# Greenfoot — Bijkomende projecten

Prof. Kris Coolsaet

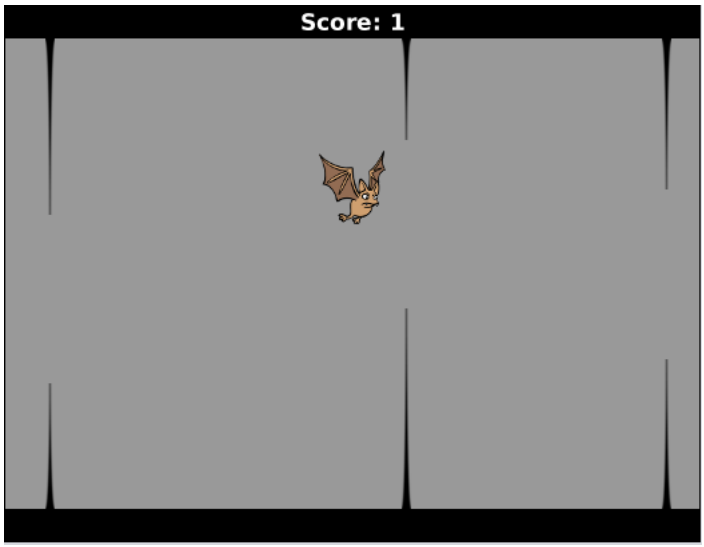
Universiteit Gent

Deze tekst bevat bijkomende projecten ter aanvulling van de cursus '*Greenfoot — Java spelenderwijs*'. Deze scenario's werden oorspronkelijk opgesteld door studenten van de lerarenopleiding Informatica van Universiteit Gent, als opdracht bij de lessen *Vakdidactiek Informatica 2*. Daarna zijn ze nog enigszins 'opgepoetst' door de lesgevers.

Broncode en ander materiaal dat je kan gebruiken bij deze projecten, vind je op `http://inigem.ugent.be/greenfoot.html`.

# 1. Flappy Bat

Flappy Bat is een aangepast versie van het spel Flappy Bird. In het spel moet je met de vleermuis zo lang mogelijk blijven vliegen zonder de grond, plafond of uitsteeksels aan te raken.




De oorspronkelijke auteur van dit scenario is *Matthias Druwé*. De gebruikte afbeeldingen zijn van zijn hand of komen uit Scratch.

## 1.1 Beschrijving

1. De vleermuis wordt bestuurd met de spatiebalk. Telkens je de spatiebalk indrukt, gaat hij een stukje omhoog.
2. Druk je niet op de spatiebalk, dan valt de vleermuis naar beneden, met een snelheid die bij elke tik van de klok groter wordt.
3. Stalagmieten en stalactieten verplaatsen zich van rechts naar links over het scherm en hebben een willekeurige hoogte. (De opening ertussen is wel steeds even groot.)
4. Telkens wanneer de vleermuis erin slaagt door een opening te vliegen, krijg je een punt. De score zie je bovenaan.
5. Het spel eindigt wanneer je een stalagmiet, stalactiet, de grond of het plafond raakt.

## 1.2 Stappenplan

**Stap 1** Maak een klasse *Grot* afgeleid van *World*. Cellen zijn 1 pixel groot en de wereld is 520 pixels breed en 400 pixels hoog. Stel ook de achtergrondafbeelding in.

Maak een klasse *Vleermuis* en plaats een object van die klasse in het midden van de grot.

Zorg dat de vleermuis 15 pixels naar omhoog gaat telkens wanneer er op de spatiebalk gedrukt wordt. (Dit doe je in de *act*-methode van *Vleermuis*.)

Gebruik *Greenfoot.getKey* en niet *Greenfoot.isKeyDown*. Zo vermijd je dat dezelfde toetsaanslag meer dan één keer wordt verwerkt. De meest compacte manier om te kijken of de spatiebalk was ingedrukt, is de volgende:

```
if ("space".equals(Greenfoot.getKey())) {
    ...
}
```

Merk op dat je de argumenten van *equals* niet in de andere volgorde mag zetten, want *getKey* geeft **null** terug wanneer er geen toets werd ingedrukt.

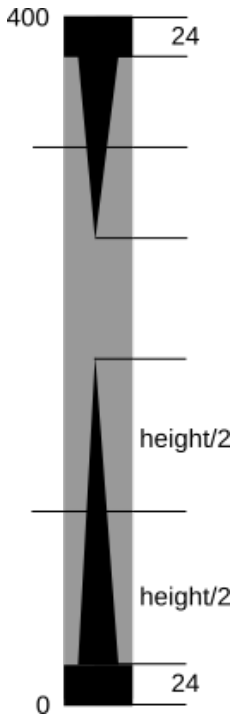
**Stap 2** Zorg ervoor dat de vleermuis naar beneden valt als er niet op de spatiebalk werd gedrukt. Het vallen moet altijd sneller gaan: eerst valt de vleermuis 5 pixels, bij de volgende tijdstap 10 pixels, de volgende keer 15, enz.

Doe de vleermuis *flapperen* door de tweede vleermuisafbeelding te gebruiken wanneer er op de spatiebalk werd gedrukt, en de eerste in alle andere gevallen.

Vergeet de snelheid niet terug op 0 te zetten wanneer de vleermuis terug naar boven vliegt.

Nu de vleermuis correct beweegt, kunnen we de stalagmieten en stalactieten introduceren. Er zijn 24 verschillende afbeeldingen, voor stalagmieten en stalactieten van verschillende hoogte. Ze komen telkens in paren die samen moeten gebruikt worden. De namen van de afbeelding zijn `stekel1onder.png`, `stekel2onder.png`, ..., `stekel12onder.png`, voor de 12 stalagmieten (die onderaan de grot staan), en `stekel1boven.png`, `stekel2boven.png`, ..., `stekel12boven.png`, voor de stalagmieten (die bovenaan de grot hangen).

De som van de hoogtes van een corresponderend paar (met hetzelfde volgnummer) is steeds dezelfde, zodat je telkens gelijke tussenruimten krijgt.




Het correct positioneren van de stekels vraagt een beetje rekenwerk, zoals geïllustreerd in de afbeelding hiernaast. De positie van een afbeelding bij Greenfoot wordt immers steeds bepaald door haar *middelpunt*. Gelukkig kan je met *getHeight*() de grootte van een figuur opvragen.

**Stap 3** Maak twee nieuwe actor-klassen aan: *Onderstekel* en *Bovenstekel*. Beide klassen moeten een constructor hebben met één parameter — het volgnummer *nr* (1–12) dat de afbeelding (en de hoogte) bepaalt van die stekel.

Schrijf in beide klassen ook een methode die teruggeeft wat de Y-coördinaat is van hun middelpunt. De grot kan deze informatie dan gebruiken om de stekels te positioneren.

Plaats een overeenkomstig paar stekels (met willekeurig volgnummer) aan de rechterkant van de wereld.

Zorg dat beide stekels langzaam naar links verschuiven terwijl het programma loopt (3 pixels per stap).

**Tips.** Met *Greenfoot.getRandomNumber*(12) krijg je een willekeurig getal in het bereik 0…11. Tel daar 1 bij op om een willekeurig volgnummer te vinden voor je stekel. Het verschuiven van de stekels doe je in hun *act*-methode.

**Stap 4** Controleer dat de vleermuis de grond of het plafond niet raakt, of één van de stekels. Is dat wel zo, dan gaat de vleermuis dood.

Om aan te geven dat de vleermuis dood is, neem je de tweede afbeelding, draai je de vleermuis 180°, en laat je hem vallen tot hij de onderkant van het venster raakt. Je luistert niet meer naar de spatietoets. Wanneer de dode vleermuis de onderkant raakt, stop je het programma met *Greenfoot.stop*().

Om in de *act*-methode onderscheid te kunnen maken tussen een levende en een dode vleermuis, kan je in de klasse *Vleermuis* een logisch (= Booleaans) veld *dood* bijhouden dat je dan in een **if**-opdracht bekijkt[1]:

[1] In paragraaf §1.3 geven we hiervoor een alternatief.

```
public void act() {
    if (dood) {
                ...
    } else {
            ...
    }
}
```

Om na te gaan of een vleermuis een stekel raakt, gebruik je *isTouching*[2]. Kijk naar de Y-positie van de vleermuis om te weten of hij de grond of het plafond raakt. Vergeet niet te compenseren voor de halve hoogte van de vleermuisafbeelding. Splits de methode *act* in afzonderlijke functies en procedures *stijg*, *val*, *sterf*, *isGeraakt*, ..., ze wordt anders te lang.

**Stap 5** Maak een klasse *Punten* waarmee je de score bovenaan het scherm weergeeft.

Verhoog de punten telkens wanneer de onderste stekel links van de vleermuis passeert.

De 'afbeelding' van het *Punten*-object komt deze keer niet uit een bestand, maar bestaat uit de tekst "Score:" en het puntental. Je kan dit op de volgende manier instellen:

```
import java.awt.Color;
 ...

public void toonPunten() {
    setImage( new GreenfootImage("Score:  " + totaal, 20,
                                 Color.WHITE, Color.BLACK) );
}
```

Het is de verantwoordelijkheid van de *onderste* stekel om het puntental aan te passen. Dit gebeurt wanneer de stekel *voor de eerste keer* in de linkerhelft van het scherm terecht komt. Om het puntental te kunnen aanpassen, moet de stekel het *Punten*-object kunnen bereiken. Geef een referentie naar het *Punten*-object mee als (bijkomende) parameter van de constructor van *Onderstekel*.

