Greenfoot — Java spelenderwijs

Prof. Kris Coolsaet

Universiteit Gent

Deze tekst bevat nota's bij de cursus '*Greenfoot — Java spelenderwijs*' ingericht door het Instituut voor permanente vorming in de wetenschappen van Universiteit Gent (IPVW) in de reeks '*Bijscholing Java voor leerkrachten informatica in het secundair onderwijs*'.

Bijkomend materiaal (broncodefragmenten, links, . . .) vind je op `http://inigem.ugent.be/greenfoot.html`. Voor elke opgave uit de tekst staan daar de volgende Greenfoot-scenario's ter beschikking (als ZIP-bestanden):

- Een *startproject*: een 'leeg' scenario met alle nodige afbeeldingen (en geluiden).
- Een project *zonder broncode*. Je kan dit uitproberen in Greenfoot om te zien wat de uiteindelijke bedoeling is van de opgave.
- Een *oplossing*, met broncode.

1. Krabbetje

Deze opgave is een aangepaste versie van het '*little crab*'-voorbeeld uit het Greenfoot-handboek, hoofdstukken 2 tot 4. Het eindresultaat is ongeveer hetzelfde, maar het uiteindelijke programma is anders gestructureerd. We gebruiken ook een ander stappenplan om tot het eindresultaat te komen.

De afbeeldingen (en geluiden) voor dit scenario komen van de handboekwebsite.

1.1 Beschrijving

1. De krab wordt bestuurd met de pijltjestoetsen. Met LINKS draait de krab vier graden naar links, met RECHTS vier graden naar rechts.
2. De kreeften verplaatsen zich 'willekeurig': in principe lopen ze rechtdoor maar er is 10% kans dat ze zich draaien, en dan over een willekeurige hoek tussen -45°en 45°.
3. Wanneer een kreeft de rand raakt, maakt hij een bocht van 130°.
4. De krab eet wormen. Zodra er 8 wormen zijn opgegeten stopt het programma en is het doel bereikt.
5. Ondertussen mag de krab echter geen enkele kreeft raken, want ook dan stopt het programma.

1.2 Stappenplan

Stap 1 Maak een wereldklasse *Strand* met de standaardinstellingen (600×400 cellen van grootte 1). Maak een klasse *Kreeft*. Laat de kreeft vooruit lopen (5 pixels per stap). Als de kreeft de rand van het strand bereikt, moet hij 130°draaien (ten opzichte van zijn huidige richting).

Schrijf een methode *tegenDeRand* die kijkt of de kreeft tegen de rand gelopen is door naar zijn X- en Y-coördinaten te kijken. In de plaats van de vaste getallen 600 en 400 te gebruiken, is het beter om aan de wereld zijn breedte en hoogte op te vragen.

Stap 2 Plaats drie kreeften op het strand wanneer het gecreëerd wordt — in de linker- en rechterbovenhoek en in de linkerbenedenhoek, telkens op 80 pixels van de randen. Verander de methode *act* van *Kreeft* zodat er één kans op 10 is dat de kreeft tijdens het lopen een bocht neemt tussen -45°en 45°.

De kreeften starten nu met een rotatiehoek van 0°. Als uitbreiding laat je ze starten in een willekeurige hoek.

Stap 3 Schrijf nu de klasse *Krab*. Net zoals de kreeft, verplaatst de krab zich 5 pixels per stap. De krab reageert op de linker en rechter pijltjestoetsen door een bocht te nemen van 4 graden naar links of naar rechts. Plaats een krab rechtsonder op het strand.

Er zijn twee methoden waarmee je het toetsenbord kan bevragen.

- *Greenfoot.isKeyDown* kijkt of een bepaalde toets *op dat moment* is ingedrukt
- *Greenfoot.getKey* vraagt de laatste toets op die door Greenfoot *nog niet is verwerkt*

In dit voorbeeld doet het er niet echt toe welke van de twee methoden je gebruikt, maar in andere situaties is het wel belangrijk om het verschil tussen beide te kennen.

Stap 4 Zorg dat het programma (de simulatie) eindigt zodra een kreeft de krab raakt.

Hier heb je de keuze in welke klasse je dit implementeert, in *Krab* of in *Kreeft*. Het programma wordt iets netter als je dit in *Kreeft* doet.

