Opdracht 1: Maak kennis met Dodo

– Algoritmisch Denken en Gestructureerd Programmeren in Greenfoot –

1 Inleiding

In deze opgavenreeks *Algoritmisch denken en gestructureerd programmeren (in Greenfoot)* kun je op een leuke manier de basisvaardigheden leren voor Objectgeoriënteerd programmeren. Na afloop kun je dan, met behulp van de Greenfoot en Java bibliotheken, programmeren in Java. Met deze basisvaardigheden ben je ook in staat om makkelijk andere Objectgeoriënteerde talen leren. Je leert gebruik maken van voorgegeven stukken programmacode en deze uit te breiden waar nodig. Hergebruik van bestaande programmacode is een van de krachten van Java. Uiteraard leer je ook jouw eigen code schrijven.

Je programmeert in de taal Java. In de opdrachten werken we met de *Greenfoot* programmeeromgeving. Dit is een omgeving die je helpt Java programma's te schrijven zonder eerst uitgebreid de theorie van programmeren te leren. Zo kun je gaandeweg Java leren terwijl je er mee bezig bent.

Je maakt kennis met de basisbouwstenen waarmee je algoritmen en programma's maakt. We zullen beginnen met makkelijke opgaven zoals een Dodo laten lopen en draaien. Dit bouwen we in de loop van de cursus uit. Je zult de Dodo complexe taken leren voltooien, zoals zelfstandig haar eieren die verspreid in de wereld liggen te verzamelen. Je zult daarbij een wedstrijd aangaan met je medeleerlingen: wie kan de slimste Dodo maken?

Als je wilt leren schaken, dan moet je de spelregels kennen. Die spelregels vertellen je wat wel en niet mag. Wil je goed kunnen schaken en het leuk vinden, dan is dit niet genoeg. Je moet ook leren wat een goede zet is. Er zijn natuurlijk talloze Java tutorials en handboeken te vinden op het internet. Die leren je de regels, maar weinig over hoe je het schrijven van een programma aanpakt. In deze cursus leer je hoe je zo'n programmeertaak systematisch aanpakt. Je leert een probleem analyseren en op een gestructureerde en overzichtelijke manier te zoeken naar een oplossing.

2 Leerdoelen

Na afloop van deze opdracht kun je:

- je weg vinden in de programmeeromgeving Greenfoot;
- uitleggen wat de Greenfoot *Run* en *Act* knoppen doen;
- methodes aanroepen en hun effect analyseren;
- het verschil tussen een **mutatormethode** en een **accessormethode** in jouw eigen woorden omschrijven;
- de methoden van een **klasse** kunnen opzoeken en benoemen;
- aan de hand van een **klassediagram**, de klassen benoemen;
- aan de hand van een klassendiagram, de **subklassen** van een klasse benoemen;

- de methoden benomen die een object erft van een ander klasse door **overerving**;
- uitleggen wat een **object** of **instantie** van een klasse is;
- eigenschappen van de **toestand** van een object benoemen;
- naamgevingsafspraken voor methoden en parameters benoemen;
- in je eigen woorden uitleggen wat een **type** is;
- omschrijven wat de typen **`int`**, `String`, **`void`** en **`boolean`** zijn;
- aan de hand van een **signatuur**, het **resultaattype** en **parameter(type)s** van een methode benoemen;
- de rol van een resultaattype en parameters in een methode kunnen beschrijven;
- de relatie tussen een **algoritme**, een **stroomdiagram** en **programmacode** in eigen woorden kunnen omschrijven;
- stapsgewijs codeaanpassingen doorvoeren en testen;
- **commentaar** in code herkennen en toevoegen;
- een methode toevoegen, compileren, uitvoeren en testen;
- **syntax foutmeldingen** herkennen en interpreteren.

3 Instructies

Voor deze opdracht heb je scenario **'DodoScenario1'** nodig. Instructies voor het installeren en openen van de Greenfoot omgeving worden in de opdracht gegeven.

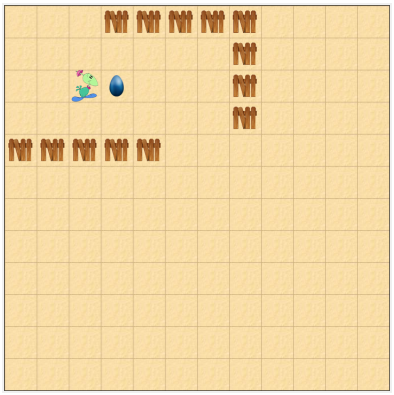
4 Uitleg: Wegwijs in Greenfoot

4.1 De wereld

Dit is onze Dodo. Ze heet Mimi en behoort tot de *Dodo* familie.

Figuur 1: Dodo Mimi

Mimi leeft in een wereld die bestaat uit 12 bij 12 vakjes. Deze wereld is begrensd, zij kan er niet uit. Haar levensdoel is het leggen en uitbroeden van eieren. In het plaatje hieronder zie je Mimi in de wereld met haar snavel naar rechts gericht. Voor haar ligt een ei. Ook staan er een aantal hekjes in de wereld.



Figuur 2: De wereld van Mimi

Mimi is een heel brave Dodo. Ze doet namelijk altijd precies wat haar wordt gezegd en geeft altijd eerlijk antwoord op jouw vragen (en ze liegt nooit). Door *methoden* aan te roepen kun je haar opdrachten geven (commando's) of vragen stellen.

Je kunt Mimi opdrachten laten uitvoeren, zoals:

`move`	stap een hokje verder
`hatchEgg`	broed een ei uit
`jump`	spring een aantal vakjes vooruit

Je kunt ook een vraag stellen waar Mimi antwoord op geeft, zoals:

`canMove`	Kun je een stapje opzij zetten?
`getNrOfEggsHatched`	Hoeveel eieren hebben je uitgebroed?

5 Aan de slag met Greenfoot

5.1 Opstarten

We gaan nu in Greenfoot oefenen met Mimi. Hiervoor moeten we eerst (als dat nog niet gedaan is) Greenfoot downloaden en installeren. Daarna openen we het voorgegeven scenario in Greenfoot.

Kies een map:

- Bedenk een vaste plek waar je alle scenario-informatie gaat opslaan. Je kunt hiervoor eventueel een USB-stick gebruiken.

Downloaden:

- Download Greenfoot: Ga naar `http://www.greenfoot.org/download` en volg de aangegeven instructies.
- Download de voorgegeven scenario bestanden die bij deze opdracht horen: Kopieer het voorgegeven *zip*-bestand met startscenario naar de door jou uitgekozen opslagplek.
- Pak het *zip*-bestand hier uit (rechts-klikken op het bestand en *Extract All...* selecteren).