[2] Deze methode is niet perfect: ze kijkt of de omhullende *rechthoeken* elkaar overlappen, en niet de figuren zelf. Het is echter niet eenvoudig om dit op te lossen.

**Stap 6** Wanneer de stekels de linkerkant van het scherm bereiken, blijven ze daar steken[3]. Zorg daarom dat de stekels zichzelf van de wereld verwijderen zodra hun X-coördinaat 0 is geworden.

**Stap 7** Tot nog toe gebruikten we slechts één paar stekels. Zorg er nu voor dat er op geregelde tijdstippen nieuwe stekelparen worden bijgemaakt aan de rechterkant van het scherm. Doe dit in de *act* van *Grot*.

Het is leuker wanneer de afstanden tussen opeenvolgende stekels niet altijd exact dezelfde zijn. Gebruik willekeurige getallen om daar wat variatie in te brengen.

**Uitdaging** Pas het programma aan zodat de snelheid telkens lichtjes stijgt na elke 5 punten die je hebt behaald.

## 1.3 Een opgepoetste versie

Hoewel het programma nu naar behoren werkt, zijn er toch nog enkele plaatsen waar we de Java-code kunnen oppoetsen en wat meer objectgeoriënteerd maken. Dit zal het later gemakkelijker maken om het programma uit te breiden.

**Stap 8** De implementatie van *Onderstekel* en *Bovenstekel* is heel gelijkaardig. Introduceer daarom een bovenklasse *Stekel* voor die twee klassen en reorganiseer de implementatie zodat de gemeenschappelijke code in de bovenklasse terechtkomt.

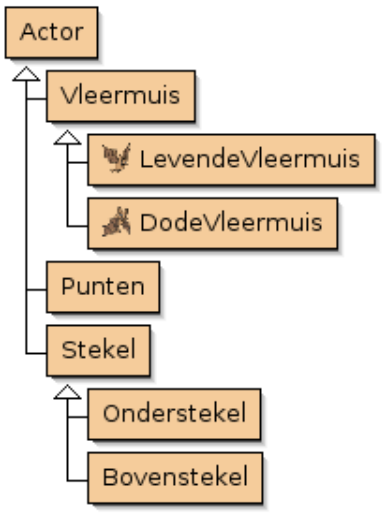
De methode *act* van *Onderstekel* doet hetzelfde als deze van *Bovenstekel* maar voegt daar bovendien de puntentelling aan toe. Je kan dit implementeren door het gemeenschappelijk gedeelte in de *act*-methode van *Stekel* te plaatsen, de methode *act* uit *Bovenstekel* te schrappen en in *Onderstekel* de volgende implementatie te gebruiken:

```
public void act() {
  super.act(); // voer de act uit van Stekel
   ...         // verhoog de score indien nodig
}
```

[3] De standaardinstelling van een Greenfootwereld laat niet toe dat actoren zich buiten het scherm verplaatsen. Je kan die instelling echter wel veranderen.

Je kan nu ook de botsingsdetectie een klein beetje vereenvoudigen: je hoeft met *isTouching* niet meer afzonderlijk op beide soorten stekels te testen, maar enkel op de *Stekel*-klasse.

In de huidige vorm van het programma, gebruiken we een logische variabele *dood* om het onderscheid te maken tussen een levende en een dode vleermuis. Je kan dit echter ook doen door twee verschillende klassen te gebruiken:

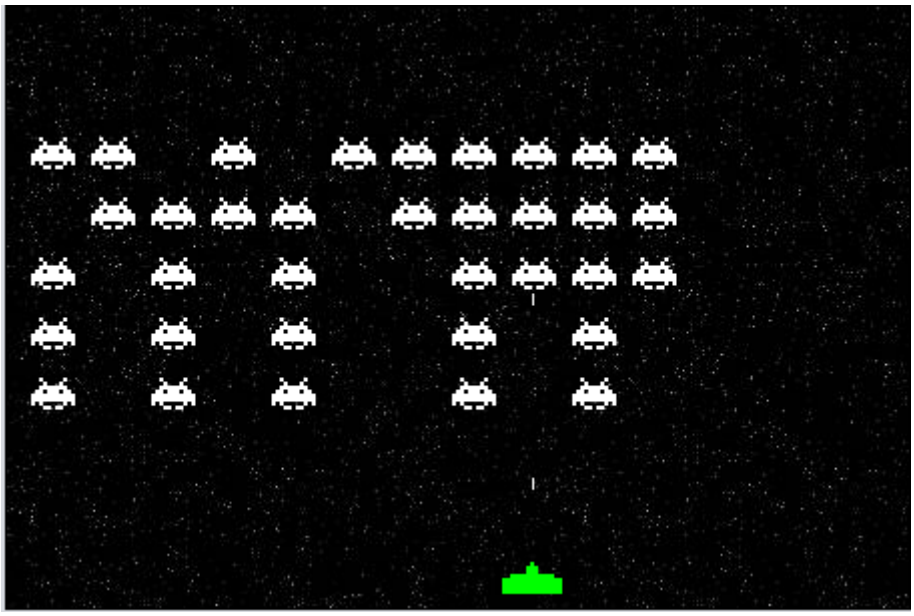



**Stap 9** Maak nieuwe klassen *DodeVleermuis* en *LevendeVleermuis* aan met een gemeenschappelijke bovenklasse *Vleermuis*. Herschik de code.

Op het moment dat de (levende) vleermuis wordt geraakt, laat je het *LevendeVleermuis*-object een nieuw *DodeVleermuis*-object op de wereld plaatsen en zichzelf van de wereld verwijderen.

# 2. Space Invaders

Deze oefening is een eenvoudige versie van het klassieke spel Space Invaders, één van de eerste videospellen ooit op de markt gebracht (in 1978). De computers waren toen zo traag dat de ontwerper speciale elektronica nodig had om het programma voldoende snel te krijgen. Gelukkig krijgen wij hulp van *Greenfoot*.



De oorspronkelijke auteur van dit scenario is *Tom Neutens*. De gebruikte afbeeldingen zijn kopieën van de originele *bitmaps* uit 1978.

**Opgelet!** We hebben deze opgave uitgewerkt met behulp van Stride, de nieuwe *frame based editor* van Greenfoot. Stride houdt het midden tussen een grafische en een tekstuele programmeertaal. Een Java-programmeur zal geen problemen hebben met de notatie die Stride gebruikt. De belangrijkste verschillen met de programmeertaal Java zijn wellicht de notatie

**for each** (**int** i **in** 1..10)

voor een lus met teller, het sleutelwoord **var** waarmee je een lokale variabele introduceert, de ⇐ voor toewijzingen en het ontbreken van kommapunten en accolades.

## 2.1 Beschrijving

1. Het groene 'kanon' onderaan bestuur je met de linker en rechter pijltjestoetsen. Het kanon schiet een laserkogel af als je op de spatiebalk drukt.
2. Er zijn vijf rijen van elf ruimtewezens. Zij bewegen zich achtereenvolgens van links naar rechts en van rechts naar links op het scherm. Ze beginnen bovenaan, en telkens wanneer ze van richting veranderen, komen ze ook een stukje dichterbij de onderrand van het scherm.
3. Je wint het spel als je alle ruimtewezens kan neerschieten. Je verliest wanneer je de ruimtewezens te dicht bij de onderrand laat komen.

We houden het hier met opzet eenvoudig. Dit scenario biedt echter heel wat mogelijkheden tot uitbreidingen die het spel dichter bij het origineel brengen.

## 2.2 Stappenplan

**Stap 1** Maak een wereldklasse *Ruimte* van 450 × 300 cellen van 1 pixel groot. (Gebruik als achtergrond de standaardafbeelding *space1.jpg* uit de Greenfoot-bibliotheek.)

Maak een klasse *Kanon* (afbeelding: *gun.png*) en plaats een object van die klasse midden onderaan het scherm, op 15 pixels van de onderrand. Zorg dat je dit kanon naar links en naar rechts kan verplaatsen met de pijltjestoetsen — 2 pixels per keer.

Gebruik *Greenfoot.isKeyDown* om te achterhalen welke toets was ingedrukt. Gebruik *move*(..) om het kanon te verplaatsen. Een positieve parameter beweegt het kanon naar rechts, een negatieve naar links.

**Stap 2** Maak een klasse *Ruimtewezen* (afbeelding: *alien-0.png*) en plaats 55 objecten van die klasse op de wereld, in 5 rijen van 11.

Het ruimtewezen links boven heeft coördinaten (15,25). De wezens staan telkens 30 pixels van elkaar.

**Stap 3** Laat de ruimtewezens bewegen van links naar rechts. (Zie beschrijving hieronder.)

We willen dat de ruimtewezens op een schokkerige manier voortbewegen — zoals in het oude videospel. Om dit te bekomen verplaats je een wezen niet telkens wanneer Greenfoot een puls geeft, maar laat je het slechts *één keer om de 12 pulsen* een grotere beweging maken (4 pixels).