Om te kijken of twee *actors* overlappen, kan je zowel *isTouching* gebruiken als *getOneObjectAtOffset*. Probeer beide uit. Wat is het verschil?

Stap 5 Schrijf de klasse *Worm*. Plaats twaalf wormen op het strand, op willekeurige posities. Wanneer een krab een worm raakt, moet de worm van de wereld verdwijnen.

Doe dit laatste in de klasse *Worm* en niet in *Krab*, opnieuw omdat dit nettere code oplevert. Let er ook op dat wormen *onder* de kreeften (en krab) worden getekend en niet erboven.

Stap 6 Wanneer er acht wormen zijn gegeten, moet het programma stoppen.

De belangrijke vraag is hier wie we laten controleren hoeveel wormen er zijn opgegeten. Er zijn hiervoor twee mogelijkheden[1]: de *wereld* of de *krab*.

Omdat we de worm zichzelf hebben laten van de wereld verdwijnen, en de krab dus eigenlijk niet weet dat hij iets gegeten heeft, blijkt het gemakkelijkst te zijn om de wereld te laten kijken of er genoeg wormen zijn opgegeten. We doen dit in de methode *act* van *Strand* — inderdaad, ook bij de wereld wordt de *act*-methode bij elke simulatiepuls opgeroepen:

```
public class Strand {
    ...
    public void act() {
        int overblijvendeWormen = getObjects(Worm.class).size();
        if (overblijvendeWormen == 4) {
            ...
        }
    }
}
```

We profiteren hier van het bestaan van de methode *getObjects* waarmee je een lijst kunt opvragen van alle objecten van een gegeven type[2].

[1] Er is nog een derde optie: je houdt het aantal wormen bij als *klassenvariabele* van *Worm*. Dit werkt echter niet. Bij het indrukken van de 'Reset'-knop van Greenfoot worden de klassen niet opnieuw ingeladen en behouden de klassenvariabelen dus hun oude waarde.

[2] Voor de stijlpuristen onder ons: het opvragen van de volledige lijst van wormen is inderdaad nogal zwaar werk om alleen maar het *aantal* te weten te komen. Als alternatief kan je *removeObject* van *World* overschrijven in *Strand* met een methode die tegelijk het aantal verwijderde objecten telt.

1.3 Afwerken

Om het geheel af te werken, voegen we nog enkele details toe aan het programma.

Stap 7 Wanneer het programma stopt, geef je op de volgende manier aan wie er 'gewonnen' is: je verwijdert in het ene geval alle kreeften van de wereld, en in het andere geval, de krab.

Stap 8 Gebruik twee verschillende afbeeldingen voor de krab die elkaar bij elke simulatiepuls opvolgen.

Er zijn verschillende manieren om dit op te lossen:

- Gebruik een variabele die bij elke stap tussen twee waarden wisselt, bijvoorbeeld 0 en 1. Kijk telkens naar de waarde van die variabele om te weten welke afbeelding je moet instellen.

```
public void act() {
    if (nummerAfbeelding == 0) {
        setImage("crab.png");
        nummerAfbeelding = 1;
    } else {
        setImage("crab2.png");
        nummerAfbeelding = 0;
    }
    ...
}
```

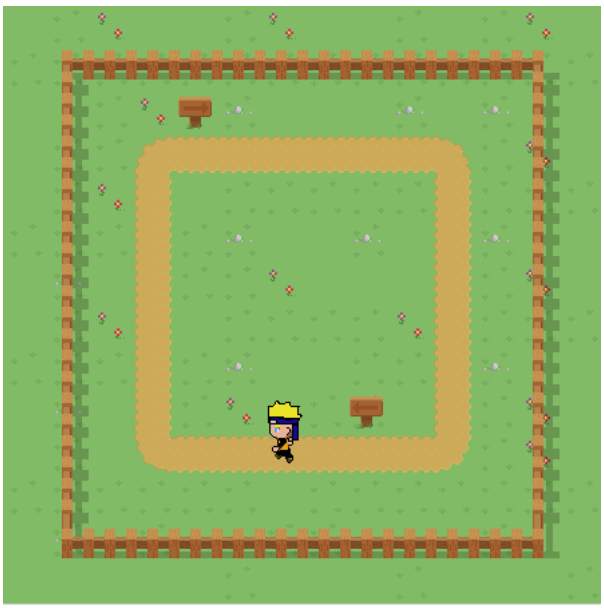
- Verhoog een variabele bij elke stap met 1. Kijk dan of die variabele even is of oneven om te beslissen welke afbeelding je gebruikt.
- Hou de afbeeldingen bij in twee variabelen en hou een derde variabele bij die afwisselend naar één van deze twee afbeeldingen wijst. Gebruik de derde variabele als *huidige* afbeelding. (Dit is de methode die in het Greenfoot-boek gebruikt wordt.)

