Opdracht 2: Constructies

– Objectgeoriënteerd Programmeren in Greenfoot –

Renske Smetsers-Weeda & Sjaak Smetsers

april, 2015

1 Inleiding

In de vorige opgave maakte je kennis met Greenfoot. Nu je wat basiskennis hebt opgedaan en in staat bent om code te lezen, kun je beginnen met het schrijven van jouw eigen code met behulp van een aantal *taalconstructies*.

2 Leerdoelen

Na het voltooien van deze opdracht kun je:

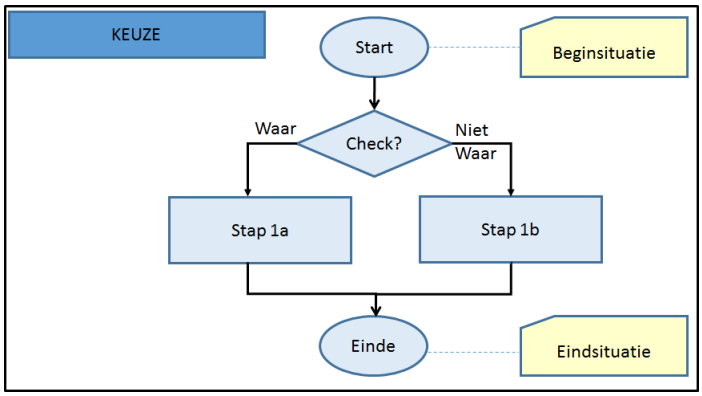
- een probleem opsplitsen in deelproblemen;
- uitleggen wat de rol van submethoden is bij het oplossen van een probleem;
- de onderdelen van een stroomdiagram benoemen;
- de regels benoemen waaraan een stroomdiagram moet voldoen;
- de stappen benoemen voor het tekenen van een stroomdiagram;
- benoemen aan welke kwaliteitscriteria een stroomdiagram moet voldoen;
- optimalisatiecriteria voor een stroomdiagram benoemen;
- in je eigen worden aangeven wat modularisatie en abstractie betekenen;
- begin- en eindsituatie voor een probleem omschrijven;
- uitleggen hoe een stroomdiagram ingezet kan worden om vroegtijdig fouten en verbeteringen in een algoritme te ontdekken;
- beoordelen of de stappen van een algoritme en het bijbehorende stroomdiagram tot de gevraagde oplossing zullen leiden;
- redeneren over de correctheid van een methode in termen van begin- en eindsituatie;
- een return statement herkennen in een stroomdiagram en in de programmacode;
- uitleggen waarom begin- en eindsituaties van een accessormethode gelijk zijn.
- de relatie tussen een stroomdiagram en programmacode beschrijven;
- algoritmes ontwerpen met gebruik van stroomdiagrammen;
- benodigde beslissingen of herhalingen voor een oplossing identificeren;
- de conditie voor een beslissing of herhaling identificeren;
- een conditie opgebouwd uit EN, NIET en `boolean` methodes toepassen;
- een algoritme met een opeenvolging, beslissing of een herhaling weergeven in een stroomdiagram;

- een stroomdiagram omzetten naar code;
- eigen programmacode opstellen, compileren, uitvoeren en testen;
- gestructureerd en stapsgewijs codeaanpassingen doorvoeren en testen;
- stappen benoemen voor het stapsgewijs analyseren en opsporen van fouten in de code (debuggen);
- in eigen woorden omschrijven wat een generiek algoritme is;
- een generiek algoritme opstellen en implementeren.

3 Theorie

Stroomdiagram

Een *algoritme* kan je overzichtelijk weergeven in een *stroomdiagram*. Dit kun je daarna omzetten in programmacode.



Figuur 1: Stroomdiagram

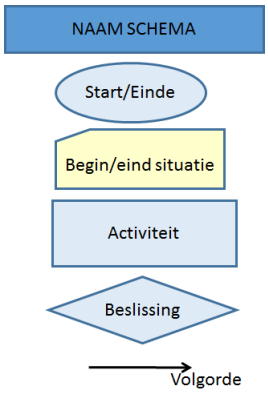
Probleem opsplitsen

Soms is een probleem erg ingewikkeld en moet je veel stappen zetten om het op te lossen. Dan kan je wel eens niet goed weten waar je moet beginnen, of te veel op details gaan letten waardoor het probleem nog groter lijkt.

Een stroomdiagram helpt je om zo'n groot probleem op te delen in kleinere (deel)problemen. Je maakt dan eerst in grote lijnen een stappenplan van wat er allemaal moet gebeuren en in welke volgorde. Dat zet je in een stroomdiagram. Elk afzonderlijk (deel)probleem splits je weer op in kleinere (deel)problemen. Je kan de deelproblemen dan één voor één aanpakken, zonder je meteen zorgen te maken over het grotere geheel. Dat is wel zo overzichtelijk. Als je een deelprobleem hebt opgelost en jouw oplossing daarvoor getest hebt, dan hoef je je niet meer druk te maken om de details daarvan. Je kunt de oplossing als bouwsteen gebruiken voor de oplossing van een volgend probleem. Deze aanpak heet *verdeel-en-heers*. Aan het einde controleer je natuurlijk nog wel of je het probleem als geheel hebt opgelost.

Hoe ziet een stroomdiagram eruit?

Een stroomdiagram bestaat uit de volgende onderdelen:



Figuur 2: Onderdelen van het stroomdiagram

Stroomdiagram regels
Voor een stroomdiagram geldt een aantal regels.