Stop het getal 12 in een variabele met de naam *pulsenPerStap*. Je kan dan gemakkelijk als latere uitbreiding de snelheid aanpassen naarmate het spel vordert.

Daarnaast moet je na elke stap afwisselen tussen de twee verschillende afbeeldingen *alien-0.png* en *alien-1.png*.

Omdat we ook straks nog in stappen zullen tellen in plaats van in pulsen, is het nuttig om een afzonderlijke methode *stap* te voorzien die je oproept vanuit *act*. Zoals *act* telkens bij elke nieuwe puls wordt opgeroepen, zorg je dat *stap* telkens bij elke nieuwe stap wordt uitgevoerd.

```
private int pulsenPerStap ⇐ 12
private int pulsen ⇐ 0
private int stappen ⇐ 0

Dit wordt voor elke puls herhaald:
public void act()
    pulsen ⇐ pulsen + 1
    if (pulsen == pulsenPerStap)
        pulsen ⇐ 0
        stappen ⇐ stappen + 1
        stap ()

Dit wordt voor elke stap herhaald:
public void stap()
    ... hier komt jouw code ...
```

**Stap 4** Zorg dat de wezens telkens na 30 stappen ($\neq$ pulsen!) van richting veranderen en op hetzelfde moment 8 pixels naar beneden zakken.

Je hebt misschien gemerkt dat *rotate*(180) geen goeie manier is om het ruimtewezen van richting te laten veranderen. (Waarom niet?) Gebruik in de plaats *move*(..) met een negatieve afstand als argument (-4 pixels).

Er zijn twee manieren om bij elke stap te bepalen of de afstand negatief is of positief.

- Je kan dit zien aan het aantal keer dat het wezen reeds 30 stappen heeft gezet en of dit even is, of oneven.
- Je kan een variabele *richting* bijhouden die je van teken verandert telkens wanneer er opnieuw 30 stappen zijn gezet. Is *richting* één, dan beweegt het wezen naar rechts, is *richting* nul, dan beweegt het wezen naar links.

```
private int richting ⇐ 1

public void stap ()
    move (4∗richting)
    if (stappen % 30 == 0)
        richting ⇐ − richting
        ...
    ...
```

Nu wisselt de inhoud van de variabele tussen de waarden 1 en $-1$. Je zou ze ook kunnen laten wisselen tussen 4 en $-4$, de afstand die je bij de verplaatsing gebruikt.

**Stap 5** Voeg een klasse *Kogel* toe met als afbeelding *bullet.png*. De kogel verplaatst zichzelf telkens naar boven met een snelheid van 4 pixels per puls. Als de kogel de bovenrand van de wereld raakt, verdwijnt hij uit de wereld.

Zorg ervoor dat je een kogel afvuurt telkens wanneer je op de spatiebalk duwt.

Vergeet niet de kogel in de juiste richting te draaien vooraleer je hem gebruikt (m.a.w., in de constructor van *Kogel*).

Het is het kanon die een nieuwe kogel aanmaakt en op de wereld plaatst, met behulp van

```
getWorld().addObject(kogel, x, y)
```

De kogel moet starten op de huidige coördinaten van het kanon. Gebruik *getX*() en *getY*() om deze coördinaten op te vragen.

De kogel kan zichzelf van de wereld verwijderen met de volgende opdracht:

```
getWorld().removeObject(this)
```

Hij doet dit wanneer hij de 'einde' van de wereld bereikt, maar straks ook wanneer hij een ruimtewezen heeft geraakt.

Wanneer je het resultaat van stap 5 uitprobeert, zal je merken dat het kanon bijna doorlopend kogels afvuurt zolang je op de spatiebalk drukt. Dit is helemaal niet de bedoeling:

**Stap 6** Zorg ervoor dat het minstens 20 pulsen duurt vooraleer een volgende kogel kan worden afgevuurd.

Hou hiervoor (in de klasse *Kanon*) een variabele *wachttijd* bij die je bij elk nieuw schot terug laat aftellen vanaf 20.

**Stap 7** Een ruimtewezen dat door een kogel wordt geraakt, moet (samen met de kogel) van de wereld verdwijnen. Laat dan ook het geluid van een explosie horen (bestandsnaam: *explosion.wav*).

Het ligt misschien voor de hand om het *Ruimtewezen*-object te laten controleren of het niet door een *Kogel*-object wordt geraakt, maar het kan ook omgekeerd: de kogel kan nagaan of hij een ruimtewezen raakt. Dit laatste blijkt in de praktijk de netste programmacode op te leveren.

Om te kijken of een kogel met een ruimtewezen overlapt, kan je de volgende code gebruiken:

```
var Actor actor ⇐ getOneIntersectingObject(Ruimtewezen.class)
if (actor != null)
    Speel een geluid af en verwijder 'actor' van de wereld
```

**Stap 8** In de plaats van het ruimtewezen gewoon van het scherm te laten verdwijnen, willen we eerst de explosie laten zien. Voeg daarom de opdracht *actor.setImage*(`"explosion.png"`) toe aan de **if**-opdracht hierboven en probeer het resultaat uit. Wat zie je? Wat is hiervoor de reden, denk je?

Omdat we het ruimtewezen onmiddellijk van de wereld laten verdwijnen, heeft het veranderen van zijn afbeelding weinig nut. We zouden op één of andere manier een vertraging moeten kunnen inbouwen. Er zijn opnieuw verschillende manieren om dit voor elkaar te krijgen.

We kunnen bijvoorbeeld in plaats van het ruimtewezen onmiddellijk van de wereld te doen verdwijnen, op één of ander manier bij het ruimtewezen registreren dat het is geraakt en het ook een tellertje laten bijhouden van hoe lang het nog kan blijven 'leven'. Hiervoor heb je enkele bijkomende velden nodig in de klasse *Ruimtewezen* en enkele extra selecties (**if**-opdrachten) in de methodes *act* en/of *stap* van die klasse. Je moet ook opletten dat de explosie niet nogmaals kan geraakt worden door een tweede kogel en dat de ontploffing zich niet meer verplaatst[1].

Wij kiezen voor een andere oplossing: we stellen de explosie zelf voor door een object van een nieuwe klasse *Ontploffing*. Dit is een heel eenvoudig object dat zichzelf na een zeker tijd (6 pulsen) van de wereld verwijdert.

**Stap 9** Implementeer deze klasse *Ontploffing* (afbeelding: *explosion.png*).

Vervang een ruimtewezen dat werd geraakt door een *Ontploffing*-object met dezelfde coördinaten.

**Stap 10** Het spelt eindigt wanneer alle ruimtewezens zijn vernietigd. Laat in dit geval een vrolijk geluid horen (*fanfare.wav*) en stop Greenfoot.

Als één van de ruimtewezens te laag bij de onderrand komt, dan ben je verloren. Speel in dit geval een treurig muziekje (*game-over.wav*) en stop het programma.

Het is eenvoudig voor een ruimtewezen om te detecteren of het dicht bij de onderrand komt. Het is moeilijker te bepalen wanneer alle ruimtewezens zijn vernietigd. Opnieuw zijn er verschillende technieken die je hierbij kunt gebruiken.

Je kan de wereld bijvoorbeeld telkens opnieuw laten tellen hoeveel ruimtewezens er nog over zijn. Die doe je in de *act*-methode van de wereld zelf. Bijvoorbeeld op de volgende manier:

[1] De fysica leert ons dat de ontploffing zich zal verplaatsen in de richting waarin het ruimtewezen zich op dat moment bewoog. Maar ze zal zeker niet van richting veranderen na 30 stappen.

```
public void act()
    if (getObjects(Ruimtewezen.class).isEmpty())
        ... Speel de fanfare en stop Greenfoot ...
```

Het enige nadeel van deze manier van werken, is dat de laatste explosie niet de kans krijgt om zich af te werken.

We kiezen er daarom voor de methode *removeObject* van *World* te overschrijven in *Ruimte*. De nieuwe methode zal niet alleen het object van de wereld verwijderen, maar tegelijk ook tellen hoeveel keer er dit een *Ontploffing*-object was. Na 55 keer is het spel ten einde.

Om te weten of de *actor* die verdwijnt een ontploffing is, vragen we op tot welke klasse hij behoort en kijken of dit de klasse *Ontploffing* is[2]:

```
public void removeObject (Actor actor)
    if (actor.getClass() == Ontploffing.class)
        aantalOntploffingen ⇐ aantalOntploffingen − 1
        if (aantalOntploffingen == 55)
            ... Het spel is ten einde ...
    super.removeObject(actor)
```

(De laatste opdracht roept de methode *removeObject* van *World* op, zodat het object ook effectief van de wereld verdwijnt.)

Voor wie dit allemaal te moeilijk lijkt, is er nog een derde alternatief: we kunnen het aantal ontploffingen ook bijhouden als *klassenvariabele* van *Ontploffing*.

