal1 * getal2);  //drukt 1000 af in de console.
println (getal1 / getal2);  //drukt 0 af in de console.
println (getal1 % getal2);  //drukt 25 af in de console.
```

Beantwoord nu de volgende vragen:

1. Had jij de laatste twee antwoorden verwacht?
2. Waarom is getal1 / getal2 gelijk aan 0?
3. Waarom is getal1 % getal2 gelijk aan 25?

Integer deling (of geheelgetallige deling)

Bij vraag 2 hebben we te maken met een zogenaamde integer deling. Als we een integer delen door een andere integer, dan is het resultaat ook een integer. We delen een getal hier door een groter getal wat normaal een kommagetal onder de 1 oplevert (0.625). Echter omdat een integer deling een heel getal oplevert is het resultaat 0. Dit ‘probleem’ doet zich niet voor als getal1 of getal2 een kommagetal is.

Modulo

Bij vraag 3 hebben we te maken met de zogenaamde *modulo* operator. De modulo operator berekent de restwaarde. 40 gaat precies nul keer in 25. Hetgeen wat overblijft is 25. Het percentageteken heeft dus niets met procenten te maken! Later zien we dat de modulo operator in diverse scenario’s handig kan zijn.

4.2 Stringconcatenatie

We hebben gezien dat de plus operator simpelweg twee getallen bij elkaar optelt. Maar deze operator heeft ook betekening in relatie tot Strings. Met het plusteken kun je namelijk meerdere Strings aan elkaar plakken.

Het resultaat van