- Elk stroomdiagram heeft een naam.
- Er zweven geen onderdelen los rond (behalve de naam van de diagram), ze zijn allemaal met pijltjes of lijntjes verbonden die de volgorde aangeven.
- Er is één start en één einde, elk met een situatie omschreven:
  - Alléén in uitzonderlijke situaties mogen er meerdere eindpunten zijn.
  - Bij een activiteit die een resultaat oplevert (dus een antwoord op een vraag) moeten begin- en eindsituatie aan elkaar gelijk zijn.
- Een activiteit (rechthoek) heeft altijd één ingaand pijltje en één uitgaand pijltje.
- Het diagram bestaat uit maximaal zeven activiteiten.

Het tekenen van een stroomdiagram
Voor het tekenen van een stroomdiagram om een probleem op te lossen, volg je de volgende stappen:

1. **Beginsituatie**: Omschrijf in een paar woorden wat het probleem is dat opgelost moet worden.
2. **Eindsituatie**: Wanneer is jouw probleem opgelost? Hoe weet je dat? Omschrijf dit in een paar woorden.
3. **Oplossingsstrategie**: Bedenk hoe je het probleem wilt oplossen.
4. **Splits** het probleem op in subproblemen en die eventueel weer in subsubproblemen. Doe dit tot elk afzonderlijk probleem klein genoeg is om gemakkelijk te worden opgelost. Elk deelprobleem bestaat uit hooguit zeven stappen/deelproblemen.
5. Voor elke stap of deelprobleem:
   - Kies een geschikte naam (betekenisvol, bestaand uit werkwoorden en geformuleerd als commando).

- Omschrijf kort wat deze stap inhoudt.
- Teken elke stap als een rechthoek en verbind deze met het vorige activiteit m.b.v. een pijltje.
- *Modularisatie*: Ga na of deze stap in meer detail uitgewerkt moet worden en dus zelf weer een stap of deelprobleem is.

6. Teken het **stroomdiagram**.
7. **Check**:
   - **Regels**: Kijk of het stroomdiagram voldoet aan de 'Stroomdiagramregels' (zie hierboven).
   - **Kwaliteit**: Kijk of het stroomdiagram eenvoudiger of mooier kan (zie de kwaliteitscriteria hieronder). Komt, bijvoorbeeld, een aantal achtereenvolgende stappen vaker voor in de diagram? Geef die stappen een naam en beschrijf dat als een deelprobleem of submethode.

Kwaliteitscriteria
Met een stroomdiagram kun je nagaan of jouw oplossing goed is (aan de kwaliteitscriteria voldoet) of slimmer kan. Dit doe je vóórdat je tijd gaat steken in het implementeren (het uitwerken van je algoritme in programmacode). Achteraf kost het je namelijk veel meer tijd en moeite. Ook wordt de kans dat je fouten maakt dan groter.
Met een stroomdiagram ga je na of:

- het algoritme *correct* is: het lost het probleem op;
- begin- en eindsituaties (juist) beschreven zijn;
- de stappen elkaar logisch opvolgen;
- het gedetailleerd genoeg is (eenduidig te interpreteren);
- de juiste keuzes (*condities*) gesteld worden en daarop de juiste beslissingen genomen worden;
- er uitzonderingen zijn;
- er geen oneindige herhalingen in zitten (waardoor het programma nooit stopt);
- of er stappen opgesplitst kunnen worden om vervolgens (in een apart stroomdiagram) in meer detail beschreven te worden (*modularisatie*);
- het overzichtelijk is, of juist meer gebruik gemaakt moet worden van (*abstractie*). Voor bepaalde stappen maak je dan een apart stroomdiagram (*submethode*). Dit doe je bij een stroomdiagram met:
  - meer dan zeven stappen;
  - een reeks zeer gedetailleerde stappen (in het kader van *modularisatie*);
  - herhaling van stappen (in het kader van *hergebruik*);
- het geoptimaliseerd kan worden (eleganter of slimmer):
  - hetzelfde in minder stappen kan;

- het eenvoudiger of overzichtelijker kan;
- onderdelen onnodig zijn of nooit uitgevoerd worden;
- er een complexiteitsverbetering te maken is.

Hoe later je verbeteringen en fouten oplost, hoe meer moeite en tijd het je kost. Je wilt er dus eerst zeker van zijn dat jouw algoritme correct, efficiënt, betrouwbaar en flexibel is (oftewel 'elegant'). Daarna zet je dit pas om in code.

4 Aan de slag met de opgaven

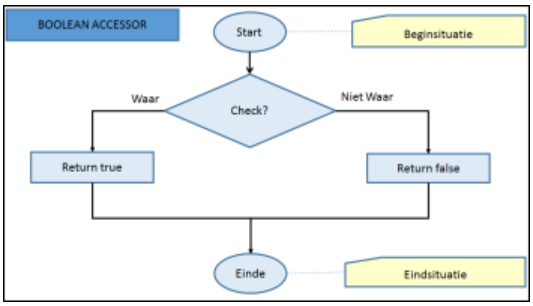
In de volgende opgaven stel je jouw eigen algoritmes op. Deze zet je op een gestructureerde manier om in een stroomdiagram en vervolgens in werkende programmacode.

Het schrijven van programmacode

1. Bedenk een plan.
2. Teken een **stroomdiagram**. Zie het stappenplan in hoofdstuk 3.
3. Zet het stroomschema om in **programmacode**. Let hierbij ook op de naamgevingsafspraken.
4. Voeg **commentaar** toe.
5. **Compileer** en 'Run'. Check of het programma doet wat je verwacht.
6. **Test** de methode door deze met de rechtermuisknop aan te roepen. Test verschillende situaties. Test ook met 'Act'.
7. **Debug**. Herstel fouten. Probeer de plek waar de fout optreedt te lokaliseren. Controleer of je programmacode op die plek in overeenstemming is met je stroomdiagram. Zo nee, pas je code aan. Zo ja, analyseer je stroomdiagram nauwkeurig om te bepalen waar je foutieve aannames hebt gemaakt.
8. **Reflecteer op en evalueer de oplossing**. Is het probleem nu opgelost? Bij het komen tot de oplossing, wat ging goed? Wat kan beter?