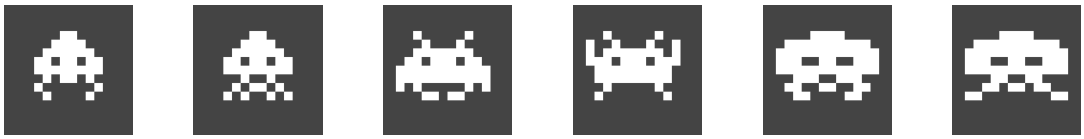
Dit werkt echter niet zonder meer. Bij het indrukken van de 'Reset'-knop van Greenfoot worden de klassen namelijk niet opnieuw ingeladen en behouden de klassenvariabelen dus hun oude waarde. We moeten daarom de constructor van *Ruimte* die klassenvariabele opnieuw laten initialiseren. (Deze constructor wordt namelijk wel opnieuw uitgevoerd bij elke 'Reset'.)

[2] In Java zouden we hier schrijven '**if** (*actor* **instanceof** *Ontploffing*)'. Stride kent deze notatie echter niet

## 2.3 Uitbreidingen

We hebben nu een werkend spelletje, maar zoals je al snel zult merken, zeer uitdagend is het niet. We geven hieronder enkele suggesties om het programma verder uit te breiden zodat het leuker wordt om te spelen, en ook meer begint te lijken op het origineel[3].

> De ruimtewezens hoeven er niet allemaal hetzelfde uit te zien.



Er zijn 3 soorten wezens. Van de eerste soort is er bovenaan één volledige rij, van de andere telkens twee rijen.

> Hoe minder ruimtewezens er zijn, hoe sneller ze bewegen.

Zoals we al in stap 3 hebben aangegeven, kan je dit doen door de variabele *pulsenPerStap* te verkleinen naargelang er minder ruimtewezens overblijven.

Omdat dit aantal pulsen voor elk *Ruimtewezen*-object hetzelfde is, gebruik je hiervoor best een klassenvariabele. Denk eerst goed na over welk object (welke klasse) je deze variabele zal laten aanpassen.

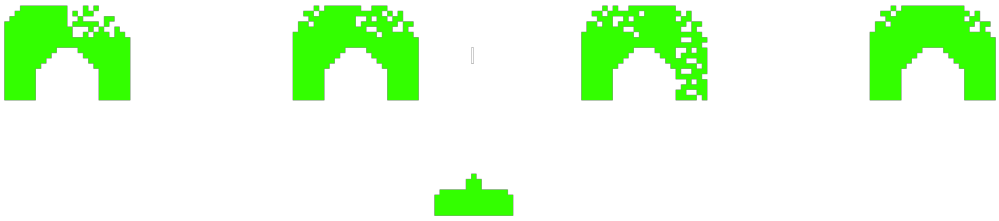
> De ruimtewezens werpen af en toe bommen af. Wordt het kanon door een bom geraakt, dan is het spel afgelopen.

Bommen en kogels hebben veel gemeen. Misschien kan je vermijden om hier al te veel te knippen en te plakken, misschien door een gemeenschappelijke bovenklasse *Projectiel* te introduceren.

> Onderaan het scherm staan drie bunkers waaronder het kanon kan 'schuilen'. De bunkers zijn redelijk goed bestand tegen bommen, maar als ze te veel geraakt zijn, zullen ze uiteindelijk toch verdwijnen,

In het oorspronkelijke videospel brokkelen de bunkers langzaam af telkens wanneer ze een bom incasseren.

[3] Het voorbeeldproject dat je kan downloaden, bevat ook enkele bijkomende afbeeldingen.



In Greenfoot is dit heel moeilijk te realiseren. Als alternatief kan je de bunkers eventueel van kleur laten veranderen naarmate ze zijn beschadigd.

> Je kan *punten* verdienen door wezens neer te schieten.

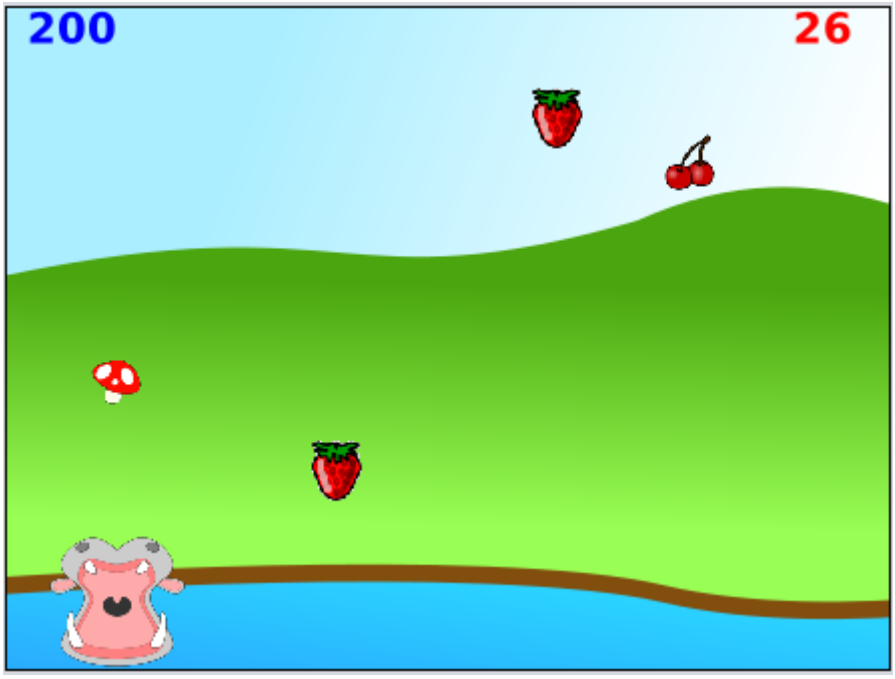
Elk type ruimtewezen correspondeert met een ander puntental — de bovenste rij is meer waard dan de onderste.

> Af en toe verschijnt er helemaal bovenaan een vliegende schotel die snel van de ene kant van het scherm naar de andere kant vliegt.

Deze schotel is heel wat punten waard.

# 3. Hap!

In dit spelletje moet een nijlpaard zoveel mogelijk fruit vangen dat echter altijd maar sneller en sneller op hem afkomt.




De oorspronkelijke auteur van dit scenario is *Thomas Tortelboom*. De afbeeldingen zijn van de hand van de auteur of komen uit de Greenfoot-bibliotheek.

## 3.1 Beschrijving

1. Je beweegt het nijlpaard naar links en naar rechts met de pijltjestoetsen.
2. Ondertussen probeer je fruit te vangen dat van boven naar beneden valt.
3. Een aardbei levert je 10 punten op, een kers 30. Paddenstoelen moet je vermijden, want die kosten je een 'leven'.
4. Je verliest ook een leven wanneer een stuk fruit de onderrand raakt vooraleer je het kan vangen.
5. Het fruit verschijnt en valt sneller naarmate het spel vordert.

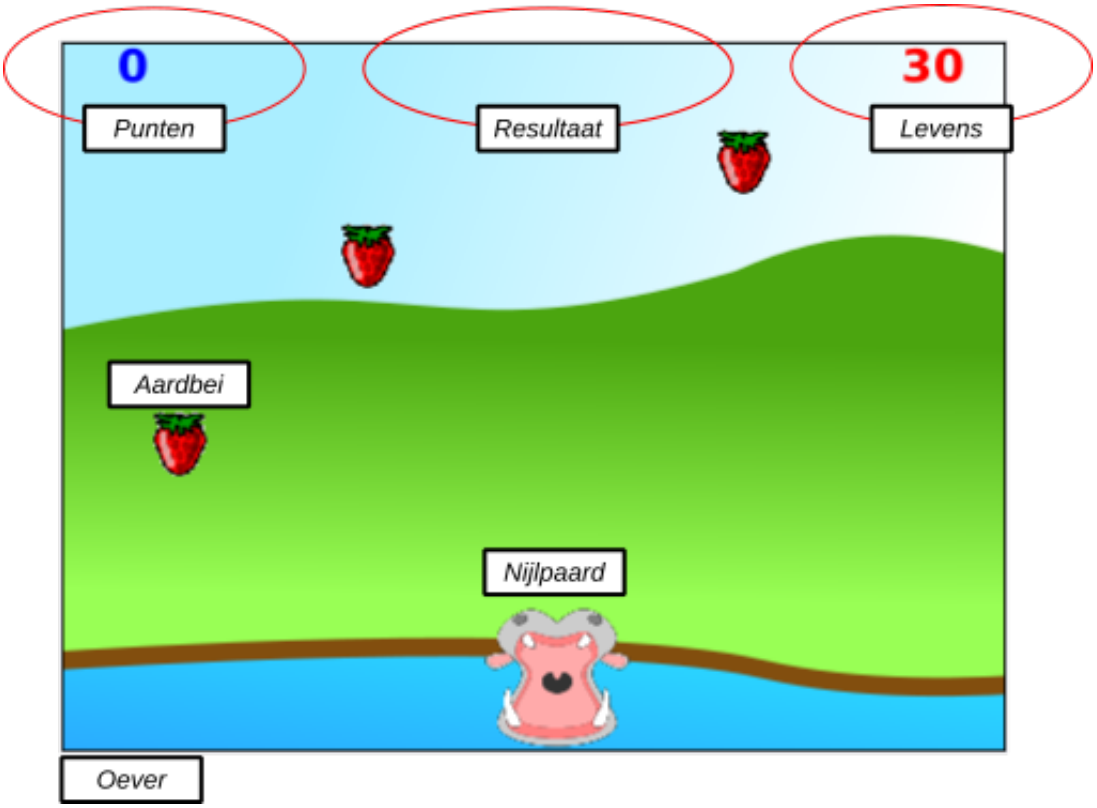
6. Het spelt stopt wanneer je 300 punten hebt behaald, of wanneer je 30 levens hebt verspeeld.