```
"Gestructureerd Programmeren" + "is een leuk vak"
```

is: "Gestructureerd Programmeren is een leuk vak".

4.3 Logische operatoren

Logische operatoren zijn van toepassing op booleans. Ze worden daarom soms ook booleaanse operatoren genoemd. De logische operatoren die we hier behandelen zijn de logische AND (in Processing: &&), logische OR (in Processing: ||) en NOT (in Processing: !).

!true	levert **false** als resultaat
!false	levert **true** als resultaat

true && true	levert **true** als resultaat
true && false	levert **false** als resultaat
false && true	levert **false** als resultaat
false && false	levert **false** als resultaat

true || true	levert **true** als resultaat
true || false	levert **true** als resultaat
false || true	levert **true** als resultaat
false || false	levert **false** als resultaat

4.4 Relationele operatoren

Relationele operatoren gebruik je om variabelen of combinaties van variabelen te vergelijken:

A != B	levert true als **A niet gelijk is aan B**
A < B	levert true als **A kleiner is dan B**
A <= B	levert true als **A kleiner of gelijk is aan B**
A == B	levert true als **A gelijk is aan B**
A > B	levert true als **A groter is dan B**
A >= B	levert true als **A groter of gelijk is aan B**

4.5 Operator precedence

Processing geeft de ene operator een hogere prioriteit dan de andere operator. Dit wordt operator precedence genoemd.
Hieronder staat een lijst met operatoren. Deze staan gesorteerd op aflopende prioriteit. Operatoren bovenaan de lijst zullen dus voorrang krijgen op operatoren die verder onderaan staan. Operatoren met een hoger prioriteit worden uitgevoerd voor de operatoren met een lagere.
Operatoren binnen een groep hebben een gelijke prioriteit.

1. Grouperen met haakjes : ()
2. Logische operator NOT: !
3. Delen, vermenigvuldigen, modulo: *, /, %
4. Plus en min: +, -
5. Relationele operatoren kleiner, kleiner of gelijk, groter, groter of gelijk: <, <=, >, >=
6. Relationele operatoren controle op gelijkheid: ==, !=
7. Logische AND: &&
8. Logische OR: ||
9. Toewijzing: =, +=, -+, *=, /=, %=

OPMERKING: In de lijst zijn alleen operatoren opgenomen die wij tijdens deze course behandelen.

4.6 Kwaliteitsafspraken

Er komt weer een kwaliteitsafspraak bij onze eerder gemaakte afspraken:

- Direct voor en na een operator staat altijd een spatie. Dit doen we om de leesbaarheid te verhogen.

5 HERHALINGEN

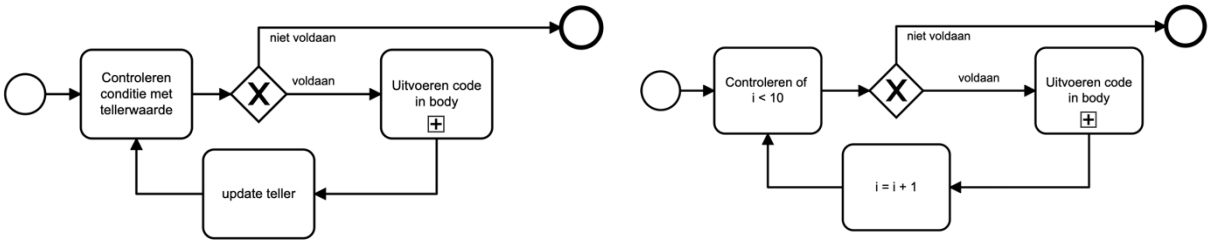
Soms wil je zaken herhaaldelijk uitvoeren. Dan wil je niet 10 keer hetzelfde commando geven. Gelukkig kan dat handiger en overzichtelijker.

5.1 De for-lus

Wanneer je iets een bepaald aantal keren uitvoert dan noemen we dat een *tellende herhaling*.

PROCESMODEL

Procesmatig ziet de tellende herhaling er als volgt uit (met rechts een voorbeeld):



0. Start.
1. Er wordt gekeken of de teller met een bepaalde beginwaarde voldoet aan de vooraf gestelde voorwaarde.
2. Als aan die voorwaarde voldaan is dan wordt de code in de body uitgevoerd, anders stopt de herhaling (als de 1e keer niet aan de voorwaarde voldaan wordt zal de code in de body dus niet worden uitgevoerd!).
3. De teller krijgt een nieuwe waarde.
4. Er wordt weer gekeken of de teller voldoet aan de vooraf gestelde voorwaarde.
5. Code uit de body wordt weer uitgevoerd wanneer de teller aan de voorwaarde voldoet.
6. De teller krijgt een nieuwe waarde.
7. Dit herhaalt zich net zolang totdat er niet meer aan de voorwaarde voldaan wordt (je moet er dus wel voor zorgen dat de teller aan de voorwaarde gaat voldoen, ander komt het programma in een oneindige lus!).
8. Einde.

Een tellende herhaling implementeren we in code met een zogenaamde for-lus.

ALGORITMISCH

Algoritmisch ziet een for-lus er als volgt uit:

VOOR (teller = beginwaarde tot of tot-en-met eindwaarde met stappen van stapgrootte) {
    doelets;
}

Bijvoorbeeld:

VOOR (teller = 1 tot en met 10 met stappen van 1) {
    doelets;
}

Tussen de accolades staan de instructies die tijdens elke iteratie van de lus uitgevoerd moeten worden. In feite telt het programma hier van 1 tot en met 10 en bij elke tel doet het programma iets. Combineer je een tellende herhaling met een array, dan ziet dat er algoritmisch bijvoorbeeld als volgt uit:

VOOR (index = 0 tot de lengte van de array met stappen van 1){
    doeletsMetArray;
}

Hier wordt elk element van de array doorlopen en bij elke stap iets gedaan met die array.

CODE

De tellende herhaling wordt dus geïmplementeerd met een *for-lus*. Deze for-lus kent de volgende syntax in Processing:

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

Dit is wel heel algemeen geformuleerd. Hieronder een voorbeeld hoe dit er uit zou kunnen zien:

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

Hier printen we de tafel van 6 in de console van Processing. *"for"* is het sleutelwoord en tussen de haakjes staan eigenlijk drie parameters, gescheiden door een puntkomma.

1. De eerste parameter, *int i = 0,* declareert een integer variabele met de naam i (i van iterator) en initialiseert deze met de waarde 0. Deze variabele is de teller van de tellende herhaling.
2. De tweede parameter, *i < 10,* beschrijft de voorwaarde waaronder de instructies binnen de lus moeten worden uitgevoerd. Hier blijft de lus zich dus herhalen zolang i kleiner is dan 10.
3. De derde parameter, *i++,* beschrijft de actie die moet worden uitgevoerd nadat de instructies in de lus zijn uitgevoerd. In dit voorbeeld wordt de teller i steeds met één opgehoogd.

In bovenstaande lus worden de instructies dus in totaal tien keer uitgevoerd.
Zoals je ziet wordt de variabele *i* ook gebruikt om binnen de lus te rekenen. Dat kan handig zijn om bijvoorbeeld de tafels af te drukken.

Afspraak:

Weet je van tevoren hoe vaak de instructies uitgevoerd moeten worden dan gebruik je de for-lus.

5.1.1 For-lus in combinatie met een 1-dimensionale array

De for-lus is uitermate geschikt in combinatie met een array te gebruiken. Gegeven een eenvoudige array met gereedschappen:

```
String[] gereedschappen = {“hamer”, “nietpistool”, “zaag”, “schroefmachine”};
```

We willen nu al het gereedschap dat in de array staat in het console afdrukken. Om dat voor elkaar te krijgen gebruiken we een for-lus:

```
for (int i = 0; i < gereedschap.length; i++){
  println(gereedschappen[i]);
}
```

We gebruiken de *.length* eigenschap van de array om het aantal iteraties vast te leggen (We gebruiken hier dus niet de harde waarde 4, want als we gereedschappen aan de array toevoegen moeten we ook deze waarde handmatig veranderen).

De vier buttons

Herinner je je de dubbele code nog bij de vier buttons?