Accessormethode
Een accessormethode levert informatie over de toestand van een object op, bijvoorbeeld als een **`int`**, **`boolean`** of `String`.

Stroomdiagram: Het stroomdiagram van een **`boolean`** accessor methode ziet er als volgt uit:



Figuur 3: Stroomdiagram van een **`boolean`** accessor methode

Toelichting stroomdiagram:

- Eerst wordt er gecontroleerd of de conditie in de ruit waar is.
- Als de conditie 'Waar' is, wordt de pijl 'Waar' naar links vervolgd en wordt de waarde **`true`** opgeleverd.
- Als de conditie 'Niet Waar' wordt de pijl 'Niet Waar' naar rechts vervolgd en wordt de waarde **`false`** opgeleverd.
- Na het opleveren van een waarde, is de methode afgelopen. Een 'return' wordt dus altijd direct opgevolgd door een 'Einde'.
- Bij een accessormethode zijn de begin- en eindsituatie aan elkaar gelijk.

Code:
In programmacode ziet dat er zo uit:

```
boolean methodeNaam( ) {    // een boolean accessormethode
      if ( check ( ) ) {    // check de conditie in de ruit
                            // als de conditie 'Waar' is
            return true;    // lever true op

      } else {              // als de conditie niet waar is
            return false;   // lever false op
      }
}
```

Toelichting code:

- Eerst wordt er gecontroleerd of de conditie `check ( )` waar is.
- Als deze **`true`** is, wordt **`true`** opgeleverd. Daarna is de methode afgelopen.
- Als de conditie **`false`** is, dan spring je naar het gedeelte waar **`else`** voor staat. Hier wordt **`false`** opgeleverd, waarna de methode afgelopen is.
- Na het retourneren gebeurt er niks meer. Probeer je dat wel, dan krijg je de foutmelding: "unreachable statement";

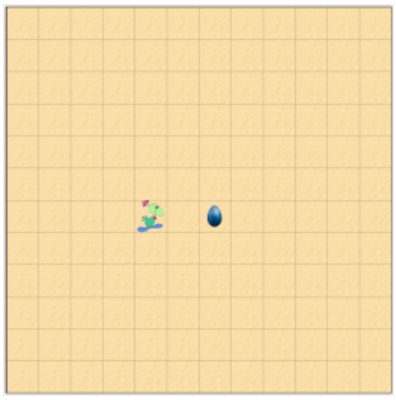
Toevoeging: Een accessormethode levert informatie op over een object. Deze hoort niets te veranderen aan de toestand. We spreken daarom af dat de begin- en eindsituaties van een accessormethode gelijk zijn aan elkaar.

4.1 Opgaven: vind het ei

Voor de volgende opgaven gebruik je het scenario **'Madagaskar 2'**.

4.1.1 Opeenvolging van instructies

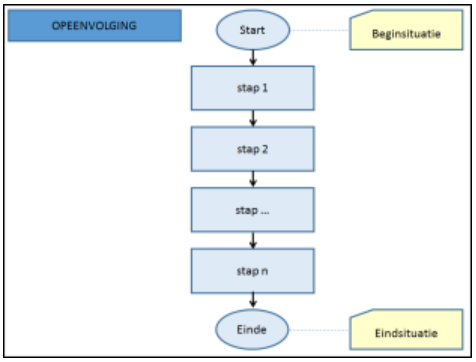
Bekijk figuur 4. Onze MyDodo, Mimi, is haar ei kwijt. Help je haar dat terug te vinden?



Figuur 4: Eerste scenario

Opeenvolging
Bij een *opeenvolging* worden de aangegeven stappen achter elkaar uitgevoerd.

Stroomdiagram: Het stroomdiagram van een opeenvolging ziet er als volgt uit:

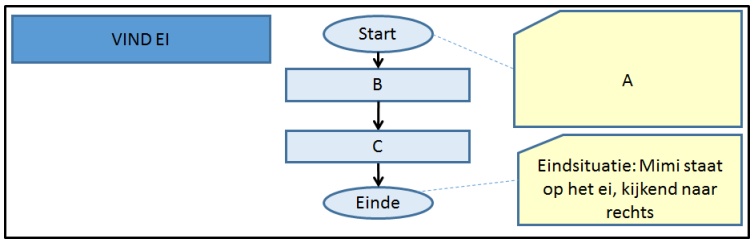


Figuur 5: Stroomdiagram opeenvolging

Code:
En in programmacode ziet dat er zo uit:

```
void methodeNaam ( ) {    // methode met opeenvolging
      stap1  ( );         // roep de methode aan in de eerste rechthoek
      stap2  ( );         // roep de methode aan in de tweede rechthoek
      stap...( );         // roep de methode aan in de volgende rechthoek
      stapN  ( );         // roep de methode aan in de n-de rechthoek
}
```

1. Open het scenario **'Madagaskar 2'**.
2. Klik met je rechtermuisknop op Mimi. Bekijk wat ze allemaal kan.
3. Wat moet Mimi doen om bij haar ei te komen?
   (a) Bedenk een strategie voor Mimi.
   (b) Schrijf op welke methodes je daarvoor gaat aanroepen.
   (c) Beschrijf de begin- en eindsituaties.
4. Zo'n strategie kun je tekenen in een stroomdiagram. Zie figuur 6. Wat moet er bij A, B en C in het stroomdiagram staan? Vul dit aan.