En *last but not least*...

Stap 9 Voeg geluiden toe: wanneer een worm gegeten wordt, wanneer een kreeft de krab te pakken krijgt en wanneer de krab zijn wormen heeft gevangen.

2. Wandeling

In deze opgave laten we Naruto[1] rondwandelen in een tuin. Eerst in een eenvoudige rondgang maar later laten we hem een ingewikkelde weg beschrijven.



De afbeeldingen voor dit scenario komen van Open Pixels © Silveira Neto en van RPG Sprites Attack © Pete Maverick Fontanilla.

2.1 Beschrijving (eenvoudige versie)

1. Naruto loopt rond in de tuin en volgt daarbij een vierkant pad.

Om een wandelende Naruto te tekenen, gebruiken we 16 verschillende afbeeldingen, vier voor elke richting.

Rechts	0°	`naruto-0.png`	`naruto-1.png`	`naruto-2.png`	`naruto-3.png`
Onder	90°	`naruto-4.png`	`naruto-5.png`	`naruto-6.png`	`naruto-7.png`
Links	180°	`naruto-8.png`	`naruto-9.png`	`naruto-10.png`	`naruto-11.png`
Boven	270°	`naruto-12.png`	`naruto-13.png`	`naruto-14.png`	`naruto-15.png`

[1] Naruto Uzumaki is een figuur uit een Japanse manga- en animareeks.

2.2 Stappenplan

Stap 1 Maak een wereldklasse *Tuin* met een raster van 57×57 cellen van 8×8 pixels. Maak een klasse *Naruto*. Laat Naruto van links naar rechts lopen. Per stap beweeg je één cel naar rechts. Gebruik telkens de volgende van de vier afbeeldingen `naruto-0.png`, ..., `naruto-3.png`.

Vermijd om een ingewikkelde **if**/**else if**-structuur te gebruiken voor het bepalen van de naam van de afbeelding. Bouw de naam op als een concatenatie van strings en getallen:

```
public void act() {
    int index = ...     // 0, 1, 2 of 3
    setImage ("naruto-" + index + ".png");
    ...
}
```

De index 0, 1, 2 of 3 kan je bepalen aan de hand van de *leeftijd* van het object, geteld in aantal pulsen. Gebruik hier voor een variabele in je object die je bij elke puls met één verhoogt.

Stap 2 Laat Naruto van rechts naar links lopen, van boven naar onder en van onder naar boven. (Test alle mogelijkheden afzonderlijk.)

Er is hier maar één addertje onder het gras: de methode *setRotation* bepaalt niet alleen in welke richting Naruto verplaatst als je *move* oproept, maar draait tegelijk ook de afbeelding — en dit willen we niet. Los dit op door de afbeelding terug te draaien nádat hij is verplaatst.

Probeer net zoals bij de vorige stap een ingewikkelde **if**/**else if**-structuur te vermijden. De index van de afbeelding bereken je op de volgende manier:

- Deel de richting (rotatiehoek) door 90. (Dit geeft indices 0, 1, 2 of 3.)
- Vermenigvuldig dit met 4. (Dit geeft 0, 4, 8 of 12.)
- Tel daarbij de rest op van de *leeftijd* gedeeld door 4. (Dit telt 0, 1, 2 of 3 op bij het vorige resultaat.)

```
int index = richting / 90 % 4 * 4 + leeftijd % 4;
```

Stap 3 Plaats Naruto in de linkerbovenhoek van het pad als de wereld gecreëerd wordt. Laat Naruto rondlopen op het pad.