```
tekenKnop(xKnop1, yKnop1, bKnop1, hKnop1, radiusKnop1, kleurKnop1);
tekenKnop(xKnop2, yKnop2, bKnop2, hKnop2, radiusKnop2, kleurKnop2);
tekenKnop(xKnop3, yKnop3, bKnop3, hKnop3, radiusKnop3, kleurKnop3);
tekenKnop(xKnop4, yKnop4, bKnop4, hKnop4, radiusKnop4, kleurKnop4);
```

Dat kunnen we nu oplossen door een for-lus 4 keer te laten itereren en tijdens elke iteratie een button te laten tekenen. Daarvoor nemen we voor nu aan dat y, breedte, hoogte en radius voor elke knop hetzelfde is en maken we voor de kleuren van de buttons een array:

```
int[] knopkleuren = {#00ffff, #ff0000, #00ff00, #0000ff};
```

De for-lus die 4 knoppen tekent ziet er dan als volgt uit:

```
int xKnop = xKnop1;
for (int i = 0; i < aantalKnoppen; i++) {
  tekenKnop(xKnop, yKnoppen, bKnoppen, hKnoppen, radiusKnoppen, knopkleuren[i]);
  xKnop = xKnop + grootteKnoppen + marge;
}
```

De code kan er dan in zijn geheel als volgt uitzien:

```
void setup() {
  size(800, 500);
  int[] knopkleuren = {#00ffff, #ff0000, #00ff00, #0000ff};
  int xKnop = 10, yKnoppen = 10, bKnoppen = 180, hKnoppen = bKnoppen, radiusKnoppen = 6;
  int aantalKnoppen = knopkleuren.length, marge = 10;

  for (int i = 0; i < aantalKnoppen; i++) {
    tekenKnop(xKnop, yKnoppen, bKnoppen, hKnoppen, radiusKnoppen, knopkleuren[i]);
    xKnop = xKnop + grootteKnop + marge;
  }
}

void tekenKnop(int x, int y, int breedte, int hoogte, int radius, int kleur) {
  fill(kleur);
  rect(x, y, breedte, hoogte, radius);
}
```

In dit voorbeeld hebben we er t.b.v. de eenvoud voor gekozen om alleen de kleuren in een array te bewaren en alleen de x-coördinaat te variëren. We hadden ook alle eigenschappen van de knoppen in een array kunnen stoppen. Probeer dat maar eens.

Uiteraard laten we de for-lus niet zo los in het hoofdprogramma staan. We hebben tenslotte de kwaliteitsafspraak gemaakt om voor elke blok code die een op zichzelf staand iets doet een methode te maken. Deze methode noemen we tekenKnoppen. Dat is tenslotte wat deze code doet: knoppen tekenen. Het schrijven van deze methodedefinitie laten we verder aan jou over.

5.1.2 Geneste for-lus in combinatie met een 2-dimensionale array

In de vorige paragraaf hebben we gezien hoe je een for-lus gebruikt in combinatie met een 1-dimensional array. Maar we zullen gegevens regelmatig opslaan in een array met meer dimensies. Neem onderstaande tweedimensionale array met getallen.

```
int[][] getallen = {
  {1, 3, 5, 7, 2},
  {1, 5},
  {5, 10, 5, 7}
};
```

We willen nu alle getallen achter elkaar printen in de console. Dat doen we met een geneste for-lus:

```
for (int i = 0; i < getallen.length; i++) {
  for (int j = 0; j < getallen[i].length; j++) {
    print (getallen[i][j]);
  }
}
```

Merk op dat we in de binnenste lus geen getallen.length gebruikt hebben, maar getallen[i].length. De eigenschap getallen.length heeft betrekking op hoeveel array's er in de array zitten (een tweedimensionale array is tenslotte een array met arrays). Elk van deze arrays heeft een eigen lengte en dus een eigen .length eigenschap.

Tot slot. Als je nu terugkijkt naar het processchema van de for-lus aan het begin van dit hoofdstuk dan valt je misschien op dat de body van de lus gedefinieerd is als een subproces. Oftewel een activiteit die weer uit meerdere activiteiten bestaat, maar waarvan we de details verborgen hebben ten behoeve van

leesbaarheid. We zien in de dubbele for-lus dan ook al wat meer complexiteit ontstaan. Dat maakt deze code lastiger leesbaar en ontwikkelaars maken meer fouten die lastiger op te sporen zijn wanneer ze de code op deze manier schrijven. Het beste is dan ook om een extra methode te maken. In ons voorbeeld krijg de binnenste for-lus dus een eigen methode:

```
for (int j = 0; j < getallen[i].length; j++) {
  print (getallen[i][j]);
}
```

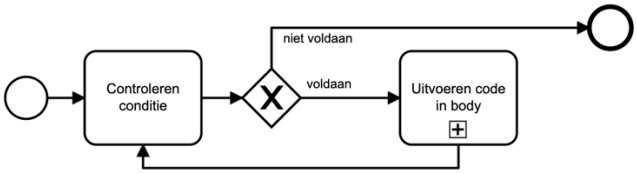
Aan jou de taak om dat eens te doen. Vraag jouw docent om hulp als je er niet uitkomt.

5.2 De while-lus

Behalve de tellende herhaling bestaan er ook herhalingen die net zolang uitgevoerd worden zolang aan een bepaalde voorwaarde wordt voldaan. Dit noemen we een *voorwaardelijke herhaling*.

PROCESMODEL

Procesmatig ziet de voorwaardelijke herhaling er als volgt uit:



0. Start.
1. Er wordt gekeken of aan een bepaalde conditie voldaan wordt.
2. Als aan die voorwaarde voldaan is dan wordt de code in de body uitgevoerd. Anders stopt de herhaling (als de 1e keer niet aan de voorwaarde voldaan wordt zal de code in de body dus niet worden uitgevoerd!).
3. Stappen 1 en 2 worden herhaald totdat er niet meer aan de voorwaarde voldaan wordt (je moet er dus wel voor zorgen dat er aan de voorwaarde voldaan gaat worden, ander komt het programma in een oneindige lus!).
4. Einde.

Een voorwaardelijke herhaling implementeren we in code met een zogenaamde while-lus.

ALGORITMISCH

ZOLANG (aan voorwaarde is voldaan) {
doelets;
}

Bijvoorbeeld:

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 of **totaal kleiner is dan 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.

CODE

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

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

Alles wat met een for-lus kan, kan ook met een while-lus. Soms is het echter wel omslachtiger. Dat komt omdat deze lus voor iets anders bedoeld is. Wanneer ik eerdergenoemde for-lus

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

wil omzetten naar een while-lus, dan 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 deze manier niet handig. Behalve dat de code langer is, zien we ook dat de variabele i nog altijd bestaat na het doorlopen van de lus. We houden dus een stukje geheugen bezet, terwijl we hier niets meer mee doen.

Zoals genoemd gebruik je een for-lus wanneer je vooraf weet hoeveel iteraties de lus moet doorlopen. De while-lus leent zich meer voor een situatie waarvan je de start of het einde niet van tevoren weet.

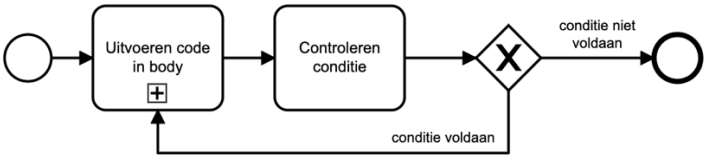
5.3 De do…while-lus

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 kunnen allerlei rare dingen invoeren waar jouw programma niets mee kan. Daarom valideer je de invoer. Wanneer deze 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 één keer zal zijn.

Ook dit is een *voorwaardelijke herhaling,* maar deze komt één of meer keer voor.

PROCESMODEL

Procesmatig ziet de deze voorwaardelijke herhaling er als volgt uit:



0. Start.
1. Code uit de body wordt uitgevoerd.
2. Er wordt gekeken of aan een bepaalde conditie voldaan wordt.
3. Als aan die voorwaarde voldaan is dan wordt de code in de body weer uitgevoerd.
4. Stappen 2 en 3 worden herhaald totdat er niet meer aan de voorwaarde voldaan wordt (je moet er dus wel voor zorgen dat er aan de voorwaarde voldaan gaat worden, ander komt het programma in een oneindige lus!).
5. Einde.

Een voorwaardelijke herhaling implementeren we in code met een zogenaamde while-lus.

ALGORITMISCH

DOE {

Code uit body uitvoeren;

} EN HERHAAL ZOLANG (aan voorwaarde is voldaan)

Voorbeeld:

DOE {

invoer = vraagInvoer();

invoerOK = valideerInvoer(invoer);

} EN HERHAAL ZOLANG (invoerOK == false)

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

CODE

In Processing wordt een voorwaardelijke herhaling die minimaal eenmaal uitgevoerd moet worden geïmplementeerd met een *do-while-lus*. Deze lus kent de volgende syntax:

do {

instructie(s);

} while ***(****voorwaarde****)***

De instructies worden eerst eenmaal uitgevoerd. Daarna wordt gecontroleerd of aan de voorwaarde wordt voldaan. Als aan die voorwaarde wordt voldaan dan worden de instructies nogmaals uitgevoerd.

Stel je wil een student simuleren die een studentenkamer zoekt. Daarvoor moet je op een advertentie reageren. Maar vaak is het niet bij de eerste keer raak. De student moet dus op minimaal 1 advertentie reageren. In code zou dat er ongeveer zo uitzien:

```
do {
	reageerOpAdvertentie();
	bekijkKamer();
	neemBeslissing();
} while (!huurovereenkomstGetekend);
```

5.4 Kwaliteitsafspraak: wanneer gebruik je welke structuur?

Welke structuur je kiest, is afhankelijk van het probleem dat je moet oplossen:

1. Wanneer je instructies een vooraf bepaald aantal keren wil laten uitvoeren, kies je de for-lus;
2. Wanneer je vooraf niet weet hoe vaak instructies uitgevoerd moeten, kies je voor een while(…)-lus;
3. Wanneer je vooraf niet weet hoe vaak instructies uitgevoerd moeten, maar wel dat dit minimaal één keer moet gebeuren, kies je voor de do-while-lus.

6 CONDITIES

6.1 Booleaanse expressies

Een booleaanse expressie (ook wel voorwaardelijke expressie) is een type expressie waarvan het uiteindelijk resultaat waar of niet waar is (het resultaat is dus een boolean). Alle expressies met relationele, booleaanse -of een combinatie van beide- operatoren zijn booleaanse expressies. Voorbeeld:

A == B && C > 0 || D == true

Als we deze expressie in drie delen opsplitsen:

- deel1: **A == B** resulteert in true of false
- deel2: **C > 0** resulteert in true of false
- deel3: **D == true** resulteert in true of false

Dan zie je dat elk deel resulteert in een true of false. Aangezien

deel1 && deel2 || deel3

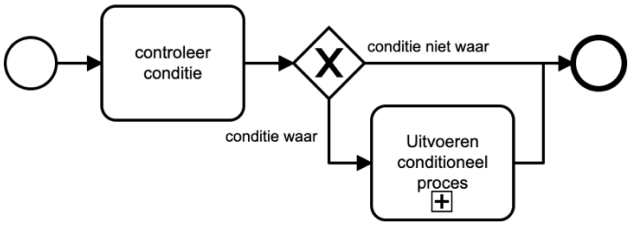
ook in een true of false resulteert hebben we hier te maken met een booleaanse expressie.

6.2 Het If statement

Sommige dingen wil je alleen laten gebeuren als er aan een bepaalde voorwaarde is voldaan.
We gebruiken booleaanse expressies om te kijken of ons programma aan bepaalde voorwaarden voldoet, zodat het programma op basis daarvan kan beslissen bepaalde acties wel of niet uit te voeren.

PROCESMODEL

Procesmatig ziet het if-statement er als volgt uit:



0. Start.
1. Er wordt gekeken of aan de gestelde conditie (booleaanse expressie) voldaan wordt.
2. Alleen als er voldaan wordt aan de conditie (booleaanse expressie resulteert in TRUE) wordt de code in de body van het if-statement uitgevoerd.
3. Einde.

ALGORITMISCH

ALS Conditie is voldaan DAN:
Voer conditionele code uit

Voorbeeld:

ALS resultaat groter of gelijk aan 5,5 DAN:
aantalStudiepunten wordt vermeerderd met 7,5

CODE

In processing kent if-statement kent de volgende syntax:

```
if (conditie) {
        instructie(s);
}
```

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

```
float resultaat, aantalStudiepunten;

// Hier moet een stukje programma komen dat het resultaat
// voor een eindtoets en het aantal tot nu toe
// behaalde studiepunten inleest.