Open de Greenfootomgeving:

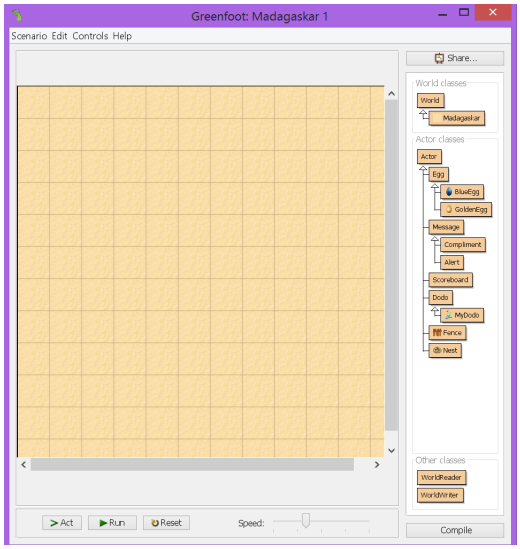
- Ga naar **Start** - *All Programs* - *Greenfoot* en selecteer hier *Greenfoot*. De omgeving hoort nu op te starten.

Open het scenario:

- Selecteer 'Scenario' in het hoofdmenu en daarna 'Open'.

- Navigeer naar het scenario **'DodoScenario1'** en kies 'Open'.
- Druk rechtsonder op de knop 'Compile'.

Als het goed is krijg je het volgende te zien:

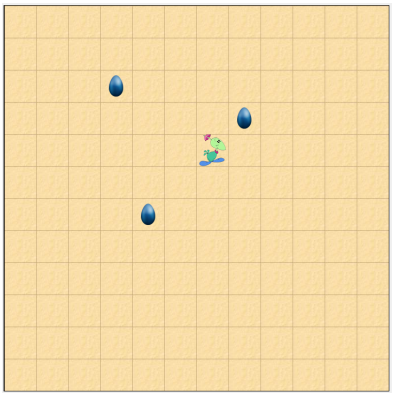


Figuur 3: Wat je te zien krijgt na het openen van 'DodoScenario1'

5.2 Objecten aanmaken

Objecten
De wereld bestaat uit objecten (ook wel *klasse-instanties*, of kortweg *instanties* genoemd). Met `new MyDodo( )` maak je een nieuwe object van `MyDodo` aan.

In de wereld hieronder zie je één `MyDodo` object en drie `Egg` objecten.



Figuur 4: Scenario met één `MyDodo` en drie `Egg` objecten

In onze opdrachten hebben we één hoofdrolspeelster, een `MyDodo`-object. Ze is een instantie van `MyDodo` en we noemen haar Mimi.

We gaan het scenario maken zoals in figuur 4 is weergegeven:

1. Klik met de rechtermuisknop op *MyDodo*, helemaal rechts in het scherm.

2. Kies **new** `MyDodo()` om een nieuw `MyDodo`-object te maken.
3. Sleep het object, onze MyDodo, naar een willekeurige vak in de wereld.
4. Leg op dezelfde manier de drie eieren in de wereld neer. Tip: houd 'Shift' ingedrukt om meerdere eieren neer te leggen.

5.3 De wereld in beweging

Greenfoot heeft een aantal knoppen onderaan het venster.

1. Klik op de *Act* knop (onderin het scherm). Wat gebeurt er?
2. Wat gebeurt er als je nog eens op *Act* klikt? En nog eens? En nog eens?
3. Experimenteer wat er gebeurt als Mimi op verschillende plekken staat. Je kunt Mimi met de muis in de wereld verplaatsen. Zorg dat je verschillende situaties probeert, bijvoorbeeld met Mimi aan de rand van de wereld met haar snavel naar buiten gericht.
4. Beschrijf heel precies in jouw eigen worden van de knop *Act* doet.
5. Druk op de *Run* knop. Wat gebeurt er?
6. Wat hebben *Act* en *Run* met elkaar te maken?
7. Wat doet de knop 'Reset'? Probeer het uit.

5.4 Methoden

Methoden en hun resultaten
Er zijn twee soorten methoden, ieder met een eigen doel:

1. *Mutatormethoden*: Dit zijn opdrachten waardoor een object iets uitvoert. Hierdoor verandert de toestand van het object. Deze mutatoren kun je herkennen aan het resultaat **void**. Bijvoorbeeld **void** `act( )`.
2. *Accessormethoden*: Dit zijn vragen die informatie over (de toestand van) een object opleveren. Bijvoorbeeld **boolean** `canMove( )` die met het resultaat **true** ('waar') of **false** ('niet waar') aangeeft of Mimi al dan niet kan lopen. Een ander voorbeeld is **int** `getNrOfEggsHatched()` met als resultaat een **int** (een geheel getal) dat aangeeft hoeveel eieren Mimi heeft uitgebroed. Deze accessoren geven enkel informatie over het object terug en dienen daarbij de toestand van het object ongewijzigd te laten.

5.4.1 Mutatormethode ontdekken

Onze MyDodo is geprogrammeerd om een aantal dingen te doen. Met *mutatormethodes* kan Mimi opdrachten uitvoeren.

1. Klik met de rechtermuisknop op Mimi. Zo zie je wat Mimi allemaal kan (met andere woorden: welke methoden je kunt aanroepen), bijvoorbeeld `act( )`, `move( )` en `hatchEgg( )`.
2. Roep `act( )` aan. Wat doet die?
3. Kijk naar jouw beschrijving van wat de *Act* knop doet (uit opgave 5.3 onderdeel 4). Zou er een verschil zijn tussen het klikken op de *Act* knop en het aanroepen van de `act( )` methode?

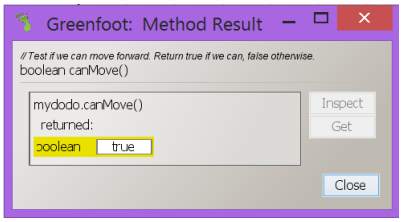
4. Leg een ei neer in de wereld. Sleep Mimi op het ei. Zet je muis op Mimi en niet op het ei. Roep met de rechtermuisknop de methode **`void`** `hatchEgg( )` van Mimi aan. Wat gebeurt er?

5.4.2 Accessormethode ontdekken

Met *accessormethodes* kan Mimi vragen beantwoorden waarmee ze informatie over haarzelf geeft.