Gebruik de Greenfoot-interface om een Naruto-object te maken en plaats dit op de linkerbovenhoek van het pad. Gebruik dan 'Inspect' om te zien wat de X- en Y-coördinaten zijn van deze positie. Doe hetzelfde voor de andere hoeken. Zo hoef je niet te tellen of te meten op welke coördinaten de cellen van het pad zich bevinden.

De eenvoudigste manier om in het programma de richting van Naruto te bepalen, is een selectie van de volgende vorm:

> "Ga ik naar rechts en is mijn X-coördinaat gelijk aan ... dan verander ik mijn richting in 'onder'. Ga ik naar onder en is mijn Y-coördinaat gelijk aan ... dan verander ik mijn richting in 'links'. Enz..."

Uitdaging Bereik hetzelfde resultaat zonder **if**- of **switch**-opdrachten.

Tip: sla de grenswaarden voor X en Y op in tabellen en gebruik *richting* / 90 als index in die tabellen. Alternatief: bereken de richting uit de leeftijd.

2.3 Beschrijving (tweede versie)

1. Naruto loopt rond in de tuin en volgt daarbij een ingewikkeld pad. De exacte weg die hij moet volgen, heeft hij gememoriseerd en volgt geen bepaald patroon.

Met andere woorden, we zullen instructies van de vorm '4 cellen naar rechts, 2 cellen naar onder, 3 naar rechts, ...' hard coderen in het programma, maar zonder telkens opnieuw te moeten 'knippen en plakken'.

2.4 Stappenplan

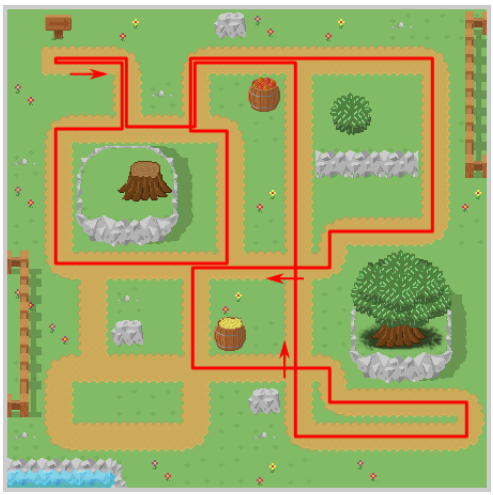
Stap 4 (Vertrek vanuit de broncode van stap 2.) Maak een nieuwe klasse *Instructies* aan (een gewone klasse, geen actor of wereld) met daarin een methode *volgendeRichting* () die een geheel getal retourneert. Laat deze methode voorlopig willekeurig 0, 90, 180 of 270 teruggeven. Gebruik deze methode in de klasse *Naruto* om telkens om de vier stappen een nieuwe richting voor *Naruto* te bepalen.

Vermijd het gebruik van klassenmethoden: maak in de constructor van *Naruto* één *Instructie*-object aan dat je dan in de *act*-methode raadpleegt.

```
public class Naruto extends Actor {

    private int richting;

    private int leeftijd;

    private Instructies instructies;

    public Naruto() {
        richting = 0;
        leeftijd = 0;
        instructies = new Instructies();
    }
```

```
    public void act() {
        if ( leeftijd  % 4 == 0) {
            richting = instructies.volgendeRichting();
        }
        ...
    }
    ...
}
```

Stap 5 Pas *Instructies* aan zodat Naruto het pad volgt uit onderstaande schets.



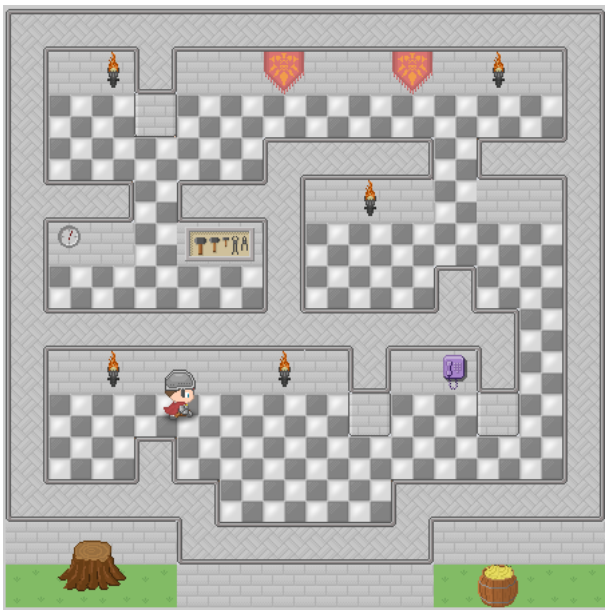
De bedoeling is dat je het pad codeert, bijvoorbeeld als een reeks gehele getallen, afwisselend een lengte en een richting (2, 0, 2, 90, 2, 180, 4, 90, . . .).