De eerste versie van dit programma heette *Jeugdclub*. In de plaats van een nijlpaard was er grote mond die probeert glazen bier te vangen. Wij kozen voor een 'bravere' versie om copyrightproblemen te voorkomen met de foto's. Het staat je natuurlijk vrij om de afbeeldingen in dit spel door je eigen creatieve versies te vervangen.

## 3.2 Eerste stappen

Voor deze opgave beginnen we met een scenario dat reeds enkele klassen bevat. Je kan het downloaden van de website voor deze cursus. Dit hebben we reeds voor jou geprogrammeerd:

- De wereldklasse heet *Oever*. Ze bestaat uit 20×15 cellen van 22×22 pixels. We hebben reeds enkele objecten op deze wereld geplaatst.



- Een object van de klasse *Punten* toont in de linker bovenhoek welke score de speler reeds heeft behaald. Een gelijkaardig object van de klasse *Levens* toont het aantal levens dat nog overblijft.
- De klasse *Resultaat* dient om op het einde van het spel te tonen of de speler is gewonnen of verloren. Bij het begin van het spel staat er reeds een object van deze klasse op de juiste plaats, maar de afbeelding ervan is nog 'leeg' — waardoor het onzichtbaar wordt.
- De klassen *Nijlpaard* en *Aardbei* tonen de bewegende objecten. Je zal later nog een klasse *Kersen* en een klasse *Paddenstoel* toevoegen.

**Stap 1** Bestudeer de Java-code die in dit startscenario reeds is geprogrammeerd. Begrijp je wat ze doet? In de Greenfoot-documentatie vind je meer uitleg bij de methodes die je misschien niet meteen herkent.

De afbeelding van een aardbei komt uit de bibliotheek van Greenfoot. Deze standaardafbeelding is echter veel te groot (39×47 pixels) voor de cellen van 22×22 pixels die we hier gebruiken. Om de prent te verkleinen, kan je twee dingen doen:

- Je zoekt in welke map de prent staat en bewerkt een kopie ervan met een tekenprogramma.
- Je laat je programma zelf de prent verkleinen.

Wij kozen voor de laatste optie: in de constructor van *Aardbei* vragen we de afbeelding op die aan dit object is toegekend, en herschalen ze naar de juiste afmetingen met de methode *scale*:

```
public Aardbei () {
    getImage().scale(25,30);
}
```

We doen hetzelfde met het nijlpaard.

Elk object dat op de wereld wordt geplaatst (elke 'acteur') heeft een bijbehorende afbeelding. Meestal komt deze uit een bestand, maar ze kan ook leeg zijn (**null**) zoals bij *Resultaat*:

```
public Resultaat() {
    setImage ((GreenfootImage)null);
}
```

of enkel uit tekst bestaan zoals bij *Punten* en *Levens*:

```
private int totaal = 0;

public Punten () {
    toon();
}

private void toon() {
    setImage( new GreenfootImage(Integer.toString(totaal),
                    25,   Color.BLUE, null) );
}
```

De eerste parameter van **new** *GreenfootImage*(..) is de tekst die wordt afgebeeld — hier het puntental, omgezet naar een string. De tweede parameter is de grootte van het lettertype (25 pixels), de derde is de kleur waarin de tekst wordt gezet (blauw) en de vierde de achtergrondkleur — of **null** als die doorzichtig is.

Je zal de methode *toon* nog nodig hebben als je later de punten wil aanpassen.

**Stap 2** Plaats onderaan een nijlpaard en zorg dat het naar links en naar rechts beweegt met de pijltjestoetsen.

Plaats drie aardbeien op willekeurige plaatsen in de bovenste helft van de wereld en laat ze naar beneden vallen.

Er zijn twee manieren om op toetsen te reageren: ofwel gebruik je *isKeyDown*(..), ofwel *getKey*(). Wij kozen voor het laatste — dit maakt het spel beter 'speelbaar': je kan de toets dan bijvoorbeeld niet ingedrukt blijven houden.

Als je geen voorzorgen neemt, vallen de aardbeien aan een helse vaart naar beneden — de '*act*()'-pulsen volgen elkaar veel te snel op voor cellen van deze grootte. Je kan een vertraging inbouwen door de aardbei telkens maar om de 30 stappen één cel naar beneden te verplaatsen: hou een tellertje bij dat je telkens met één verhoogt en reageer pas wanneer dit de waarde 30 heeft bereikt (waarna je het terug op 0 zet[1]). Hou er misschien rekening mee dat we straks dit getal 30 beetje bij beetje zullen verminderen om het fruit steeds sneller uit de hemel te laten vallen.

[1] In andere opdrachten hebben we een alternatief gebruikt waarbij we enkel reageren wanneer het totaal aantal pulsen precies deelbaar is door 30. Dit werkt hier niet zo goed omdat het dan moeilijker wordt om de snelheid na verloop van tijd mooi te laten toenemen.

**Stap 3** Komt de aardbei onderaan het scherm of raakt ze het nijlpaard, dan verdwijnt ze.

Elke 210 pulsen komt er een nieuwe aardbei bij, bovenaan het scherm, op een willekeurige X-positie.

Zoals in vele andere scenario's heb je hier de keuze of je het nijlpaard laat detecteren of het een aardbei raakt, of omgekeerd. Omdat er maar één nijlpaard is, en omdat het in principe ook mogelijk is dat er meerdere aardbeien tegelijk het nijlpaard kunnen raken, kiezen we er hier voor om de klasse *Aardbei* de botsingsdetectie te laten doen.

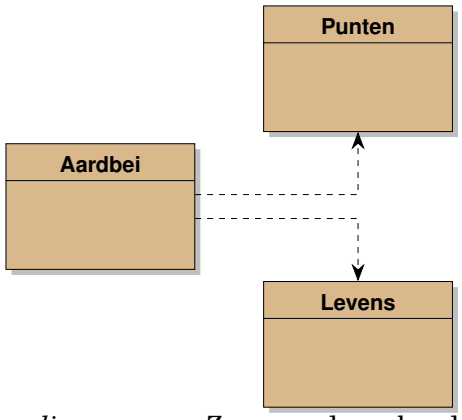
We kijken ook pas of de aardbei en het nijlpaard elkaar overlappen wanneer de aardbei bijna onderaan het scherm is aangekomen — het is niet voldoende dat de aardbei het het topje van de neus van het nijlpaard treft om opgegeten te worden.

## 3.3 Punten en levens tellen

In een volgende stap willen we punten toekennen aan elke aardbei die is opgegeten en levens aftrekken voor elke aardbei die de onderkant van het scherm bereikt. We staan echter eerst even stil bij het ontwerp van het programma: hoe zorgen we ervoor dat elke aardbei gemakkelijk toegang heeft tot het *Punten*-object en het *Levens*-object bovenaan het scherm? Er zijn opnieuw verschillende opties[2].

Als eerste alternatief kunnen we velden *punten* en *levens* voorzien in de klasse *Aardbei*. Deze velden kan je dan aanspreken in de methode *act*() wanneer een aardbei de rand bereikt of het nijlpaard raakt.

De velden moeten echter eerst een correcte waarde krijgen. Wanneer een nieuw *Aardbei*-object wordt aangemaakt, moet je er referenties aan toekennen naar de bestaande *Punten*- en *Levens*-objecten.




[2] De diagrammen op deze en volgende pagina's zijn *klassendiagrammen*. Ze geven de verbanden aan tussen verschillende klassen in het programma. Een pijl met een streepjeslijn betekent '*verwijst naar*' of '*gebruikt minstens één object van*'. Een volle lijn met een driehoekige pijl betekent '*is een*' of '*is een uitbreiding van*'.