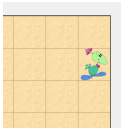
1. Zoals je wellicht weet, een Dodo kan niet vliegen. Maar ze kan wel lopen. Tenminste, als ze daardoor niet uit de wereld stapt. Voordat zij een stapje zet kan ze kijken of dat mag. Zet Mimi ergens in het midden van de wereld. Als je Mimi zou vragen of ze een stapje kan zetten, welk antwoord verwacht je?

2. Roep de methode **`boolean`** `canMove( )` aan. Er verschijnt een venstertje zoals in het plaatje hieronder. Je ziet dat er **`true`** ('waar') staat. Wat betekent het antwoord dat je terugkrijgt? Is dat hetzelfde antwoord als je in de vorige vraag (5.4.2 onderdeel 1) verwacht had?



Figuur 5: Dialoog voor `canMove( )`

3. Verplaats Mimi zodat ze aan de grens van de wereld staat met haar snavel naar buiten gericht, zoals in figuur 6. Roep **`boolean`** `canMove( )` opnieuw aan. Wat levert het nu op?



Figuur 6: Dodo staat aan de grens van de wereld

4. Roep nu de methode **`int`** `getNrOfEggsHatched( )` aan. Wat levert die op?

5. De **`int`** `getNrOfEggsHatched()` methode kun je gebruiken om te zien hoeveel eieren MyDodo heeft uitgebroed. Het levert een **`int`** (een geheel getal) op. Kun je een situatie maken zodat de methode **`int`** `getNrOfEggsHatched( )` de waarde 3 oplevert? (Met andere woorden, kun je je Dodo drie eieren laten uitbroeden?). Tip: gebruik hiervoor eerst `hatchEgg( )`.

5.5 Overerving

Klassen
Elk object hoort bij een klasse. Zo is Mimi een *instantie* van de klasse `MyDodo`.

Klassendiagram:
Dat `MyDodo` tot het Dodoras behoort kun je zien in het klassendiagram, in Greenfoot rechts op het scherm. Zie figuur 7. De pijl in het diagram geeft een 'is-een' relatie aan, `MyDodo` behoort tot de klasse `Dodo`. Dus `MyDodo` 'is-een' `Dodo`.

Figuur 7: Klassendiagram Dodo

Methoden van MyDodo:
Mimi is een `MyDodo`. Mimi heeft methoden die je kunt aanroepen. Door met de rechtermuisknop te klikken op Mimi kun je zien wat een `MyDodo` (en Mimi dus ook) allemaal kan. Bijvoorbeeld `move( )`.

Methoden van Dodo:
Maar Mimi (onze `MyDodo`) kan veel meer. Er zijn dingen die alle Dodo's in het algemeen kunnen. Mimi kan dat dan natuurlijk ook, ze is tenslotte ook een Dodo! Klik met je rechtermuisknop op Mimi (in de wereld). Bovenaan de lijstje zie je 'inherited from Dodo' staan. Als je daarop klikt, dan zie je ineens meer methoden, bijvoorbeeld `layEgg()`. Dat zijn de methoden van Dodo, de dingen die alle Dodo's kunnen. Omdat `MyDodo` een `Dodo` is (zie de klassendiagram in figuur 7), erft (oftewel *inherits* in het Engels) ze deze Dodo-methoden. Een `MyDodo` kan dus alle `Dodo` methoden uitvoeren.

Overerving:
Met *overerving* kun je een nieuwe klasse introduceren als uitbreiding van een bestaande klasse. In zo'n situatie wordt de bestaande klasse ook wel de *superklasse* genoemd, terwijl de uitgebreide klasse aangeduid wordt met de term *subklasse*. Omdat `MyDodo` een subklasse van `Dodo` is kan een `MyDodo` dus methoden uitvoeren uit zowel de `MyDodo`-klasse als van de `Dodo`-klasse.

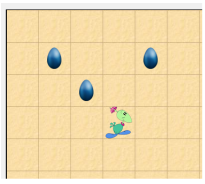
Stel er zou een nieuwe Dodosoort geboren worden: `IntelligentDodo`. Deze `IntelligentDodo` is dan een subklasse van `Dodo`. Die kan dan alles wat een `Dodo` kan. Nu hoef je, bijvoorbeeld, niet opnieuw te beschrijven dat deze nieuwe dodosoort een ei kan leggen: `layEgg( )`. Dit voorkomt dat we heel veel dubbele code krijgen. Met overerving beschrijven we alléén nog maar de extra dingen die zij kan (haar intelligente gedrag).

Maar, wat zijn dan de dingen die alle Dodo's kunnen? We gaan nu kijken naar de *mutatormethodes*.

1. Noem minstens drie `Dodo`-methoden die Mimi erft.
2. De methode **`void`** `turnLeft( )` behoort tot de klasse Dodo. Kan Mimi die methode ook uitvoeren? Probeer het maar.
3. Bekijk de klassendiagram. Welke andere 'is-een' relaties zie je? Noem er ten minste twee.
4. Zet nu een tweede MyDodo in de wereld. Vergelijk haar methodes met die van Mimi (die al in de wereld staat). Zie je verschillen?

5.6 Toestanden

Toestand van een object
Je kunt verschillende objecten in de wereld zetten, bijvoorbeeld een Dodo en een ei. Je kunt ook meerdere objecten van dezelfde klasse in de wereld zetten. Hier zie je drie objecten van de klasse `Egg` en één van de klasse `MyDodo`.



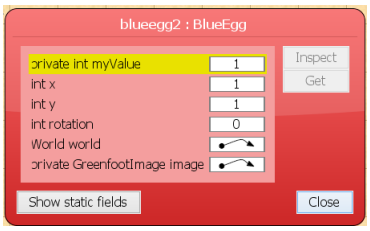
Figuur 8: Wereld met meerdere objecten

Alle objecten van dezelfde klasse hebben dezelfde methodes, en dus hetzelfde gedrag. De drie objecten van `Egg` zien er hetzelfde uit en kunnen precies hetzelfde doen (ze hebben dezelfde methodes). Toch zijn het andere objecten, of *instanties*.

Net als dat jij anders bent dan de persoon die naast je zit, terwijl jullie allebei personen zijn. Dat jullie ieder op een andere plek zitten, of wat anders voor het ontbijt hebben gehad, maakt je al anders. In onze Greenfoot wereld is dat net zo. Elk object heeft een eigen toestand. Twee objecten kunnen bijvoorbeeld op verschillende coördinaten staan.