Gebruik in het programma constanten om de vier richtingen aan te geven, zowel voor de duidelijkheid als voor het gemak.

```
private static int R =   0;      // rechts
private static int O = 90;       // onder
private static int L = 180;      // links
private static int B = 270;      // boven
```

2.5 Variant: patrouille

Tot slot bespreken we nog kort een variant op hetzelfde thema.



1. Een ridder loopt rond in de zalen van het kasteel.
2. Hij volgt hierbij het muurbloempjesalgoritme: hij zorgt ervoor aan zijn linkerkant nooit het contact met de muur te verliezen .

Om de plaats van de muren te kennen, gebruik je een *kaart*, bijvoorbeeld in de vorm van een tabel (array) van strings:

```
private static String[] KAART = {
        "xxxxxxxxxxxxx",
        "x  x          x",
        "x             x",
        "x       xxx xxx",
        "xxx xxx       x",
        "x     x       x",
        "x     x   x   x",
        "xxxxxxxxxxxx x",
        "x        x  x x",
        "x             x",
        "x  x          x",
        "xxxxx    xxxxx",
        "    xxxxxx    "
};
```

Deze opgave heeft een bijkomende moeilijkheid die veroorzaakt wordt door het diepte-effect in de afbeelding van de zaal. Het is niet evident om de ridder *achter* de muren te laten verdwijnen tijdens zijn patrouille.

Niet zo: 

Maar zo: 

Uitdaging Los dit probleem op zonder in de broncode te spieken.

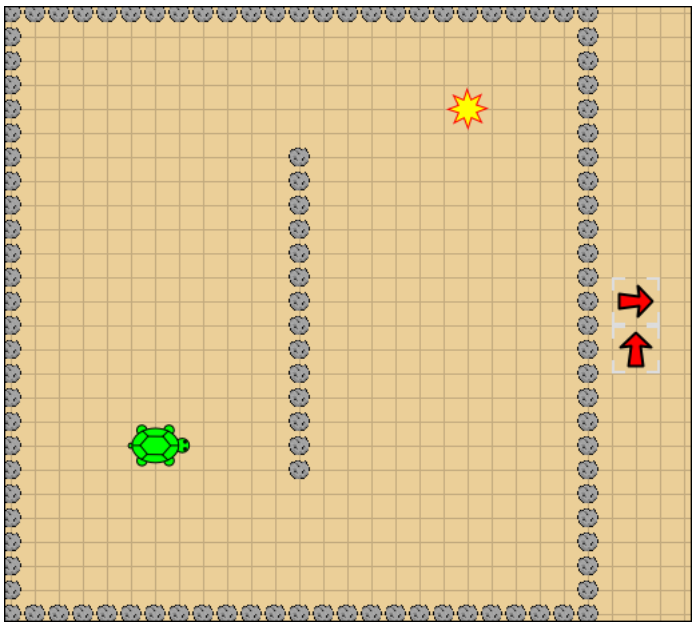
Tip: Kijk welke afbeeldingen er in het project worden gebruikt.

Andere programmeersystemen voor het maken van grafische spelletjes en simulaties met diepte-effect tekenen hun cellen steeds van boven naar onder (en van links naar rechts). Als je in dit geval de muren als afzonderlijke objecten implementeert, dan zal de ridder automatisch achter de muren verdwijnen.

In Greenfoot is het echter niet mogelijk om een dergelijke tekenvolgorde af te dwingen.

3. Kopoba

Dit project is gebaseerd op een bekende koeiendoolhofpuzzel maar dan in een sterk vereenvoudigde versie.



De afbeeldingen voor dit scenario komen van de handboekwebsite of zijn van de hand van de auteur.