Figuur 6: Stroomdiagram eerste scenario

5. Pas de code aan in de `act( )` methode van MyDodo zodat Mimi doet wat in jouw stroomdiagram staat.
   Tip: Weet je niet meer wat je moet doen om code aan te passen? Volg dan de volgende stappen:
   - Klik met je rechtermuisknop op MyDodo in de klassendiagram (aan de rechterkant van het scherm).
   - Kies dan 'Open editor'.
   - Zoek de methode `act( )` op.
   - Tussen de accolades { en } komt je code te staan.
   - De twee rechthoeken in het stroomdiagram bevatten de twee regels die je moet toevoegen. Tik de tekst uit de twee rechthoeken over, wat er dan ongeveer zo uit ziet:

```
public void act ( ) {
    ROEP DE METHODE AAN IN DE EERSTE RECHTHOEK VAN STROOMDIAGRAM
    ROEP DE METHODE AAN IN DE TWEEDE RECHTHOEK VAN STROOMDIAGRAM
}
```

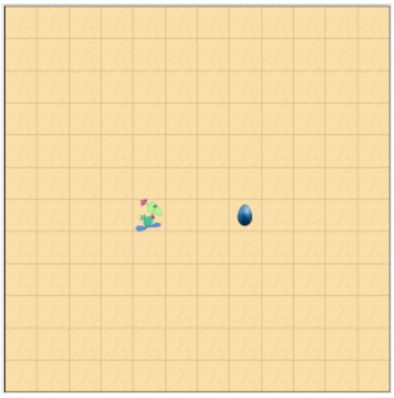
   - Pas de twee regels aan zodat het echte code wordt. Bijvoorbeeld: een aanroep van 'move' schrijf je op als `move ();`
6. Compileer (met knop 'Compile') en herstel eventuele fouten.
7. Klik op de 'Act' knop onderin het scherm.
8. Test of jouw programma doet wat je verwacht. Wordt de eindsituatie in jouw stroomdiagram bereikt? Doet de programma niet wat je verwacht? Kijk hieronder voor tips over het debuggen van de code.

Debuggen
Test na elke kleine wijziging of jouw programma doet wat je verwacht. Doet het niet wat je verwacht? Dan moet je de gemaakte stappen in de omgekeerde volgorde nalopen:

1. Controleer of jouw code overeenkomt met jouw stroomdiagram.
2. Controleer of jouw stroomschema overeenkomt met de stappen (of methode aanroepen) die je bedacht hebt.
3. Controleer of de stappen die je bedacht hebt wel kloppen en tot een oplossing van het probleem leiden.

Maak er een gewoonte van om dit meteen na élke aanpassing te doen. Zo spoor je een fout sneller op.

4.1.2 Meer opeenvolging van instructies



Figuur 7: Scenario

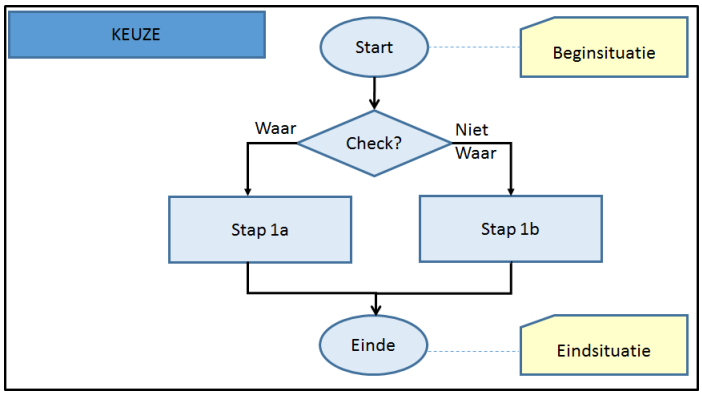
1. We gaan verder met het vorige scenario. Pas de wereld aan zodat die eruit ziet zoals in diagram 7
2. Klik met je rechtermuisknop op Mimi.
3. Welke methode(s) van MyDodo ga je aanroepen om bij het ei te komen?
4. Teken het bijbehorende stroomschema.
5. Voeg de code toe aan de `act( )` methode van MyDodo.
6. Pas ook het commentaar aan.
7. Compileer en test het programma.

4.2 De `if .. then .. else`

4.2.1 Niet door het hek lopen

Keuzes
Als de conditie waar is, dan moet je iets doen. Anders moet je iets anders doen.
Je kunt hiervoor gebruik maken van een **`if`** .. `then` .. **`else`** statement.

Stroomdiagram: Het stroomdiagram van een keuze ziet er als volgt uit:



Figuur 8: Stroomdiagram met een keuze **`if`** .. `then` .. **`else`**

Toelichting stroomdiagram:

- Eerst wordt er bij 'Check?' gecontroleerd of de conditie in de ruit waar is.
- Als de conditie 'waar' is, wordt de pijl 'Waar' naar links vervolgd en wordt de 'stap1a' uitgevoerd.
- Als de conditie 'niet waar' wordt de pijl 'Niet waar' naar rechts vervolgd en wordt de 'stap1b' uitgevoerd.
- Daarna is de methode afgelopen.

Code:
In de code ziet dat er zo uit:

```
void methodeNaam( ) {     // methode met keuze
      if ( check( ) ) {   // check de conditie in de ruit
                          // als de conditie waar is
            stap1a ( );   // roep de methode aan in de rechthoek na 'Waar'

      } else {            // als de conditie niet waar is
            stap1b ( );   // roep de methode aan in de rechthoek na 'Niet waar'
      }
}
```