De toestand van een object kun je bekijken door met de rechtermuisknop op het object te klikken, en dan 'Inspect' te kiezen.

Figuur 9: Toestand van het `Egg`-object met coördinaten (2,2)



Figuur 10: Toestand van het `Egg`-object met coördinaten (1,1)

We bekijken nu toestanden van objecten.

1. Sleep Mimi naar de linkerbovenhoek van de wereld.
2. Klik met de rechtermuisknop op Mimi en kies 'Inspect'. Op welke coördinaten (**`int`** `x` en **`int`** `y`) staat MyDodo?
3. Sleep Mimi naar de rechterbovenhoek van de wereld. Wat zijn de coördinaten van het vakje in de rechterbovenhoek?
4. Sleep Mimi naar een willekeurige hokje. Bedenk wat de coördinaten van het hokje zijn.
5. Controleer jouw antwoord met 'Inspect'.

5.7 Parameters en resultaten

Resultaten
In 5.4 hebben we gezien dat een accessormethode informatie over de toestand van een object oplevert. Zo'n antwoord heet een *resultaat*. Bijvoorbeeld **`int`** `getNrOfEggsHatched`() die een **`int`** (geheel getal) als resultaat heeft.

Parameters
Methoden kunnen ook bepaalde waardes meekrijgen die ze meer informatie geeft voor een bepaalde taak. Die meegegeven waardes heten *parameters*. Een methode kan 0 of meer parameters hebben. Voorbeelden:

- **`void`** `jump (`**`int`** `distance`) heeft één parameter, `distance` waardoor de methode weet hoe ver er gesprongen moet worden. Daarnaast weten we dat `distance` een **`int`** (geheel getal) is. Door de aanroep van `jump (3);` zal Mimi 3 vakjes verplaatsen. Door de aanroep van `jump (5);` zal Mimi 5 vakjes verplaatsen. De methode `jump` is zo geschreven dat deze alléén kan worden aangeroepen met de aanvullende informatie.
- **`void`** `move( )` heeft geen parameters.

Een methode met parameters is flexibeler dan een methode die geen parameters heeft. De methode **`void`** `jump(`**`int`** `distance)` kan Mimi over verschillende afstanden laten springen, terwijl **`void`** `move( )` telkens maar één stap zet.

Types
Parameters en resultaten hebben een type. Voorbeelden van verschillende types zijn:

Type	Betekenis	Voorbeeld
`int`	geheel getal	2
`boolean`	'waar' of 'niet waar'	**`true`**
`String`	tekst	"Mijn fiets is gestolen!"
`List`	lijst	[1,2,3]

Een type kan ook een klasse zijn, zoals een lijst (`List`) van dingen.

Het type geeft aan welke soorten waardes een parameter of resultaat moet hebben. Een aanroep van **`void`** `jump(`**`int`** `distance)` zonder parameters mag dus niet, want deze methode *moet* een **`int`** meekrijgen.

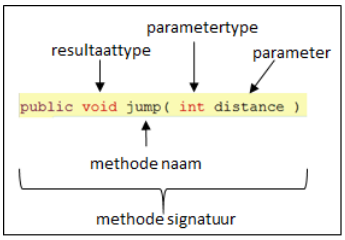
We hadden gezien dat je met 'Inspect' informatie over de toestand van Mimi kon bekijken. Accessoren zijn gemaakt om die informatie als resultaat op te leveren. We bekijken nu een voorbeeld van een methode met zo'n resultaat.

1. Klik met je rechtermuisknop op Mimi.
2. Er is ook een methode van de klasse `Actor` waarmee je de coördinaten van een object kan opvragen. Kun je die vinden? Tip: `MyDodo` behoort ook tot de klasse `Actor`. Dit kun je zien in het klassendiagram. Zoek tussen de 'inherited from Actor' methoden naar een methode met een `X`. Doe hetzelfde voor de `Y`.
3. Wat valt je op aan de naam van de methode? Waarom zou er 'get' bijstaan? Kijk eventueel terug bij 5.4.
4. Wat voor een type leveren die methoden op? Hoe herken je dat aan de hand van de methode waar je op klikt?

We bekijken nu een paar voorbeelden van methoden met parameters.

1. Zet Mimi ergens midden in de wereld neer. Klik met de rechtermuisknop op Mimi. Roep de methode `jump(int distance)` aan. Aan deze methode geef je een **`int`** (geheel getal) mee ('distance') dat aangeeft hoe ver Mimi moet springen. Toets een kleine waarde in en kijk wat er gebeurt.
2. Roep de methode `jump(int distance)` weer aan, deze keer met een getal groter dan 11. Wat gebeurt er? Kun je dat beredeneren?
3. Wat gebeurt er als je een waarde invoert die geen **`int`** (geheel getal) is, bijvoorbeeld 2.5?
4. Bekijk de foutmelding. Er staat 'incompatible types'. Dat betekent dat je iets anders hebt ingevoerd dan verwacht is. De methode verwacht namelijk een **`int`** (geheel getal), maar je hebt een **`double`** (decimaal getal) ingevoerd.
5. Wat gebeurt er als je een woord, bijvoorbeeld "twee" invoert?

Signatuur
Aan de hand van de signatuur kun je zien welke parameters, parametertypes en resultaattype bij een methode horen. Wat dat betekent leggen we hier uit.



Figuur 11: Signatuur van een methode

- *Methodenaam:* voor het eerste haakje staat de naam van de methode. In dit voorbeeld: `jump`.
- *Parameter:* tussen de haakjes '(' en ')' staat aangegeven welke parameters de methode als input krijgt. In dit voorbeeld: `distance`.
- *Parametertype:* het type (zoals **`int`**, **`boolean`**, `String`) van de parameter staat voor de naam van de parameter. In dit voorbeeld: **`int`**.
- *Resultaattype:* het type (zoals **`int`**, **`boolean`**, **`void`**) van het resultaat staat voor de naam van de methode. In dit voorbeeld is het resultaattype **`void`**. Deze methode levert niets op. Het is dus een mutatormethode.
- *Methode signatuur:* het geheel van alle bovengenoemde onderdelen. In dit voorbeeld: **`public void`** `jump (int distance)`. (wat **`public`** is komt later aan bod)

Het type in de signatuur geeft aan welke soorten waarden een parameter of resultaat moet hebben.