3.1 Beschrijving

1. Een schildpad kruipt rond op het veld. Zijn acties worden bepaald door wat hij onderweg tegenkomt. (Hij wordt dus niet gestuurd door toetsen of de muis.)
2. Waar er stenen liggen op het veld kan de schildpad niet verder.
3. Komt de schildpad terecht op een pijl (op exact dezelfde positie), dan draait hij in de aangegeven richting en loopt verder.
4. De simulatie stopt wanneer de schildpad precies bovenop de ster komt te staan. De ster verdwijnt en de tekst 'Gelukt!' verschijnt in het midden van het veld.

Je voert dit scenario uit op een iets andere manier dan gewoonlijk:

Druk eerst op de 'Reset'-knop van de Greenfoot-interface. Druk nog niet op 'Run' maar verplaats eerst de pijlen naar de plaatsen waar je denkt dat ze moeten staan opdat de schildpad de ster zou kunnen bereiken. (Dit is het puzzelaspect van het scenario.) Druk dan op de 'Run'-knop en kijk of de schildpad haar doel bereikt.

Toegegeven, als puzzel is dit eerder iets voor je driejarige broertje, maar als je je fantasie laat werken, vind je zeker enkele interessante uitbreidingen van het programma die het de moeite waard maken. Je kan ook inspiratie opdoen bij de oorspronkelijke koeiendoolhofpuzzel.

3.2 Stappenplan

Stap 1 Maak een wereldklasse *Veld* met een raster van 57×51 cellen van 9×9 pixels. Maak een *Schildpad*-klasse en doe de schildpad over het veld kruipen. (Test elk van de vier richtingen!) Per stap kruip je één cel naar rechts en je hebt vier schildpadafbeeldingen die elkaar telkens moeten opvolgen.

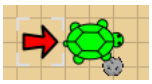
Het wisselen van de afbeeldingen gebeurt dus op dezelfde manier als bij Naruto in §2.2. In dit geval hebben we slechts 4 afbeeldingen in plaats van 16: dezelfde afbeeldingen worden gebruikt in elke richting, maar dan gedraaid.

Stap 2 Maak actor-klassen *Steen* en *Doel* (die voorlopig niets doen). Wanneer het programma opstart moeten het doel en de stenen op het veld worden geplaatst zoals in de figuur.

Stenen en doel zijn twee cellen hoog en breed en bevinden zich steeds op *even* X- en Y-coördinaten. Gebruik **for**-lussen om de stenen te plaatsen.

Stap 3 Zorg dat de schildpad niet verder kruipt wanneer ze een cel zou betreden die een steen bevat. In dat geval blijft de schildpad 'ter plaatse trappelen': ze verandert niet van positie maar de afbeeldingen blijven wisselen.

Dit is niet zo gemakkelijk als het lijkt. Je kan *isTouching* niet gebruiken omdat de schildpad wel *langs* een steen mag passeren, zoals in de figuur hiernaast.



Gebruik in de plaats *getOneObjectAtOffset* om een steen te vinden. Deze methode neemt als parameters de verschillen in X- en Y-coördinaten tussen de naburige en de huidige cel. Bij ons zijn die als volgt:

Richting	0°	90°	180°	270°
X-verschil	3	0	-3	0
Y-verschil	0	3	0	-3

Merk op dat we hier een afstand van 3 eenheden gebruiken en niet 4 zoals je wellicht zou verwachten. Dit is omdat *getOneObjectAtOffset* niet kijkt of het midden van de steen zich op 3 eenheden van het midden van de schildpad bevindt, maar wel of de steen de opgegeven positie (3 eenheden van het midden van de schildpad) *overlapt*!

Om het onderscheid tussen de vier richtingen te maken willen we hier, zoals eerder, liever een meervoudige selectie (**if**/**else if**-structuur) vermijden. Gebruik daarom '*richting* / 90' als index in een tabel (array) met X- en Y-verschillen. Je kan zelfs beide tabellen tot één tabel combineren.

Stap 4 Maak één klasse *Pijl* waarmee je vier verschillende pijlen kunt aanmaken, één voor elke richting. Zorg dat de schildpad van richting verandert wanneer ze precies op zo'n pijl terechtkomt.

Zoals in §1.2 kan je kiezen welke klasse (*Pijl* of *Schildpad*) de schildpad van richting doet veranderen indien nodig. Wij hebben een lichte voorkeur voor *Pijl* omdat dit het werk beter verdeelt tussen de verschillende klassen. (We zullen straks bij *Doel* dezelfde keuze maken.)