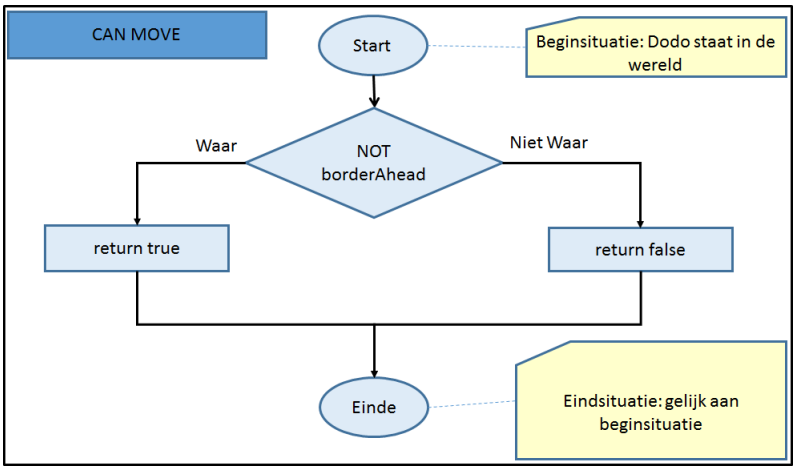
Toelichting code:

- Eerst wordt er met `check( )` gecontroleerd of de conditie **`true`** is.
- Als de conditie **`true`** is (dus `check( )` == **`true`**), dan wordt de code tussen de accolades { en } uitgevoerd (`stap1a( )`). Daarna is de methode afgelopen.
- Als de conditie **`false`** is (dus `check( )` == **`false`**), dan wordt er gesprongen naar de **`else`**. De code na de **`else`** tussen de accolades { en } wordt uitgevoerd (dus `stap1b( )`). Daarna is de methode afgelopen.

Toevoeging: Als er niets te doen valt in het geval de conditie onwaar is (en je dus in de **`else`**-tak terecht zou komen) dan mag je de **`else`**-tak gewoon weglaten.

In opdracht 1 hadden we gezien dat Mimi niet uit de wereld kan stappen. We bekijken nu opnieuw de methode **`boolean`** `canMove( )`.

1. Bekijk het stroomdiagram hieronder.



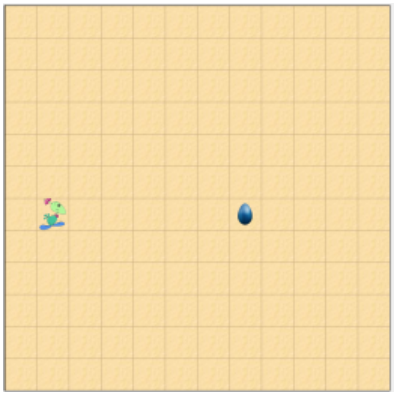
Figuur 9: Stroomdiagram `canMove( )` in MyDodo

2. Leg uit waarom de 'Eindsituatie' goed is.
3. We hebben gezien dat Mimi niet voorbij de wereldgrens kan. Maar Mimi trekt zich niks aan van hekjes. Ze loopt er gewoon doorheen! Zet een stuk hek in de wereld neer en kijk of Mimi er doorheen kan lopen of niet.
4. Gebruik de `Dodo` methode **`boolean`** `fenceAhead( )`. Wat doet deze?
5. We willen niet dat Mimi door een hek kan lopen (want dan heeft een omheining niet echt zin). We gaan daarvoor de methode **`boolean`** `canMove( )` aanpassen.

   Vul de volgende zin aan: Mimi kan bewegen als ze NIET voor een omheining staat EN .....
6. Pas de conditie in het stroomdiagram (de ruit) hierop aan.
7. Pas nu ook de conditie in de code aan. Tip: 'EN' schrijf je in code als '&&'.
8. Pas ook het commentaar aan.
9. Compileer en test het programma. Werkt het programma niet correct? Volg de stappen zoals beschreven bij hoofdstuk 'Debuggen' van 4.1.1.

4.3 De `while` loop

De opdracht luidt nog altijd: "Help Mimi haar eitje vinden". Bekijk de volgende wereld. Hoe zou je dit aanpakken?



Figuur 10: Scenario

Je kunt het natuurlijk net zo aanpakken als bij de vorige opgaven, door telkens een bepaald aantal keren `move( )` aan te roepen. Maar wat als Mimi nou 1003 stappen zou moet zetten om bij haar eitje te komen? Dan ben je als programmeur op die manier wel even bezig door 1003 keer `move( );` in te tikken. Dat heeft een paar nadelen:

- Je moet veel typen of knippen-en-plakken (en dat is saai).
- Je zou zomaar per ongeluk 1004 aanroepen van `move( )` kunnen hebben i.p.v. 1003. Jouw programma werkt dan niet meer goed.
- Jouw programma is niet flexibel of algemeen. Het werkt alleen voor één specifieke situatie. Het zal niet werken als Mimi in een volgend scenario maar 42 stapjes hoeft te zetten om bij haar eitje te komen.

Om jouw programma algemener (*generiek*) te maken moet je het algoritme wat slimmer uitwerken. Wat je eigenlijk wilt is een herhaling:

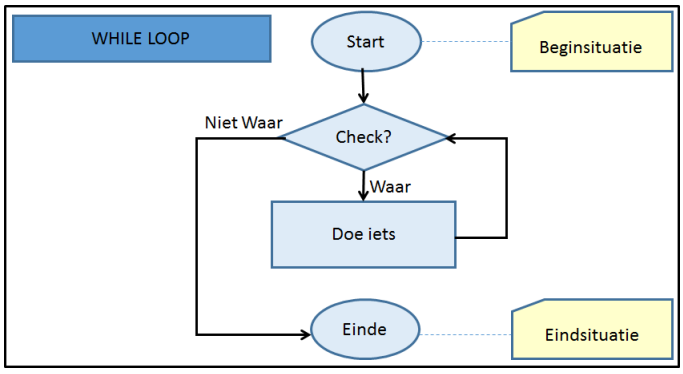
"Zolang Mimi haar ei nog niet heeft gevonden, moet ze een stapje zetten."

En dus, als ze haar ei heeft gevonden is ze klaar.

Generiek algoritme
Een algemeen algoritme dat in meerdere beginsituaties te gebruiken is noemen we *generiek*. Deze lost niet één bepaald probleem op, maar kan gebruikt worden om heel veel vergelijkbare problemen op te lossen.