5.8 Omschrijven, lezen, aanpassen, compileren en testen van programma's

We hebben nu geoefend met het aanroepen van methoden van `MyDodo`. Om leukere dingen met Mimi te kunnen doen moet je zelf nieuwe methoden schrijven. Hiervoor moet je natuurlijk eerst

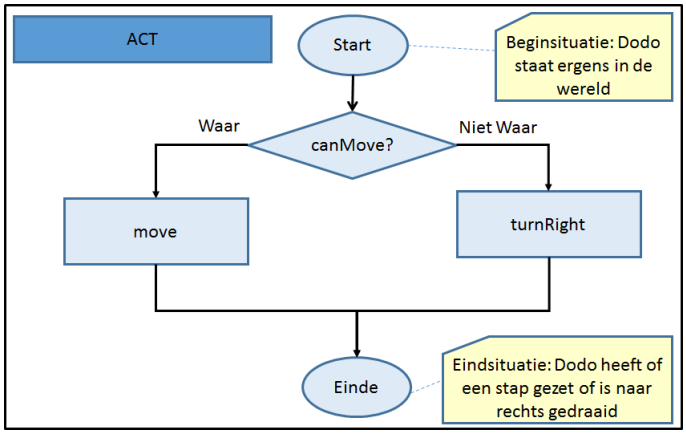
de programmacode kunnen lezen. Ook moet je jouw aanpassingen kunnen omschrijven om de wijzigingen daarna in de code door te voeren.

5.9 Gedrag omschrijven

Algoritme
Een *algoritme* is een reeks van heel precies omschreven instructies (of stappenplan). *Programmacode* is een algoritme geschreven voor een computer. Deze vertelt dus precies, stap voor stap, wat de computer moet doen.

Als een stappenplan voldoende nauwkeurig is opgeschreven, zou iemand anders het precies na kunnen doen. Het lijkt een beetje op een recept. Met hetzelfde probleem (*beginsituatie*) zou die andere persoon dan tot dezelfde oplossing komen (*eindsituatie*). Het verschil met een recept is dat een stappenplan nog preciezer is. In een recept zou kunnen staan: roer tot glad geheel, maar de ene keer zou je toetje misschien wat minder glad kunnen zijn dan de andere keer. Dat mag niet in een algoritme. Voor elke stap moet duidelijk zijn wat deze precies inhoudt. Een stap die op verschillende manieren geïnterpreteerd kan worden heet *ambigu*. Ambigue stappen proberen we te vermijden: het resultaat moet elke keer precies hetzelfde zijn.

Stroomdiagram
Een algoritme kan je overzichtelijk weergeven in een stroomdiagram.
We bekijken als voorbeeld het algoritme van `act( )` van `MyDodo`. Dit heb je al onderzocht in opgave 5.4.1 onderdeel 2.



Figuur 12: Stroomdiagram van de code `act( )`

Toelichting stroomdiagram:

- Linksboven staat de naam van de methode waar het stroomdiagram bij hoort, *Act*.
- Rechtsboven staat de beginsituatie in een notitieblokje beschreven.
- Onderaan staat de eindsituatie in een notitieblokje beschreven.
- Er wordt bovenaan bij 'Start' begonnen.
- Volg de pijl naar de ruit. Een ruit betekent dat er een keuze gemaakt moet worden, afhankelijk van het resultaat van 'canMove?' wordt er een ander pad vervolgd:

  - Als 'canMove?' 'Waar' is, dan wordt de pijl 'Waar' naar links gevolgd.
  - Als 'canMove?' 'Niet Waar' is, dan wordt de pijl 'Niet waar' naar rechts gevolgd.

  Dit gedrag komt overeen met dat van de **`if`** .. `then` .. **`else`** in de code.

- Bij een rechthoek wordt een methode uitgevoerd. Dus, afhankelijk van welke pijl er gevolgd is, zal of 'move' of 'turnRight' uitgevoerd worden.
- Bij het bereiken van 'Einde' is *Act* afgelopen.

5.9.1 Programmacode lezen

Programmacode lezen
We bekijken als voorbeeld weer dezelfde methode `act( )` in `MyDodo`.

```
/**
 * Go to the edge of the world and
 * walk along the border
 */
public void act( ) {
    if ( canMove( ) ) {
        move( );
    } else {
        turnRight( );
    }
}
```

Toelichting code:

- *Commentaar*: de tekst tussen /* en */ bovenaan (in de Greenfoot editor in blauw weergegegeven) is commentaar. Greenfoot/Java doet daar niets mee. Programmeurs schrijven commentaar boven en in hun methoden om anderen te helpen hun programma's te begrijpen. Maar ook voor jezelf kan het nuttig zijn, als je bijvoorbeeld een weekje niet naar je code hebt gekeken kun je met zo'n toelichting snel weer zien wat de methode ook al weer deed zonder uitvoerig de programmacode zelf te hoeven bestuderen. Voor commentaar op maar één regel kun je ook // gebruiken.
- *Signatuur*: deze is **`public void`** `act( )`.
- *Access-modifier*: deze is **`public`**. Wat dit betekent en welke andere mogelijkheden er zijn, komt later aan bod.
- *Methodenaam*: deze is `act`.
- *Resultaattype*: dat is **`void`**. Zoals we in hoofdstuk 5.4 zagen betekent dit dat de methode iets uitvoert en geen waarde oplevert.
- *Body*: deze staat tussen de eerste { en de bijbehorende accolade }. Dit is de code die uitgevoerd wordt als je `act( )` aanroept, oftewel, wat de methode doet. In dit voorbeeld staat een *conditional statement*: dit is de **`if`** .. `then` .. **`else`** in de `act( )`. Dat werkt zo:
  - Tussen de haakjes '(' en ')' staat een conditie. Er wordt eerst gecontroleerd of de conditie **`true`** is of niet.
  - Als deze **`true`** is, dan wordt het gedeelte direct achter de conditie uitgevoerd.

– Als deze **`false`** is, dan wordt het gedeelte achter de **`else`** uitgevoerd.

Dus als je op *Act* knopje drukt of via de rechtermuisknop **`void`** `act( )` kiest, dan wordt:

– eerst gekeken of conditie waar is. De methode `canMove( )` wordt aangeroepen om te kijken of Dodo een stapje kan zetten.

– Als `canMove( )` **`true`** is, dan wordt `move( )` aangeroepen en zet Dodo een stapje naar voren.

– Als `canMove( )` **`false`** is, dan wordt `turnRight( )` aangeroepen en draait Dodo een kwartslag naar rechts.