if (resultaat >= 5.5){
      aantalStudiepunten = aantalStudiepunten + 7.5;
}
```

In dit geval wordt bij het aantalStudiepunten 7½ opgeteld wanneer het resultaat 5½ of groter is.

Let op: De voorwaarde achter het *if* sleutelwoord dient tussen haakjes te staan!

We kunnen ook een samengestelde voorwaarde gebruiken (zie ook de paragraaf over booleaanse expressies). Dat betekent dat aan twee of meer voorwaarden moet worden voldaan voor het wordt uitgevoerd. Bijvoorbeeld als je voor twee toetsen beide minimaal een 5.5 had moeten hebben om je studiepunten te verdienen. Het algoritme is dan als volgt:

ALS resultaat1 groter of gelijk aan 5.5 EN resultaat2 groter of gelijk aan 5.5 DAN:
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 en het aantal tot nu toe
//    behaalde studiepunten inleest.

if (resultaat1 >= 5.5 && resultaat2 >= 5.5){
      aantalStudiepunten = aantalStudiepunten + 7.5;
}
```

We hebben nu resultaat1 en resultaat2 als twee losse variabelen gedeclareerd. Echter er zijn situaties dat er meer toetsen zijn en dus meer toetsen te halen zijn. In dat geval zou je deze cijfers beter op kunnen slaan in een array. Bedenk alvast hoe de code er in dat geval uit zou zien. Tijdens de lessen gaan we hier nog verder op in.

Je zou ook nog kunnen zeggen dat studiepunten worden toegekend op basis van het toets resultaat *of* op basis van een vrijstelling. Dit gaat er als volgt uitzien:

ALS resultaat groter of gelijk aan 5,5 OF heeftVrijstelling is waar DAN:
aantalStudiepunten wordt vermeerderd met 7,5

De vertaling naar Processing ziet er dan als volgt uit:

```
float resultaat, aantalStudiepunten;
boolean heeftVrijstelling;

//    Hier moet een stukje programma komen dat het resultaat
//    voor en de eventuele vrijstelling en het aantal
//    tot nu toe behaalde studiepunten inleest.

if (resultaat >= 5.5 || heeftVrijstelling){
      aantalStudiepunten = aantalStudiepunten + 7.5;
}
```

Je kunt ook verder combineren. Gebruik dan haakjes zodat je zeker weet dat de conditie op de juiste manier geëvalueerd wordt:

```
if (resultaat1 >= 5.5 && (resultaat2 >= 5.5 || heeftVrijstelling))
```

Het resultaat hiervan is niet hetzelfde als het resultaat van:

```
if ((resultaat1 >= 5.5 && resultaat2 >= 5.5) || heeftVrijstelling))
```

In het eerste geval kan alleen resultaat2 door een vrijstelling worden gecompenseerd terwijl in het tweede geval de vrijstelling voor beide resultaten geldt.

Het belang van accolades

We hebben het eerder al over accolades gehad. Ook bij de if constructies zijn ze cruciaal. Als je bij een voldoende toetsresultaat niet alleen de studiepunten wil aanpassen maar ook wil aanvinken dat een student voor een course geslaagd is, moeten er twee 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 de accolades voor.

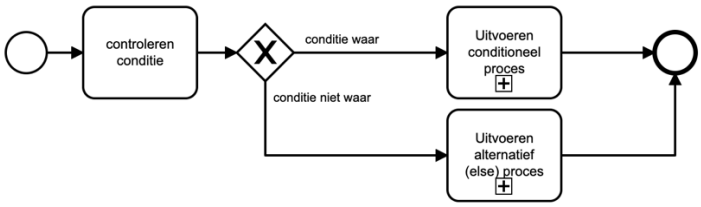
Dus:

```
float resultaat, aantalStudiepunten;
boolean heeftVrijstelling, isGeslaagd;

//    Hier moet een stukje programma komen dat het resultaat
//    voor en de eventuele evc en het aantal tot nu toe
//    behaalde studiepunten inleest

if (resultaat1 >= 5.5 || heeftVrijstelling) {
      aantalStudiepunten = aantalStudiepunten + 7.5;
      isGeslaagd = true;
}
```

Zonder de accolades is alleen de eerste opdracht (`aantalStudiepunten = aantalStudiepunten + 7.5;`) 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 aanlokkelijk perspectief is, is dit toch echt niet de bedoeling.

6.3 De andere gevallen

Vaak is de situatie ingewikkelder dan hierboven geschetst en wil je in de ene situatie bepaalde instructies uitvoeren, maar in alle andere gevallen andere code. Hieronder het processchema.