Dit gebeurt best in de constructor:

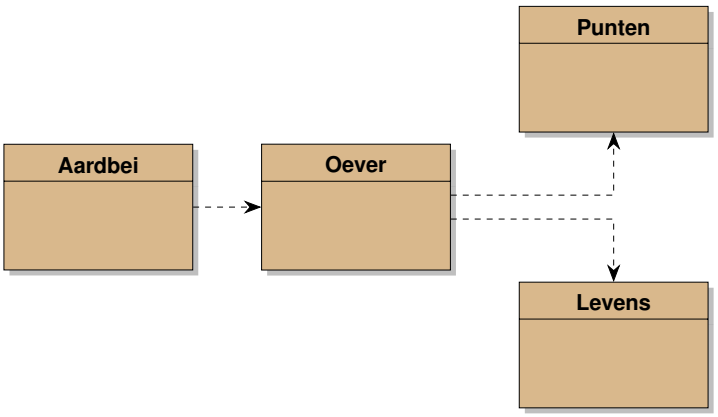
```
public class Aardbei {

    private Punten punten;

    private Levens levens;

    public Aardbei (Punten p, Levens l) {
        ...
        punten = p;
        levens = l ;
    }

    public void act() {
        // doe iets met punten en levens
        ...
    }
}
```

en wanneer *Oever* dan nieuwe aardbeien aanmaakt, geef je (verwijzingen naar) de bestaande *Punten*- en *Levens*-objecten mee als argumenten:

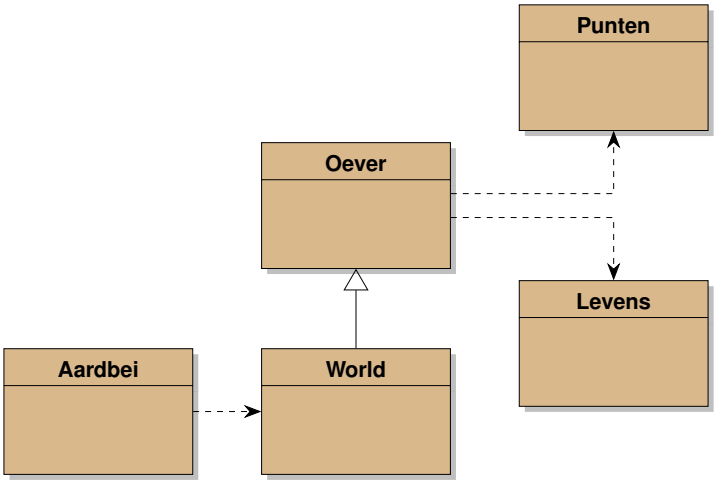
```
public class Oever {

    private Punten punten;

    private Levens levens;

    public Oever () {
        ...
        Aardbei aardbei = new Aardbei (punten, levens);
        ...
    }
}
```

Voor wie dit nogal omslachtig vindt — en het wordt er niet beter op wanneer je ook nog het *Resultaat*-object in *Aardbei* wil gebruiken — is er nog een tweede mogelijkheid: je zorgt ervoor dat de *Punten*- en *Levens*-objecten kunnen bereikt worden langs een tussenweg: via het *Oever*-object.

De klasse *Oever* bevat immers al referenties naar deze beide objecten, het is dus enkel nog een kwestie om in *Oever* gepaste publieke methoden te voorzien die iets met deze objecten doen.




Je kan het *Oever*-object doorgeven aan het *Aardbei*-object in de constructor op dezelfde manier als we hierboven met de punten den levens hebben gedaan, maar er is ook een andere manier. Het *Oever*-object dat je nodig hebt, is immers niets anders dan de wereld waarop de aardbei is geplaatst — en er is reeds een methode *getWorld*() waarmee je die wereld kunt opvragen.




Er is echter een vervelende complicatie: het retourtype van *getWorld* is *World* en niet *Oever*! Dit betekent dat Java je enkel zal toelaten om voor het object dat *getWorld*() teruggeeft methoden op te roepen uit de klasse *World*, maar geen methoden die je aan *Oever* toevoegt. Java is niet slim genoeg om door te hebben dat onze wereld steeds tot de klasse *Oever* zal behoren!

Vanaf de nieuwere versies van Greenfoot bestaat hievoor gelukkig een oplossing: gebruik de methode *getWorldOfType*(..) zoals in methode *getOever*() hieronder die de wereld terug met als type *Oever* in plaats van *World*:

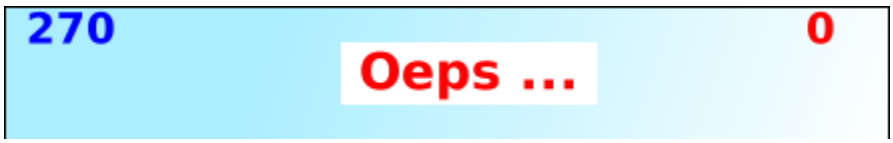
```
public class Aardbei {

    public Oever getOever() {
        return getWorldOfType(Oever.class);
    }

    ...
}
```

Met deze techniek hoef je de constructor van *Aardbei* helemaal niet aan te passen en kan je nieuwe aardbeien aanmaken met een eenvoudige '**new** *Aardbei*()'.

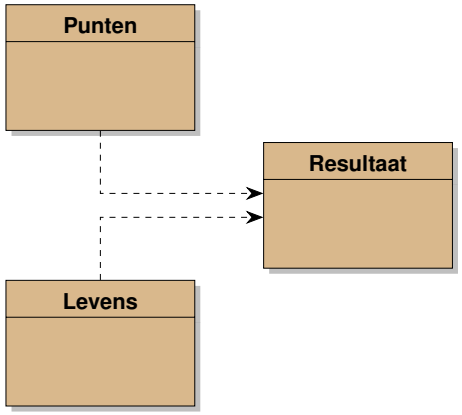
**Stap 4** Voeg aan *Oever* een methode toe waarmee je de punten met 10 kunt verhogen. Je zal hiervoor ook aan de klasse *Punten* iets moeten wijzigen. Voeg op dezelfde manier een methode toe waarmee je de levens met één kunt verminderen.

Zorg ervoor dat punten en levens tijdens het spel worden aangepast. Gebruik hiervoor de methode *getOever* zoals hierboven uitgelegd.

Om het spelletje af te werken, willen we nu ook nog dat het programma stopt als je voldoende punten hebt behaald of te veel levens hebt verloren en dat je een gepaste melding ziet bovenaan het scherm.




Hiervoor heb je onder andere de klasse *Resultaat* nodig. En de vraag stelt zich meteen wiens taak het zal zijn om dit resultaat aan te passen.




Welke objecten weten het best wanneer het spel ten einde is? Inderdaad, het *Punten*- en het *Levens*-object. Je zal dus het programma moeten aanpassen zodat aan beide objecten bij constructie een verwijzing naar het *Resultaat*-object wordt meegegeven.

**Stap 5** Beëindig het spel zoals hierboven beschreven.

**Stap 6** Telkens wanneer er een nieuwe aardbei bovenaan het scherm verschijnt, drijf je het tempo een klein beetje op.

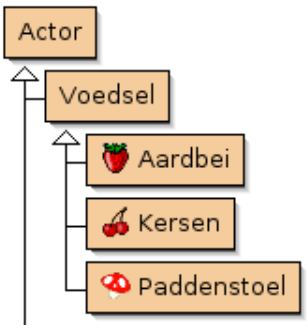
Het getal 30 dat je gebruikt om te weten wanneer de aardbei een cel naar beneden moet (de '*snelheid*'), verminder je telkens met 1. In plaats van 210 te nemen voor het aantal pulsen dat je wacht vooraleer je een nieuwe aardbei aanmaakt, neem je $7 \times$*snelheid*.

Opgelet: *snelheid* hoort niet thuis in de klasse *Aardbei* — want het ganse spel wordt hierdoor beïnvloed. Bewaar de snelheid daarom in de klasse *Oever* en zorg voor een gepaste 'getter'.

Ziezo, je hebt goed gewerkt. Je kleine zus kan het spelletje nu uitproberen[3].

## 3.4 Meer fruit

Om het spel volledig af te werken, hoef je enkel nog kersen en paddenstoelen toe te voegen.




Er is weinig verschil met de aardbeien die we reeds geprogrammeerd hebben — zo weinig zelfs dat we ons heel wat werk besparen door een gemeenschappelijke bovenklasse *Voedsel* in te voeren waarvan de drie andere klassen worden afgeleid.

[3] Testen vormt een belangrijk onderdeel van het software-ontwikkelingsproces.

Wat zijn dan nog de verschillen?

- Elk voedsel heeft een eigen afbeelding, met eigen afmetingen (Kersen: 25×28, Paddenstoel: 25× 23).
- Kersen leveren 30 punten op in plaats van 10.
- Wanneer je een paddenstoel opeet, kost je dat een leven. Laat je die echter passeren dan kost je dat niets.

Dit betekent dat we in *Voedsel* een heel stuk van de oorspronkelijke code van *aardbei* kunnen overnemen, behalve het stukje in *act*() dat beslist wat er met de punten en levens moet gebeuren. Hiervoor introduceren we dan een nieuwe methode *doeOnderaan*() die we in elke klasse op een andere manier invullen.

```
public class Voedsel {
    ...
    public void act() {
        pulsen = pulsen + 1;
        if (pulsen >= getOever().getSnelheid()) {
            setLocation (getX(), getY() + 1);
            pulsen = 0;
            if (getY() == 14) {
                doeOnderaan();
                getWorld().removeObject (this);
            }
        }
    }
}
```

**Stap 7** Voeg kersen en paddenstoelen toe aan het programma zoals hierboven beschreven.

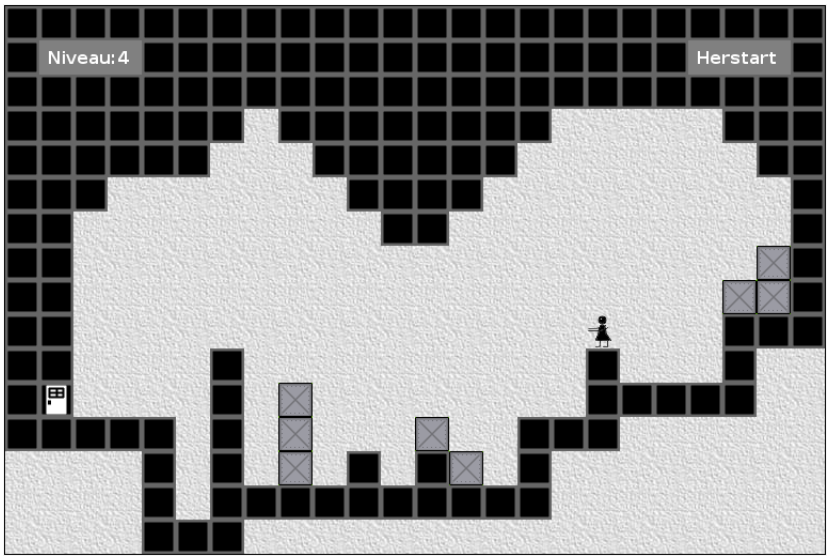
Laat niet enkel aardbeien verschijnen bovenaan het scherm, maar ook paddenstoelen (met 1 kans op 6) en kersen (1 kans op 8).