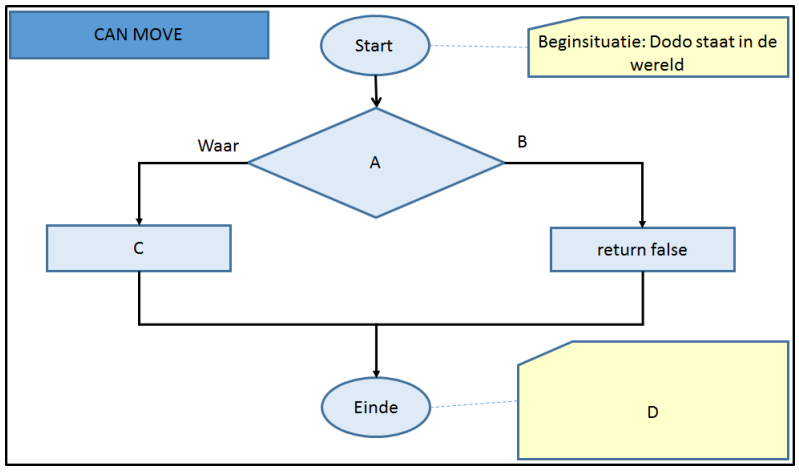
We gaan de programmacode nu in Greenfoot opzoeken en bekijken:

1. Klik in het klassendiagram met de rechtermuisknop op `MyDodo` en kies 'Open editor'.
2. Zoek de methode `act( )` op. Tip: gebruik Ctrl+F.
3. Komt de bovenstaande uitleg van `act( )` overeen met wat jij dacht in opgave 5.4.1 onderdeel 2?

5.9.2 Programmacode en het bijbehorende stroomdiagram

We bekijken nu de methode `canMove( )`.

1. Open de code voor `MyDodo` in de editor en zoek de methode **`boolean`** `canMove( )`.
2. Leg voor elke regel code uit wat die regel precies betekent. Toelichting: In de code wordt het symbool '!' gebruikt. Dit staat voor *negatie* (wordt als 'not' of, in het Nederlands, als 'niet' uitgesproken).
3. Bekijk het stroomdiagram in figuur 13. Dit geeft het algoritme van **`boolean`** `canMove( )` weer. Toelichting:
   - De ruit geeft een keuze aan.
   - Als de conditie in de ruit 'Waar' is, dan wordt de linker pijl vervolgd en die stappen uitgevoerd.
   - Anders (de conditie is 'Niet waar'), dan wordt de rechter pijl vervolgd.



Figuur 13: Stroomdiagram `canMove( )` in MyDodo

4. Wat moet er bij A, B, C en D in het stroomdiagram staan? Vul dit aan.

5.9.3 Commentaar toevoegen aan code van `move( )`

We bekijken nu de methode `move( )`.

1. Zoek de code van de methode `move( )` op in `MyDodo`.
2. Wat is de signatuur van de methode?
3. Wat is de resultaattype van deze methode?
4. Bekijk de body van de methode, oftewel, wat tussen de eerste accolade { en de bijbehorende } staat.
5. Er staat een **`if`** .. `then` .. **`else`** statement. Afhankelijk van of `canMove( )` 'Waar' of 'Niet waar' is gebeurt er iets anders. Sleep Mimi naar verschillende posities in het scherm en voer met de rechter muisknop `move( )` aan. Probeer beide situaties na te bootsen. Wat gebeurt er precies?
6. Ga nu terug naar de code en bekijk het commentaar boven de methode (dus het stukje tussen de /* en */). Deze beschrijft niet goed wat de methode doet. Er gebeurt namelijk veel meer. Pas het commentaar aan zodat deze beter omschrijft wat de methode doet.

5.9.4 Compileren, Uitvoeren en Testen

Aanpassingen
Voer aanpassingen stapsgewijs door. Omdat een tikfoutje makkelijk gemaakt is (door bijvoorbeeld een ';' te vergeten in te typen) doe je er goed aan om bij elke (kleine) wijziging in de code opnieuw te compileren, uit te voeren en testen.

Dat er iets gewijzigd is zie je aan het klassendiagram. Het blokje van de gewijzigde klasse is grijs gearceerd. Dat betekent dat de code van die klasse veranderd is en (opnieuw) moet worden gecompileerd.

Na elke aanpassing:

1. *Compileren*: druk op de *Compile* knop rechtsonderin het venster
2. *Uitvoeren*: druk op *Run*.
3. *Testen*: controleer of de methode doet wat je verwacht. Roep de methode aan via de rechtermuisknop of druk op *Act*. Vergelijk daarbij de begin- en eindsituaties.

Tip: Leer jezelf aan om deze stappen telkens meteen na elke aanpassing te doen. Als je namelijk een foutje hebt gemaakt vind je zo nog makkelijk de oorzaak. Als je veel meer wijzigingen in één keer doorvoert zonder tussendoor steeds te testen kan debuggen (het opsporen van fouten in je code) een hels karwei worden!

Je hebt net commentaar toegevoegd aan de code bij `move( )`. Dit zou aan het gedrag van het programma niets moeten veranderen. Het wordt wel herkend als een aanpassing.

1. Sluit de editor.
2. Hoe herken je van welke klasse de code gewijzigd is?
3. Druk op de knop 'Compile' rechtsonderin om alle klassen opnieuw te compileren.

4. Herstel zo nodig de fouten die de compiler meldt.
5. *Run* en test de programma. Doet de methode `act( )` nog wat je verwacht?

5.9.5 Nieuwe methode toevoegen

Naamgevingsafspraken
Methodes en parameters hebben namen. Er zijn algemene afspraken over de manier waarop je in Java een naam kiest. Door je daaraan te houden is je code makkelijker leesbaar voor anderen.
Naam van een methode:

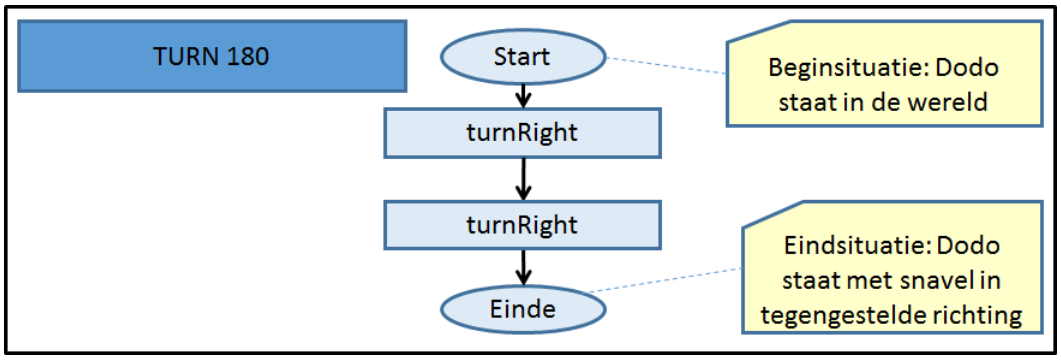
- is betekenisvol: hij komt overeen met wat de methode doet
- is in de vorm van een commando: bestaat uit één of meer werkwoorden
- bestaat uit letters en cijfers: bevat geen spaties, komma's of andere 'rare' tekens ('_' uitgezonderd)
- is geschreven in lowerCaseCamel: begint met een kleine letter, elk volgend 'woord' begint met een hoofdletter
- bijvoorbeeld: `canMove`