Opnieuw hebben we last van het feit dat *Greenfoot* in de eerste plaats overlappende objecten kan detecteren en niet direct naar hun coördinaten kijkt. Om dus te zien of het midden van de schildpad zich precies boven het midden van de pijl bevindt, volstaat het niet om *getOneObjectAtOffset* te gebruiken, maar moet je de coördinaten van het resultaat ook vergelijken met die van het origineel.

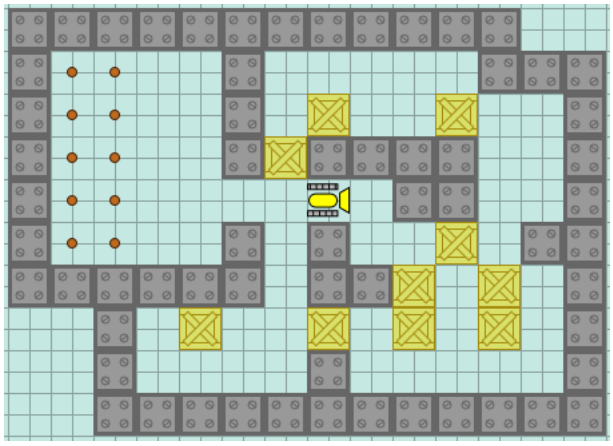
```
Actor actor = getOneObjectAtOffset(0,0,Schildpad.class);
if (actor != null && actor.getX() == getX() && actor.getY() == getY()) {
    ...
}
```

Stap 5 Werk het scenario nu verder volledig af: beëindig de simulatie als de schildpad zijn doel *precies* bereikt en toon het 'Gelukt!'-bericht.

Het 'Gelukt!'-bericht is gewoon een afbeelding. Zoek niet wanhopig naar een tekstmethode in de Greenfoot-documentatie.

4. Sokoban

Dit is een implementatie van de beroemde Sokoban-puzzel die in allerhande vormen op het Internet te vinden is.



De afbeeldingen voor dit scenario zijn van de hand van de auteur.

4.1 Beschrijving

1. Je kan een bulldozer van cel naar cel verplaatsen met de vier pijltjestoetsen. Vanzelfsprekend kan een bulldozer niet bewegen als er een muur in de weg staat.
2. De bulldozer kan een kist verschuiven door er vanuit de naburige cel tegen te duwen. Dit lukt niet als er één cel verder een muur of een andere kist staat. Twee kisten tegelijkertijd verduwen, kan dus niet.
3. De bedoeling van de puzzel is om alle kisten te verschuiven naar een doelcel (aangegeven door een rode stip).
4. Een kist krijgt een donkere kleur wanneer ze zich boven een doelstip bevindt.
5. Een kist in een doelpositie kan nog steeds verschoven worden (en terug een lichtere kleur krijgen).

4.2 Stappenplan

Stap 1 Maak een wereldklasse *Vloer* met een raster van 58×42 cellen van 8×8 pixels. Maak actor-klassen *Muur*, *Kist*, *Doel* en *Bulldozer* die voorlopig niets doen. Plaats bij het creëren van de wereld de juiste objecten op de juiste plaats.

Zoals in §2.5 kan je best een 'kaart' gebruiken om de wereld op te bouwen zonder al te veel knippen en plakken.

```
private static final String[] KAART = {
        "OOOOOOOOOOOO  ",
        "O..  O     OOO",
        "O..  O x  x  O",
        "O..  OxOOOO  O",
        "O..    s OO  O",
        "O..  O O  x OO",
        "OOOOOO OOx x O",
        "  O x  x x x O",
        "  O    O     O",
        "  OOOOOOOOOOOO"
};
```

Opdat je gemakkelijk een andere kaart in je programma zou kunnen instellen, mag je de grootte van het raster niet hard coderen, maar laat je ze afhangen van de afmetingen van de kaart.

Stap 2 Laat de bulldozer bewegen door de vier pijltjestoetsen. Bij het indrukken van een toets beweegt de bulldozer één cel naar rechts (= vier pulsen). Anders blijft hij stilstaan.