**Uitdaging** Aardbeien en kersen lijken zodanig goed op elkaar dat je er geen twee verschillende klassen voor nodig hebt maar ze objecten kunt maken van één en dezelfde klasse *Fruit*. Dat maakt het ook gemakkelijker om later nog ander fruit aan het programma toe te voegen. Hetzelfde kunnen we zeggen over de punten en levens.

Pas dit aan in het programma.

# 4. Block Dude

Deze oefening is gebaseerd op het klassieke spelletje '*Block dude*', oorspronkelijk ontwikkeld door *Brendan Sterner* voor de Texas Instruments TI-84 rekenmachine. De bedoeling van het spel is om te ontsnappen uit verschillende 'grotten' door het oppakken en neerzetten van blokken en daarbij goed op te letten dat je niet komt vast te zitten.




Dit scenario werd oorspronkelijk uitgewerkt door *Nils Mak*. De afbeeldingen zijn van de hand van de auteur of komen uit de Greenfoot-bibliotheek.

**Opgelet!** Net zoals bij *Space Invaders* (hoofdstuk 2) gebruiken we hier *Stride* voor dit scenario. Je kan deze opgave echter even gemakkelijk programmeren in Java.

## 4.1 Beschrijving

1. Je bestuurt het 'mannetje' met de pijltjestoetsen. Met LINKS kijk je naar links en stap je één cel vooruit, als er tenminste geen muur of blok in de weg staat. Met RECHTS kijk je en stap je op dezelfde manier naar rechts.
2. Je kan met OMHOOG een muur of blok van één cel hoog beklimmen. Daarvoor moet je er vlak naast staan en ernaar kijken.

3. Met OMLAAG kan je een blok oppakken of een blok terug neerzetten, telkens in de kijkrichting. Je kan op die manier een blok ook op een ander blok of op een muur plaatsen, maar je kan een blok niet oppakken vanop een ander blok of muur.
4. De speler en de blokken vallen recht naar beneden wanneer ze niet door een blok of muur worden ondersteund.
5. Er zijn verschillende grotten, één voor elk niveau. Wanneer je op één niveau de deur bereikt, ga je automatisch naar het volgende niveau.
6. Rechts bovenaan is er een knop waarmee je het huidige niveau kan herstarten als je vast komt te zitten.

## 4.2 De grot

Vooraleer we de beweging van het mannetje programmeren, willen we eerst de grot opbouwen. Hoe de grot is ingedeeld — welke objecten in welke cellen staan — beschrijven we aan de hand van een soort 'kaart'. (In het *Sokoban*-scenario uit *Greenfoot – Java spelenderwijs* doen we iets gelijkaardigs.)

**Stap 1** Open het start-project *blockdude*. Zoals je ziet, hebben we reeds een aantal klassen klaargezet, enkele met bijbehorende programmacode. Zo is de klasse *Indelingen* al volledig voor jou geprogrammeerd. Met *Indelingen.get*(*n*) kan je bijvoorbeeld de 'indeling' van niveau *n* ophalen. Bekijk de code van deze klasse. Begrijp je hoe we hier de indeling van een niveau voorstellen?

De indeling van een grot wordt voorgesteld als een tabel (array) van strings (type: *String*[]). Elk element in deze tabel is een string van dezelfde lengte, namelijk de breedte van de grot, uitgedrukt in aantal cellen. De hoogte van de grot komt overeen met het aantal elementen in de tabel (de 'lengte' van de tabel). Elk letterteken beschrijft één cel, op de volgende manier:

| Letterteken | Element |
|---|---|
| `0` | Muur |
| `.` | Lege ruimte |
| `x` | Blok |
| `e` | Deur (einde) |
| `<` | Speler die naar links kijkt |
| `>` | Speler die naar rechts kijkt |

**Stap 2** Bekijk de (wereld)klasse *Grot*. Snap je hoe de methoden *volgendNiveau* en *herstartNiveau* werken?

Merk op dat de klasse drie verschillende constructoren heeft. De eerste moet je straks zelf programmeren, de andere twee zijn reeds ingevuld. Begrijp je wat de opdracht **this**(..) doet in deze constructoren?

Een opdracht van de vorm **this**(..) voert de ene constructor uit als start van een andere.

```
Eerste constructor
private Grot (int niveau, String[] indeling) {
    super(indeling[0].length(),indeling.length,32)
     ...
}

Tweede constructor - roept de eerste op
public Grot (int niveau) {
    this(niveau, Indelingen.get(niveau))
}

Derde constructor - roept de tweede op
public Grot () {
    this(0)
}
```

Hier betekent dit dat een oproep van de derde constructor, bijvoorbeeld '**new** *Grot*()', hetzelfde doet als een een oproep van de tweede constructor met als argument 0: '**new** *Grot*(0)'. En deze oproep heeft dan weer hetzelfde effect als de volgende oproep van de eerste constructor: '**new** *Grot*(0, *Indelingen.get*(0))'.

**Stap 3** Vul de eerste constructor aan met code die blokken, muren, e.d. in de grot plaatst zoals aangegeven in de *indeling*. Je hoeft voorlopig geen rekening te houden met de richting waarin het mannetje kijkt.

De variabele *indeling* is een *tabel* van strings. Het aantal rijen in de grot is dus gelijk aan *indeling.length*. Het aantal kolommen komt dan weer overeen met het aantal lettertekens in elk van die strings. Aangezien elke rij even lang is, kunnen we hiervoor dus bijvoorbeeld de lengte van de eerste string gebruiken: *indeling*[0].*length*() — let op de ronde haakjes!

Merk ook op dat de rijnummers overeenkomen met de Y-richting en kolomnummers met de X-richting. Het opvullen van de grot met de juiste objecten, volgt dus het volgende patroon:

```
for each (int rij in 0 .. indeling.length−1)
    for each (int kolom in 0 .. indeling[rij ].length() − 1)
        var char code ⇐ indeling[rij].charAt(kolom)
        Plaats het juiste object (zie code) op positie x=kolom, y=rij
```

Gebruik *addObject*(*object*, $x$, $y$) om een *object* op de wereld (in de grot) te plaatsen op positie $(x,y)$.

**Stap 4** Maak enkele objecten aan van het type *Info* door rechts op de klasse te klikken in het Greenfoot-venster en een parameter in te vullen. Gebruik dit om de labels 'Niveau: 0' en 'Herstart' op de juiste plaats op de wereld te zetten. Gebruik *Inspect* om na te gaan wat de X- en Y-coördinaten van die labels zijn.


**Stap 5** Voeg op het einde van de eerste constructor van *Grot* de code toe die nodig is om de twee labels te plaatsen. (Klikken op HERSTART heeft voorlopig nog geen effect.)

## 4.3 Bewegen

Greenfoot biedt verschillende manieren om een object over de wereld te laten bewegen. Meestal gebruikt men *move* en *turn* omdat die gemakkelijk te begrijpen zijn, maar deze opdrachten zijn vooral bedoeld voor werelden die van bovenaf worden bekeken, omdat de afbeelding van het object meedraait met *turn*. In het *Wandeling*-scenario uit *Greenfoot – Java spelenderwijs* hebben we dit opgelost door na elke beweging *setRotation* te gebruiken om de afbeelding weer in de juiste richting te draaien, in dit scenario gebruiken we een andere techniek: we verplaatsen een object door rechtstreeks zijn coördinaten aan te passen.

Inderdaad, om bijvoorbeeld een object één cel naar links of rechts te verplaatsen, hoef je enkel zijn X-coördinaat met één te verminderen of te vermeerderen, en analoog met de Y-coördinaat om naar onder of naar boven te bewegen.

| **1 cel naar** | |
|---|---|
| Links | *setLocation* (*getX*()−1, *getY*()) |
| Rechts | *setLocation* (*getX*()+1, *getY*()) |
| Onder | *setLocation* (*getX*(), *getY*()+1) |
| Boven | *setLocation* (*getX*(), *getY*()−1) |

(Let op: De Y-as wijst naar onder, dat is het omgekeerde van wat je in de wiskunde gewoon bent.)

**Stap 6** Zorg ervoor dat de speler één stap naar links beweegt wanneer je op de LINKS-toets drukt en één stap naar rechts wanneer je op de RECHTS-toets drukt. Gebruik *Greenfoot.isKeyDown* om te kijken of een toets was ingedrukt.