Naam van een parameter:

- is betekenisvol: het komt overeen met wat de parameter betekent
- bestaat uit één of meer zelfstandige naamwoorden
- bestaat uit letters en cijfers: bevat geen spaties, komma's of andere 'rare' tekens ('_' uitgezonderd)
- is geschreven in lowerCaseCamel: begint met een kleine letter, elk volgend 'woord' begint met een hoofdletter
- bijvoorbeeld: `nrOfEggs`

Zie `http://google-styleguide.googlecode.com/svn/trunk/javaguide.html` voor een compleet overzicht van stijl- en naamgevingsafspraken.

We gaan nu een nieuwe methode toevoegen aan MyDodo.



Figuur 14: Stroomdiagram TURN180

1. Bekijk het stroomdiagram in figuur 14.
2. Open de code van de klasse `MyDodo` in de editor.
3. We gaan een nieuwe methode toevoegen aan deze klasse. Typ de volgende code over onderin het editorscherm. Doe dit wel voor de allerlaatste '}', anders valt de methode buiten de klasse en gaat de compiler klagen.

```
public void turn180( ) {
    turnRight( );
    turnRight( );
}
```

4. Compileer de code.
5. Roep jouw nieuwe methode **`void`** `turn180( )` aan door deze met de rechtermuisknop te selecteren en test of die werkt zoals verwacht.
6. Ga terug naar jouw code. Zet een beschrijving als commentaar boven de methode `turn180( )` waarin je kort uitlegt wat hij doet.
7. Je hebt weer een aanpassing gemaakt. Compileer opnieuw en herstel eventuele fouten.
8. Test opnieuw met de rechtermuisknop of de methode doet wat je verwacht.

5.9.6 Methode aanroepen in act (A)

Methode aanroepen in act
Door een methode aan te roepen in `act( )` zorg je ervoor dat deze wordt uitgevoerd als je op *Act* of op *Run* klikt. Het verschil tussen het klikken op de *Act* knop en het klikken op de *Run* knop is dat in het eerste geval de `act()` methode eenmalig wordt aangeroepen terwijl in het tweede geval dit herhaaldelijk gebeurt.

Om een methode vanuit de `act` aan te roepen moet je **`void`** `act( )` aanpassen:

```
public void act( ) {
    methodeNaam( );
}
```

1. Open in de code voor `MyDodo` in de editor en zoek de methode **`void`** `act( )`.
2. Haal de code tussen de accolades '{' en '}' weg.
3. Roep daar de methode **`void`** `turn180( )` aan. Kijk naar het voorbeeld hierboven hoe je dat precies moet doen.
4. Pas ook het commentaar boven `act( )` aan.
5. Compileer de code. Krijg je een foutmelding die je niet kunt oplossen? Kijk dan bij de hoofdstuk 5.10.
6. Voer de programma uit met *Run*.
7. Test de programma met *Act*. Doet het programma wat je verwacht?

5.10 Foutmeldingen

De compiler is erg kieskeurig. Soms maak je een foutje in de code of vergeet je iets. Dan zal de compiler daarover klagen. Het is handig als je enkele veel voorkomende klachten herkent, zodat je het probleem gemakkelijk kunt vinden en oplossen. Laten we er een paar bekijken.

1. Open de klasse `MyDodo` in de editor. Ga naar de code van de `act( )` methode. Verwijder de ';' achter `turnRight( );` . Sluit de editor en compileer. Welke foutmelding zie je onderin het scherm?
2. Herstel de ';' en compileer opnieuw. Dat zou nu probleemloos moeten gaan.
3. Test of het programma nog doet wat je verwacht.
4. Verander iets in de spelling van `turn180( )` en compileer. Welke foutmelding krijg je?
5. Herstel de spellingsfout, compileer en test opnieuw.
6. Verander `turn180( )` in `turn180(5)`.
7. Compileer. Welke foutmelding krijg je? Wat betekent de melding?
8. Verwijder de 5. Klik op de 'Compile' knop bovenin de editor. Dit compileert alleen deze ene klasse. Wat verschijnt er onderaan het scherm van de editor? Wat betekent dat?
9. Test of jouw programma nog doet wat je verwacht.

Syntaxfouten
Als je iets verkeerd intikt dat de compiler niet begrijpt, dan heet dat een *syntax error*. Een aantal veel voorkomende fouten zijn:

Gemaakte fout	Compiler foutmelding
ontbrekende ';' aan einde regel	';' expected
ontbrekende '( )' in header	'(' expected
ontbrekende '{' bij begin body	';' expected
ontbrekende '}' bij einde body	illegal start of expression
ontbrekende '( )' bij methode aanroep	not a statement
tikfout (let ook op hoofd/kleine letters)	cannot find symbol
verkeerde type parameter meegegeven	method cannot be applied to given types
geen parameter meegegeven	method cannot be applied to given types
verkeerd returntype	incompatible types: unexpected return value

Logische fouten
Daarnaast zijn er andere soorten programmeerfouten. Daarbij klaagt de compiler niet, maar doet het programma niet wat je verwacht had. Dat wordt een *logische fout* genoemd. Waar de fout dan zit is soms erg lastig te vinden. Het zoeken naar dat soort fouten heet *debuggen*. In het kader van 'voorkomen is beter dan genezen' zullen we in de volgende hoofdstuk aandacht besteden aan het gestructureerd aanpakken van het programmeren. Dat kan de kans op fouten aanzienlijk verkleinen.