Stel dat de student bij een voldoende resultaat zijn studiepunten krijgt, maar bij een onvoldoende resultaat een punt aftrek zou krijgen. De aftrek van punten moet ook worden verwerkt. Je krijgt dan het volgende algoritme:

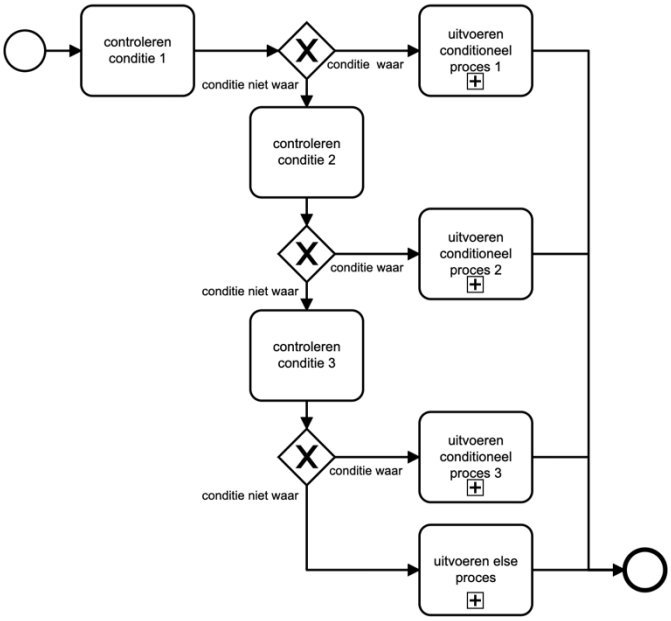
ALGORITME

ALS resultaat groter of gelijk aan 5.5, DAN:
aantalStudiepunten wordt vermeerderd met 7,5
ANDERS:
aantalStudiepunten wordt verminderd met 1

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 en het aantal tot nu toe
//    behaalde studiepunten inleest.

if (resultaat >= 5.5) {
      aantalStudiepunten = aantalStudiepunten + 7.5;
}
else {
      aantalStudiepunten = aantalStudiepunten - 1;
}
```

Tot slot zullen er situaties zijn waarbij in de ene situatie code uitgevoerd moet worden, in een andere situatie andere code, in weer een andere situatie weer andere code en in alle overige gevallen weer ander code. Het processchema ziet dat eruit zoals hiernaast.



ALGORITMISCH

Algoritmisch ziet dat er als volgt uit:

ALS conditie 1 voldaan, DAN:
Instructie(s)1
ANDERS ALS conditie 2 voldaan, DAN:
Instructie(s)2
ANDERS ALS conditie 3 voldaan, DAN:
Instructie(s)3
ANDERS:
Overige instructie(s)

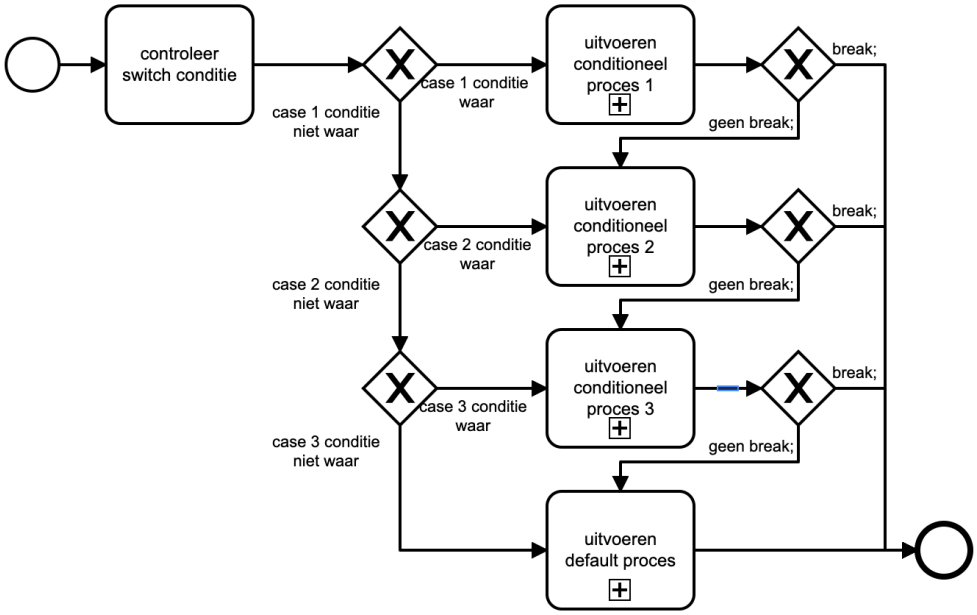
CODE

In Processing ziet het er dan zo uit:

```
boolean conditie1, conditie2;
//     Hier moet een stukje programma dat conditie1 en conditie2 ‘uitrekent’

if (conditie1 == true) {
       instructie1();
}
else if (conditie2 == true) {
       instructie2();
}
else {
       restinstructie();
}
```

6.4 De switch

Soms heb je te maken met een ingewikkelder keuze, zoals weergegeven in onderstaand processchema.



Stel we hebben bij AIM de regel dat je maximale cijfer 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 twee keer kan dit nog maximaal een 8 zijn, na drie keer een 7 en na vier 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.

ALGORITMISCH

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

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

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

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

De term *break* is een sleutelwoord dat opdracht geeft om deze *structuur* te verlaten.
De werking van de methode *constrain()* is terug te vinden in de *API* van Processing.

7 HET EVENT-MODEL VAN PROCESSING

In dit hoofdstuk gaan we in op de interactie tussen de gebruiker en het programma. Dit gebeurt aan de hand van gebeurtenissen oftewel events die binnen het grafische scherm plaatsvinden.

7.1 Events

We hebben altijd op twee plaatsen uitvoer kunnen tonen. De eerste is de message area of console. Deze is bedoeld voor de ontwikkelaar en deze uitvoer ziet een gebruiker niet.
De tweede is het grafische scherm dat de gebruiker van jouw programma wel ziet. Dit scherm noemen we ook wel een GUI (Graphical User Interface).

In een GUI kun je op willekeurig plaatsen klikken met je muis. Zo'n klik is een voorbeeld van een event. Zie een **event als een actie van de gebruiker dat zorgt voor een kort intermezzo tijdens de het uitvoeren van jouw programma**. Tijdens dit intermezzo wordt de code uitgevoerd die hoort bij het event.

Gegeven is bijvoorbeeld onderstaand programma.

```
int xCirkel = 50, yCirkel = 50, diamCirkel = 50, kleurCirkel = 0;
final int WIT = 255;
final int ROOD = #FF0000;

void draw(){
  background(WIT);
  tekenCirkel(xCirkel, yCirkel, diamCirkel, kleurCirkel);
}

void tekenCirkel(int x, int y, int diameter, int kleur) {
  fill(kleur);
  ellipse(x, y, diameter, diameter);
}
```

Het programma binnen de *draw()* blijft zich oneindig herhalen[2]. In dit programma wordt dus steeds een zwarte cirkel getekend op een witte achtergrond.

[2] Standaard wordt deze lus 60 keer per seconde uitgevoerd. Dit gaat dus aanzienlijk sneller dan de waarnemingssnelheid van het menselijk oog, dus merken wij daar niets van.

We willen nu, nadat er op een willekeurige plaats binnen de GUI is geklikt, dat de cirkel rood wordt. De variabele waarin de kleur wordt bijgehouden heet kleurCirkel. Als er een muisklik event plaatsvindt moet deze variabele dus een nieuwe waarde krijgen (ROOD);

We moeten er dus voor zorgen dat het muisklik event wordt gedetecteerd. Gelukkig helpt Processing je hierbij. In de API staat een aantal events beschreven, waaronder mouseClicked(), mouseMoved(), mouseReleased(), mousePressed(), keyPressed() etc. Bestudeer de API op Processing.org en zorg dat je de verschillen begrijpt.

In ons voorbeeld hebben we mousePressed() nodig om de kleur te veranderen op het moment dat er op de muis wordt geklikt. We voegen daarom de volgende code toe aan ons programma:

```
void mousePressed(){
  kleurCirkel = ROOD;
}
```

Processing controleert telkens, vlak voordat de draw lus wordt uitgevoerd, op het plaatsvinden van een event. Als op dat moment met de muis geklikt is zal de variabele kleurCirkel zijn nieuwe waarde krijgen en zal tekenCirkel() in de draw() vanaf dat moment een rode cirkel tekenen.

Opmerking: Helaas moet je de variabele `kleurCirkel` globaal declareren, er staan geen parameters in de header. Processing ziet een event header met parameters als een gewone methodedefinitie!

7.2 Kwaliteitsafspraak: Scheiding van state en view

De inhoud van het geheugen op een bepaald moment (regel in het programma) noemen we de toestand (*state*) van het programma op dat moment. De manier waarop de informatie uit het geheugen getoond wordt aan de gebruiker noemen we de view.

Wat je in voorgaand voorbeeld misschien is opgevallen is dat we het tekenen in de draw() deden en het veranderen van de waarde van `kleurCirkel` in het event deden. Hier zien we de laatste kwaliteitsafspraak die we voor dit deel van de course Structured Programming maken:

We zorgen voor een duidelijk scheiding tussen

a. *de code aangaande de toestand van het programma (en het aanpassen daarvan) en*
b. *de code aangaande de presentatie van de gegevens aan de gebruiker.*

Middels scheiding van toestand (manipulatie) en view zorgen we voor code die beter te begrijpen is en beter onderhoudbaar.

OPEN UP
NEW HAN_UNIVERSITY OF APPLIED SCIENCES
HORIZONS.