Verplaatsen alleen is niet genoeg, je wil ook dat de juiste afbeelding van het mannetje wordt getoond — *man-links.png* als het mannetje naar links kijkt, *man-rechts.png* als het naar rechts kijkt. We stellen voor dat je dit op de volgende manier programmeert:

**Stap 7** Voeg een veld *richting* toe aan *Speler* dat bijhoudt in welke richting de speler kijkt: -1 voor links en +1 voor rechts. Pas de constructor van *Speler* aan zodat je deze richting als parameter kan meegeven, en hou nu ook rekening met de kijkrichting bij het bevolken van de *Grot*-klasse vanuit een gegeven indeling. Pas *act* aan zodat bij het indrukken van een toets ook de nieuwe kijkrichting wordt ingesteld.

Schrijf een methode *updateAfbeelding* (in *Speler*) die de afbeelding van de speler aanpast aan zijn huidige *richting*. Roep deze methode op als laatste opdracht van *act*[1] en van de constructor.

Gebruik *setImage*(..) om de afbeelding in te stellen.

Met wat je tot nu al hebt geprogrammeerd, kan je niveau 0 van het spel al uitspelen. Je moet enkel nog detecteren wanneer je de deur bereikt. Er zijn hiervoor twee mogelijkheden: ofwel laat je de speler dit detecteren, ofwel laat je dit over aan de deur. We kiezen voor de tweede mogelijkheid, ook al ligt die minder voor de hand. Door de broncode over de verschillende klassen te verspreiden, wordt het programma overzichtelijker.

[1] Het lijkt misschien wat overdreven om bij elke tik van de Greenfoot-klok opnieuw de afbeelding in te stellen, zoveel verandert die immers niet. Dit maakt het echter heel wat eenvoudiger om een foutloos programma te schrijven. Greenfoot laadt trouwens hetzelfde afbeeldingsbestand slechts één keer in, dus zoveel nutteloos werk wordt er niet gedaan.

**Stap 8** Pas de methode *act* van *Deur* aan zodat het spel naar een volgende niveau gaat zodra de deur een speler 'raakt'.

In stap 2 zag je dat de klasse *Grot* een methode *volgendNiveau* had. Het komt er dus op aan dat de deur aan 'zijn' grot vertelt deze methode uit te voeren:

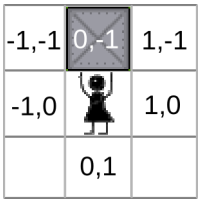
```
getWorldOfType(Grot.class).volgendNiveau()
```

In Greenfoot bestaat er een methode *getWorld* die de huidige actieve wereld teruggeeft. Deze wereld is echter niet van het type *Grot* maar van het type *World*. De methode *getWorldOfType* die we hier gebruiken, heeft wel een wereld terug van het juiste type.

Met wat je nu al hebt geprogrammeerd, kom je in het volgende spelniveau niet ver: je kan door muren en blokken stappen, je valt niet naar beneden wanneer je niet wordt ondersteund en je kan geen blokken oppakken.

Het komt er in vele gevallen op neer om te kijken wat er zich naast, onder, boven of schuin boven de speler bevindt. Hierbij gebruik je best de methode *getOneObjectAtOffset*(*x*,*y*,*klasse*).

Deze methode kijkt of er op de opgegeven positie een object bevindt van een opgegeven *klasse* en geeft dit object terug — of **null** wanneer er daar geen dergelijk object te vinden is.




De coördinaten *x*, *y* die je meegeeft als parameter zijn geen *absoluut* coördinaten, maar zijn *relatief* ten opzichte van het object dat de methode oproept (dat is wat men met het Engelse woord *offset* bedoeld). Hiernaast schetsen we de *offsets* die je voor dit scenario nodig hebt.

Als *klasse*-parameter kan je ook *Actor*.**class** meegegeven. Zo kijk je bijvoorbeeld of je door een blok of muur (of een ander ding) wordt ondersteund:

```
if (getOneObjectAtOffset(0,1,Actor.class) == null)
        Ik wordt niet ondersteund
else
        Ik wordt wel ondersteund
```

**Stap 9** Pas het programma aan zo dat

- Een speler niet naar links kan stappen wanneer er links iets staat, en niet naar rechts wanneer er rechts iets staat. Opgelet! De richting waarin de speler kijkt, moet wel veranderen.
- Een speler naar beneden valt wanneer hij niet ondersteund is. Terwijl een speler valt, mag je niet op de toetsen reageren.
- Een blok naar beneden valt wanneer het niet ondersteund is.

Je kan het vallen testen door in het Greenfoot-venster, vóór je het scenario uitvoert, een blok (of het mannetje) vast te nemen met de muis en naar een andere cel te verslepen.

**Stap 10** Hierboven zijn we wat te streng geweest: de speler kan wel opzij bewegen wanneer daar een *deur* staat.

Schrijf daarom een (hulp)methode *kanBewegen* (in *Speler*) die **true** of **false** teruggeeft als de speler al dan niet een stap kan zetten in de richting waarin hij kijkt. (Je mag aannemen dat de speler niet aan het vallen is.) Pas je programma aan zodat het die methode gebruikt.

Je kan hier handig gebruik maken van het feit dat we de richting waarin de speler kijkt hebben aangegeven met $\pm 1$. Schrijf dus bijvoorbeeld *niet*

```
if (richting == −1)
    if (getOneObjectAtOffset(−1, 0) == null)
        return true
    else
        return false
else if (richting == +1)
    if (getOneObjectAtOffset(+1, 0) == null)
        return true
    else
        return false
```

maar kort dit af tot één enkele opdracht:

```
return getOneObjectAtOffset(richting, 0) == null
```

**Stap 11** Schrijf een hulpmethode *kanOmhoogBewegen* die aangeeft of de speler (schuin) omhoog kan stappen in de huidige kijkrichting. Hiervoor moet er vóór de speler een object staan, maar één cel hoger niet (of het moet een deur zijn). Een speler mag ook bovenop een deur gaan staan.

Zorg dat het programma nu ook correct reageert op de OMHOOG-toets.

## 4.4 Een blok oppakken, verplaatsen en terug neerzetten

Wanneer een speler een blok oppakt, moet hij dit blok ook blijven 'dragen': ervoor zorgen dat het zich altijd één cel boven zijn positie bevindt. In dit geval is het dus niet de *act*() van *Blok* die voor de beweging zorgt, maar die van *Speler*.

Om dit mogelijk te maken, voeg je aan *Speler* een veld *blok* toe (van het type *Actor*[2]) met een verwijzing naar het blok dat gedragen wordt, of anders **null**.

**Stap 12** Laat het programma reageren op de OMLAAG-toets door een blok op te pakken dat naast de speler ligt, in de kijkrichting. Doe niets wanneer de speler al een blok draagt.

Breidt de methode *act* van *Speler* onderaan uit met code waarmee een blok dat gedragen wordt telkens de correcte positie krijgt.

Pas de methode *updateAfbeelding* aan zodat er voor een mannetje dat een blok draagt een andere afbeelding gebruikt wordt — de afbeeldingsbestanden heten *man-links-blok.png* en *man-rechts-blok.png*.

Heb je dat nog niet gedaan, dan zou je nu best de *act*() van *Speler* opsplitsen in kleinere methodes. Schrijf bijvoorbeeld een methode *beweeg*() die zich enkel bezighoudt met het reageren op de toetsen, en op haar beurt methoden *zetStap*, *zetStapOmhoog*, *pakBlok* (en straks: *zetBlok*) gebruikt, telkens in kijkrichting.

**Stap 13** Implementeer de methode *zetBlok* en roep die op wanneer OMLAAG is ingedrukt terwijl je een blok draagt.

Is er plaats, zet dan het blok naast je neer. Kan dit niet omdat daar reeds een ander ding ligt, kijk dan of de plaats erboven leeg is en plaats het blok daar.

[2] Het veld zou eigenlijk beter van het type *Blok* zijn, maar dat brengt je later in de problemen.

**Stap 14** Zorg dat het niveau terug van bij het begin opstart wanneer je op de HERSTART-knop klikt (rechts bovenaan het scherm). Schrijf hiervoor een nieuwe actor-klasse *HerstartKnop* die ongeveer hetzelfde doet als *Info*, maar net iets meer...

De correcte manier om dit te doen, is om van *HerstartKnop* een *uitbreiding* te maken van *Info*. De constuctor van *HerstartKnop* moet dan de constructor van *Info* oproepen met behulp van **super**(..):

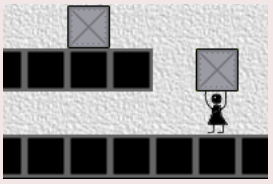
**class** *HerstartKnop* **extends** *Info*

*Constructors*
  **public** *HerstartKnop*()
    **super** (`"Herstart"`)

*Methoden*
  **public** *act*()
    *Nog te implementeren...*

Gebruik *Greenfoot.mouseClicked*(**this**) om te kijken of men op de knop heeft geklikt.

**Uitdaging** Wanneer je je verplaatst terwijl je een blok draagt, wordt er niet gekeken of dit blok wel overal kan passeren: voor we naar links stappen, gaan we bijvoorbeeld wel na of de plaats links van de speler vrij is, maar kijken we niet naar de plaats daarboven.



Bij de grotindelingen die we je hebben bezorgd, vormt dit geen probleem, maar je kan gemakkelijk grotten bedenken waar dit wel zo is.

Pas het programma aan zodat je hierop controleert.