De twee verschillende afbeeldingen voor de bulldozer moeten elkaar afwisselen als hij beweegt — maar *niet* als hij stilstaat! Test ook wat er gebeurt wanneer de bulldozer over een doelstip rijdt.

Kijken of er een pijltjestoets is ingedrukt, en welke van de vier, kan je doen met een **if**/**else if**-structuur met vijf gevallen. Gebruik echter liever een **while**-lus die de vier mogelijke toetsnamen overloopt[1]. Dit wordt dan een lus met dubbele conditie, want we moeten ook voorzien dat er geen toets is ingedrukt.

Verzamel deze code in een methode *bepaalNieuweRichting*. Je hoeft deze methode niet op te roepen wanneer de bulldozer zich *tussen* twee cellen bevindt.

[1] Op het eerste zicht zou je ook een *Map* kunnen gebruiken die elke toetsnaam met een richting associeert. Dan moet je echter *Greenfoot.getKey* gebruiken i.p.v. *Greenfoot.isKeyDown* en dit heeft een ongewenst effect — zie §1.2.

Stap 3 Schrijf in de klasse *Bulldozer* een methode *buur* (*richting*) die de actor teruggeeft uit de naburige cel in de opgegeven richting (of **null** als de cell leeg is). Gebruikt deze methode in een nieuwe methode *kanBewegen* (*richting*) die kijkt of de bulldozer kan bewegen in de opgegeven richting. De bulldozer mag niet door muren of kisten rijden.

(De methode *buur* zal later nog nuttig blijken als we kisten gaan verschuiven.)

Gebruik *getOneObjectAtOffset* om de naburige actor te vinden, zoals in §3.2.

Stap 4 Schrijf gelijkaardige methoden *buur* (*richting*) en *kanBewegen* (*richting*) voor de klasse *Kist*. Gebruik die om toe te laten dat de bulldozer toch in een richting met een kist kan rijden als hij die kist in dezelfde richting kan verschuiven. (Voorlopig verschuif je de kist echter nog niet.)

Je merkt dat je heel wat uit de broncode van *Bulldozer* rechtstreeks kan overnemen in *Kist*. Knippen en plakken getuigt echter niet van een goed softwareontwerp. Vandaar ...

Stap 5 Schrijf een gemeenschappelijke bovenklasse *Beweger* voor *Bulldozer* en *Kist* en verplaats de methode *buur* (en al wat ervoor nodig is) naar die klasse. Test of je project nog werkt.

Stap 6 Voeg een methode *verschuif* (*richting*) toe aan *Kist* en gebruik die methode bij *Bulldozer* om een naburige kist te verplaatsen.

Als je wil vermijden om twee keer de buur van de bulldozer op te vragen — één keer wanneer je bepaalt of de bulldozer kan bewegen en één keer om de kist te vinden die moet verschuiven — kan je een verwijzing naar het kist opslaan in een veld van *Bulldozer*.

Let op dat de kist bij *elke* simulatiepuls moet opschuiven en niet enkel wanneer de bulldozer zich precies in een cel bevindt. De controle of de kist kan verschuiven, hoeft echter enkel maar één keer om de vier pulsen. Let ook op dat de afbeelding van de kist niet draait wanneer je ze naar boven of naar onder schuift.

Uitdaging Op het eerste zicht is het scenario nu afgewerkt, op het verkleuren van de kisten na. Er is echter één situatie die we niet hebben voorzien. Kan je bedenken welke fout er nog in het programma zit zonder verder te lezen?

Stap 7 Pas de kleur (de afbeelding) van een kist aan wanneer ze al dan niet met een doelstip overlapt.

Wat we niet hebben voorzien, is dat een kist zich *bovenop* een doel kan bevinden. Wanneer we dan met *getOneObjectOffset* vragen welk object zich op die positie bevindt, is het niet te voorzien of Greenfoot de doelstip of de kist zal teruggeven. Het is daarom beter om de lijst van *alle* objecten op die positie op te vragen, ze te overlopen en alle *Doel*-objecten erin te negeren.

Stap 8 Pas de methode *buur* aan zoals hierboven geschetst. Ze mag alleen een buur retourneren die *geen* doelstip is.

En nu zie je waarom het goed was om een bovenklasse *Beweger* te introduceren. Nu is er maar één plaats waar we deze aanpassing moeten doen.