6 Samenvatting

In deze opdracht heb je kennis gemaakt met Greenfoot. Je kunt nu:

- je weg door Greenfoot vinden;
- de methodes van objecten verkennen en uitvoeren;
- het verband tussen een algoritme, z'n stroomdiagram en code uitleggen;
- code van methodes vinden in de Greenfoot editor;
- stapsgewijs aanpassingen doorvoeren in de code;
- een methode toevoegen, compileren, uitvoeren en testen;

6.1 Opgaven Diagnostische toets

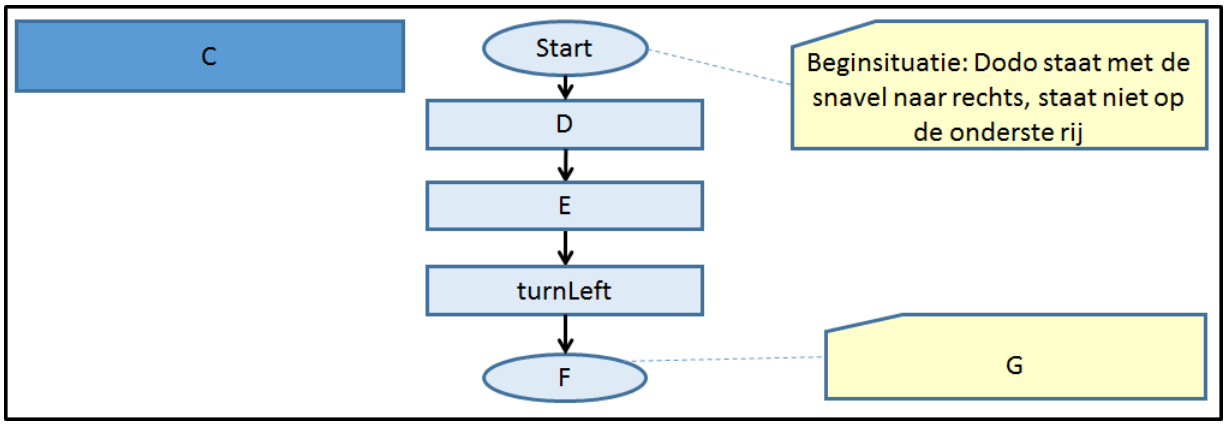
1. Leg in jouw eigen woorden uit, aan de hand van een voorbeeld, wat een resultaattype is.
2. Hieronder zie je een overzicht van de verschillende methoden. Vul deze tabel aan.

Methode	Resultaattype	Parametertype	Behoort tot klasse	Te gebruiken door object
`getNrOfEggsHatched()`		nvt	MyDodo	MyDodo
`canMove()`	boolean			
`jump()`				
`layEgg()`	void			
`turnLeft()`				Dodo en MyDodo
`canMove()`			MyDodo	
`getX()`				

3. Hieronder zie je een aantal namen voor methodes. Welke voldoet het beste aan de afspraken?
   - `walk_to_egg`
   - `eggWalker`
   - `WalkToEgg`
   - `walkToEgg`
   - `WALK_TO_EGG`
4. Hieronder zie je een aantal namen voor een parameter die het aantal eieren aangeeft. Welke voldoet het beste aan de afspraken?
   - `nr_of_eggs`
   - `EggCounter`
   - `nrOfEggs`
   - `eggs`
   - `NR_OF_EGGS`
5. Wat zijn de coördinaten van de linkeronderhoek?
6. Hieronder zie je de code van een nieuwe methode.

```
public A moveDown ( ){
    turnRight( );
    move( );
    B
}
```

Het bijbehorende stroomdiagram is:



Figuur 15: Stroomdiagram `moveDown()`

Wat hoort er te staan bij A, B, C, D, E, F en G in de code en het stroomdiagram?

6.2 Uitwerking diagnostische toets

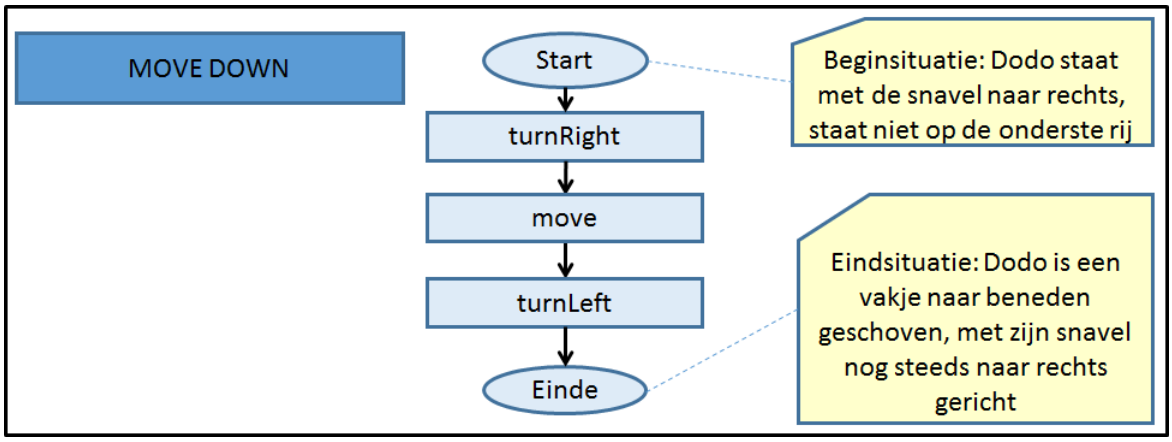
Hier volgt een voorbeeld uitwerking voor de vragen in de diagnostische toets.

1. Een resultaattype geeft aan welke soort waarde een methode kan teruggeven/opleveren. In het geval van een mutatormethode die 'iets doet' is dit een **`void`**. In het geval van een accessormethode die informatie oplevert is dit bijvoorbeeld een een **`boolean`** (die kan waar of onwaar zijn), of een **`int`** (geheel getal).

2.

Methode	Resultaattype	Parametertype	Behoort tot klasse	Te gebruiken door object
`getNrOfEggsHatched()`	int	nvt	MyDodo	MyDodo
`canMove()`	boolean	nvt	MyDodo	MyDodo
`jump()`	void	int	MyDodo	MyDodo
`layEgg()`	void	nvt	Dodo	Dodo en MyDodo
`turnLeft()`	void	nvt	Dodo	Dodo en MyDodo
`canMove()`	boolean	nvt	MyDodo	MyDodo
`getX()`	int	nvt	Actor	alle objecten

3. `walkToEgg`
4. `nrOfEggs` (toelichting: de mogelijkheid `eggs` is niet betekenisvol)
5. De coördinaten van de linkeronderhoek zijn: (0,11)
6. Zie figuur 16 voor het stroomdiagram.



Figuur 16: Stroomdiagram `moveDown( )`

7 Jouw werk opslaan

Je bent klaar met de eerste 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: `Opdr1_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 ...'.