Herhaling
Zolang een conditie waar is, dan moet je iets doen.
Je kunt hiervoor gebruik maken van een herhaling, ook wel **`while`** statement genoemd.

Stroomdiagram: Het stroomdiagram van een herhaling ziet er als volgt uit:



Figuur 11: Stroomdiagram voor herhaling

Toelichting stroomdiagram:

- Eerst wordt er bij 'Check?' gecontroleerd of de conditie in de ruit waar is.
- Als de conditie 'niet waar' is, dan is de methode afgelopen.
- Als de conditie 'waar' is, wordt de stap in de rechthoek uitgevoerd.
  Daarna wordt er teruggegaan naar de ruit. Is de conditie 'Check?' nog steeds 'waar'? Dan wordt het pad van 'Waar' weer vervolgd (dit heet een loop). Anders is de methode afgelopen.

Code:
In de code ziet dat er zo uit:

```
void methodeNaam( ) {          // herhaling
      while ( check( ) ) {     // check de conditie in de ruit
                               // als de conditie waar is
            doeIets( );        // roep de methode aan in de rechthoek

      }
}
```

Toelichting code:

- Eerst wordt er met `check( )` gecontroleerd of de conditie **`true`** is.
- Als de conditie **`false`** is (dus als `check( ) == false`), dan is de methode afgelopen.
- Als de conditie **`true`** is (dus als `check( ) == true`), wordt de code tussen de accolades { en } uitgevoerd. In dit geval `doIets( )`.
  Daarna wordt er teruggegaan naar de controle van de conditie `check( )`. Als de conditie nog steeds **`true`** is, dan wordt de code tussen de accolades weer uitgevoerd (de loop). Anders is de methode afgelopen

Toevoeging:

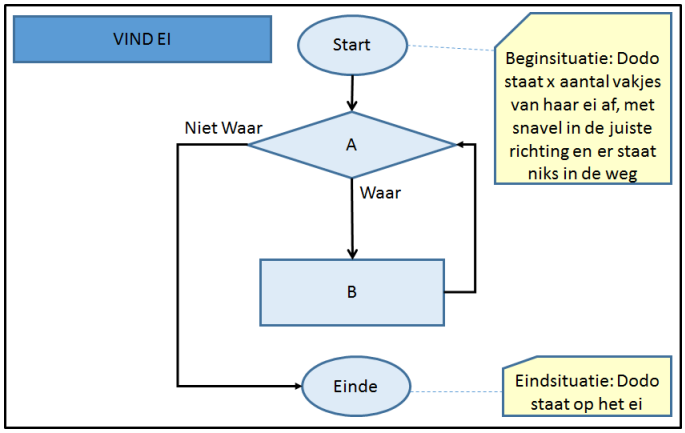
- Vaak wordt in condities gebruik gemaakt van de ontkenning of negatie (in code: '!') en uitgesproken als NIET. Bijvoorbeeld, "NIET ei gevonden". Dit komt overeen met de manier waarop je het algoritme in woorden omschrijft: "zolang iets NIET het geval is, dan... ".
- In het gedeelte `doIets( )` moet ooit iets gebeuren worden waardoor de conditie op een gegeven moment 'niet waar' wordt. Op dat moment stopt de herhaling. Een veel gemaakte fout bij het gebruik van een **`while`** is dat je nooit uit de loop komt omdat de conditie altijd 'waar' blijft. Je hebt dan een oneindige herhaling opgeschreven.

Help Mimi haar ei vinden. Jouw oplossing moet generiek zijn. Gegeven is de volgende begin-situatie:

- Mimi staat een aantal (0 of meer) hokjes van haar ei vandaan;
- Mimi kijkt in de juiste richting (ze hoeft niet meer te draaien, alléén stappen te zetten);
- Er staat niets in de weg tussen Mimi en haar ei (bijvoorbeeld een hek).

4.3.1 Opeenvolging als `while`

1. We gaan verder met het vorige scenario, maar openen de wereld die hoort bij figuur 10 als volgt:
   (a) Klik met de rechtermuisknop in de wereld.
   (b) Kies **`void`** `populateFromFile( )`.
   (c) Ga naar het map 'worlds'.
   (d) Kies 'world_Aanroepen6movesAlsWhile.txt'
2. Beredeneer dat het volgende generieke algoritme juist is: "Zolang Mimi haar ei nog niet heeft gevonden, moet ze een stapje zetten."We hebben dus een herhaling: "stapje zetten", en een conditie voor die herhaling: "ei nog niet gevonden" (oftewel: "NIET ei gevonden").
3. Bekijk het stroomdiagram in figuur 12. Wat moet er herhaaldelijk uitgevoerd worden? Vul dit in bij B.



Figuur 12: Stroomdiagram "Zolang **niet** ei gevonden, zet een stap."

4. Bekijk de conditie. Welke **`boolean`** methode van Mimi kun je gebruiken om de conditie te checken? Beschrijf in jouw eigen woorden wanneer deze methode **`true`** en **`false`** oplevert.
5. Vul de juiste conditie in bij A in het stroomdiagram. Tip: gebruik de **`boolean`** methode uit de vorige stap in combinatie met 'NIET'
6. We gaan nu de code aan de `act( )` methode van MyDodo toevoegen. Dat zal er ongeveer zo uit gaan zien:

```
public void act ( ) {
    while ( CONDITIE IN RUIT ) {
        DOE DE AANROEP IN DE RECHTHOEK
    }
}
```

7. Vervang de code in `act( )` door het bovenstaande.
8. In de ruit staat 'NIET'. In code schrijven we dat als '!'. De conditie in de ruit schrijf je als code zo: `! foundEgg( )`. Vervang de tekst in hoofdletters na de **`while`** door de juiste code aanroep.
9. Vervang ook de rest van de tekst in hoofdletters door een methode van Mimi. Tip: de code behorende bij B in het stroomdiagram.
10. Pas het commentaar boven de `act( )` methode ook aan.
11. Compileer en test het programma. Werkt het programma niet zoals verwacht? Volg de stappen zoals beschreven bij hoofdstuk 'Debuggen' van 4.1.1.

4.3.2 Loop tot einde van de wereld

Schrijf een methode waarmee Mimi van een willekeurige plaats naar de rand van de wereld loopt.

1. We gaan verder met de vorige scenario, maar beginnen met een lege wereld:
   (a) Klik met de rechtermuisknop in de wereld.
   (b) Kies **`void`** `populateFromFile( )`.
   (c) Ga naar het map 'worlds'.
   (d) Kies 'world_empty.txt'
2. Zet Mimi op een willekeurige plek in de wereld neer.
3. Bedenk een algoritme waarmee Mimi naar de rechter rand van de wereld loopt. Het algoritme moet onafhankelijk zijn van waar ze initieel staat. Tip: Vul aan: "Zolang NIET .... moet Mimi .... "
4. Teken het bijbehorende stroomschema.
5. Bij welke beginsituaties werkt jouw algoritme?
6. Schrijf de bijbehorende methode **`void`** `walkToEdgeOfWorld( )`.
7. Zet ook commentaar bij jouw methode.
8. Compileer en test jouw methode met de rechtermuisknop. Doe dit met Mimi op verschillende plaatsen in de wereld.
9. Draai Mimi 180 graden. Werkt jouw methode ook de andere kant op? Loopt Mimi dan naar de linkerrand van de wereld toe? Pas zo nodig de commentaar bij jouw methode aan.

4.3.3 Om een hek heen lopen (zelf opeenvolging schrijven)

We gaan nu Mimi nog slimmer maken. Als er iets in de weg ligt tussen haar een haar ei, dan moet ze daar natuurlijk omheen lopen.

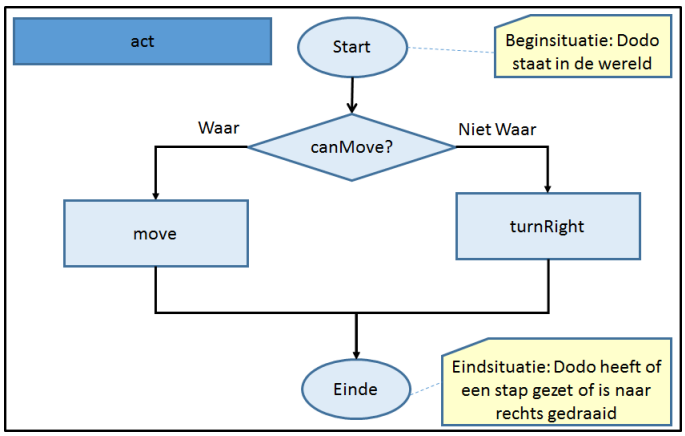
Figuur 13: Loop om het hek

We gaan de code zo aanpassen dat als Mimi een hek tegenkomt, ze er bovenlangs omheen loopt. Dat doen we als volgt:

1. We gaan verder met de vorige scenario. We passen de **void** act( ) eerst aan zodat deze weer is zoals in opdracht 1:
   (a) Open de code voor MyDodo.
   (b) Zoek de methode **void** act( ) op.
   (c) Tik de volgende code over:

```
public void act( ){
    if( canMove( ) ){
        move( );
    } else {
        turnRight( );
    }
}
```

2. Compileer en test jouw code.
3. Beredeneer dat jouw programma overeenkomt met het stroomdiagram in figuur 14.



Figuur 14: Stroomdiagram act( )

4. Open de wereld "world_eggFenceInWay". Weet je niet meer hoe? Kijk dan bij opgave 4.3.1 onderdeel 1.
5. Als Mimi nu tegen een hek aanloopt dan draait ze naar rechts en loopt ze verder. Check dit. Loopt ze gewoon door de hek heen, controleer dan of je opgave 4.2 goed hebt gemaakt.
6. We willen dat Mimi keurig om het hek loopt, zoals afgebeeld in figuur 13. Vul de volgende strategie aan.
   - Als Mimi geen stap vooruit kan zetten, dan:
     - draai naar links
     - zet een stap
     - draai naar ...
     - ...
   - Anders (dus Mimi kan wel een stap vooruit zetten): zet een stap.
7. Zorg dat Mimi na afloop weer naar rechts kijkt.
8. Teken het bijbehorende stroomdiagram.

9. Pas de code in de `act( )` methode van MyDodo aan zodat deze overeenkomt met jouw nieuwe stroomdiagram.
10. Compileer, run en test het programma. Werkt het programma niet correct? Volg de stappen zoals beschreven bij hoofdstuk 'Debuggen' van 4.1.1.

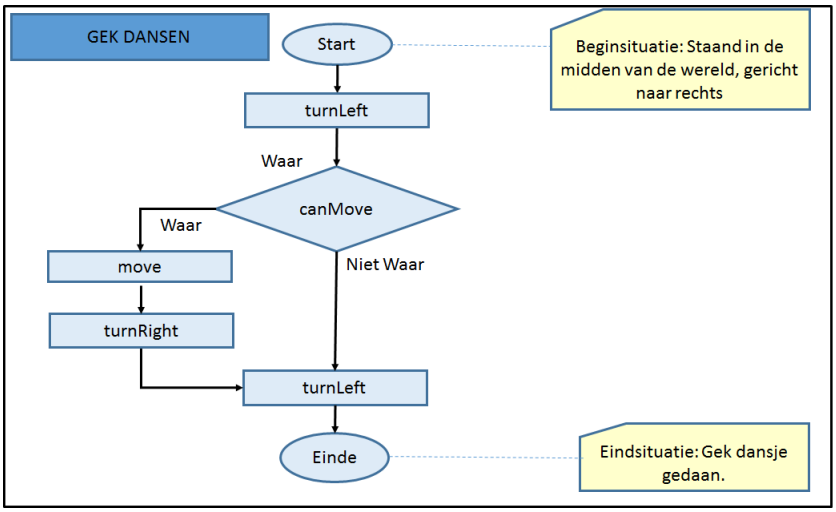
5 Samenvatting

Je hebt geleerd:

- een generieke oplossing te bedenken voor een probleem;
- een algoritme op te stellen als een opeenvolging van stappen, keuzes (**`if`** .. `then` .. **`else`**) of herhalingen(**`while`**);
- een algoritme weer te geven in een stroomdiagram;
- een stroomdiagram om te zetten naar programmacode;
- gestructureerd en stapsgewijs code aanpassen, compileren, uitvoeren en testen;
- stapsgewijs debuggen;
- na te denken over de kwaliteit van een oplossing.

5.1 Diagnostische toets

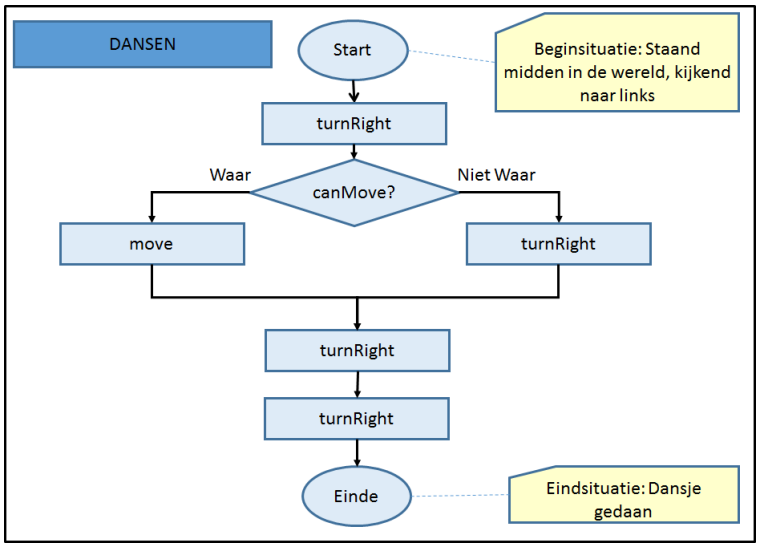
1. Gegeven is de methode **`boolean`** `alleEitjesGevonden()` die aangeeft of Mimi al haar eitjes gevonden heeft. Welke van de volgende is waar over de bijbehorende methode?
   - Deze methode heeft een **`boolean`** parameter.
   - De begin- en eindsituatie van de methode zijn aan elkaar gelijk.
   - Deze methode levert een **`boolean`** op.
   - Dit is een mutatormethode.
2. Noem 2 voordelen van het gebruik maken van submethodes?
3. Noem 2 redenen om meteen na elke kleine aanpassing te testen?
4. Beschrijf wat de begin- en eindsituaties met de programmacode te maken hebben
5. Mimi leert dansen. Teken het stroomdiagram dat hoort bij de volgende danspas:
   - Beginsituatie: Je staat in het midden van de wereld, kijkend naar links.
   - Danspas: Draai naar rechts. Als je een stapje vooruit kan zetten, doe dat. Anders draai je weer naar rechts. Daarna draai je nog 2 keer rechts.
6. Gegeven het stroomdiagram in Figuur 15, schrijf de bijbehorende programmacode.



Figuur 15: Stroomdiagram voor Gek Dansen

5.2 Uitwerking diagnostische toets

1. Gegeven is de methode **`boolean`** `alleEitjesGevonden()`.
   - Niet waar: Deze methode heeft een geen parameters.
   - Waar: Het is een opleverende methode. Daarvoor hebben we afgesproken dat de begin- en eindsituatie aan elkaar gelijk zijn.
   - Waar: Deze methode levert een **`boolean`** op.
   - Niet waar: Dit is een opleverende methode.
2. Om overzichtelijkheid, onderhoudbaarheid, herbruikbaarhid en testbaarheid te bevorderen, om herhaling van code te beperken
3. Eenvoudiger (sneller) opsporen van fouten, eenvoudiger te verifiëren of aanpassingen doet wat het moet doen (en niet doet wat het niet moet doen)
4. De beginsituatie beschrijft de voorwaarden waaronder de methode correct zal functioneren. De eindsituatie beschrijft de verwachtte situatie na het uitvoeren van de methode. Op deze manier kun je makkelijk testen of de code doet wat verwacht wordt. Verder bevordert het hergebruik van de code in andere onderdelen van het programma (of in andere programma's).
5. Zie figuur 16



Figuur 16: Antwoordmodel voor 5

6.
```
/*
* Doe een gek dansje
* beginsituatie: staand in het midden van de wereld, kijkend naar rechts
*/
public void gekDansen( ) {
    turnLeft( );
    while ( canMove( ) ) {
        move( );
        turnRight( );
    }
    turnLeft( );
}
```

6 Opslaan en inleveren

6.1 Opslaan

Je bent klaar met de tweede opdracht. Sla je werk op, want je hebt het nodig voor de volgende opdrachten.

1. Kies in Greenfoot 'Scenario' in het bovenste menu, en dan 'Save As ...'.
2. Vul de bestandsnaam aan met jouw eigen naam en het opgavenummer, bijvoorbeeld `Opdr2_Michel`.
3. Voeg hier foto's of scans toe van de stroomdiagrammen.

Alle onderdelen van het scenario bevinden zich nu in een map die dezelfde naam heeft als de naam die je hebt gekozen bij 'Save As ...'.