

Spelen met Java



Java leren
door het
maken van
een spel in
Greenfoot



Spelen met Java

**Java leren door het maken van een
spel in Greenfoot**

© 2012 - 2013 Ton van Beuningen

Voorwoord

Deze syllabus is de uiteindelijke versie en zal alleen wijzigen als er fouten in de syllabus staan of als er nieuwe mogelijkheden aan Java of Greenfoot worden toegevoegd.

De code in deze syllabus is getest met Java JDK 1.7.0_05 en met Greenfoot versie 2.2.1.

De kritische opmerkingen van Gijs Wassink zijn de kwaliteit van deze syllabus zeer ten goede gekomen.

Inhoud

Voorwoord	2
0. Inleiding.....	5
1. 0.1. Benodigdheden	5
2. 0.2 Het spel <i>The World of Garp</i>	5
1. Een allereerste begin.....	6
3. 1.1 Eerste kennismaking met Greenfoot	6
4. 1.2 De eerste twee spelelementen	7
5. 1.3 Van klasse naar object.....	9
6. 1.4 Zij doen niets	9
2. Besturing en beweging	11
7. 2.1 De code van Greenfoot	11
8. 2.2 Compileren	13
9. 2.3 Het rechte pad van Garp	14
10. 2.4 Garp rechtop	15
11. 2.5 De eerste stappen op het dievenpad	18
12. 2.6 Het einde van de wereld	20
13. 2.7 Het geheim van het dievenpad	23
3. De wereld van Garp	27
14. 3.1 Garp en Gnomus samen in de wereld.....	27
15. 3.2 De wereld verrijken	30
16. 3.3 De stilte voorbij	31
17. 3.4 Erop of eronder	33
4. Leven en niet laten leven	34
18. 4.1 Diamanten verzamelen	34
19. 4.2 Orde op zaken stellen.....	34
20. 4.3 Als een rots in de branding	36
21. 4.3 De ontploffing in beeld.....	37
22. 4.3 De ontploffing met geluid	39

23. 4.4 Garp ontploft.....	40
24. 4.5 Gnomus is aan de beurt	41
25. 4.6 Het einde van Garp	42
5. De buit verdelen	44
26. 5.1 De diamanten tellen.....	44
27. 5.2 Het einde nadert	48
Extra opdrachten.....	53
Bijlage 1: Kleuren.....	54
Bijlage 2: De klasse Score	55

0. Inleiding


Om een spel te maken kun je verschillende programma's gebruiken. Hier is de keuze gemaakt voor Greenfoot. Je leert dan meteen programmeren in Java. Allereerst wordt verteld wat je nodig hebt en waar je dat kunt downloaden. Daarna wordt het spel uitgelegd, dat we gaan maken.

0.1. Benodigheden

Greenfoot is op Java gebaseerd. Daarom moet je twee keer iets downloaden: namelijk het programma Greenfoot en de programmeertaal Java. Ga naar



<http://www.greenfoot.org> en klik op The software. Daar kun je zowel Java als Greenfoot downloaden. Voor Java wordt je naar de site van Oracle doorverwezen. Kies daar voor de latest Release en zorg ervoor dat je de Java Development Kit (JDK) 7 of hoger voor het juiste platform downloadt en installeert.

Daarna ga je terug naar de site van Greenfoot en download je de versie van Greenfoot die geschikt is  voor het besturingssysteem op de computer. Deze installeer je en als Java goed geïnstalleerd is, zal de installatie van Greenfoot Java vinden en is Greenfoot klaar voor gebruik. Als dat niet het geval is, zal Greenfoot bij het opstarten alsnog vragen waar Java staat en kun je eerst Java installeren als dat nodig is.

0.2 Het spel *The World of Garp*

Het spel dat we stap voor stap gaan maken, heet '*The World of Garp*', afgeleid van het boek van John Irving '*The World According to Garp*'. Het spel heeft verder niets met het boek te maken.



Garp die aangestuurd wordt door de speler van het spel, moet zo veel mogelijk diamanten verzamelen. Alleen zijn er wel een paar obstakels. Zo is er een moordenaar / dief, Gnomus, die hem achterna zit en Garp wil doden om de diamanten te kunnen stelen.

Als Gnomus Garp heeft gedood, is het spel ten einde. Er zijn ook hinderlagen in de vorm van bommen. Als Garp daar tegenaan loopt, is Garp dood en is het spel geëindigd. Garp heeft gewonnen als hij 25 diamanten heeft verzameld. Ook dan stopt het spel.

Maak kennis met Garp: <http://www.youtube.com/watch?v=wMp3UULdNt8>

Maak kennis met Gnomus: <http://www.youtube.com/watch?v=A1C1NiGWccA>

1. Een allereerste begin

In dit hoofdstuk maak je kennis met Greenfoot. Je maakt twee klassen aan en een aantal objecten. Ook leer je wat een klasse is en wat een object is.

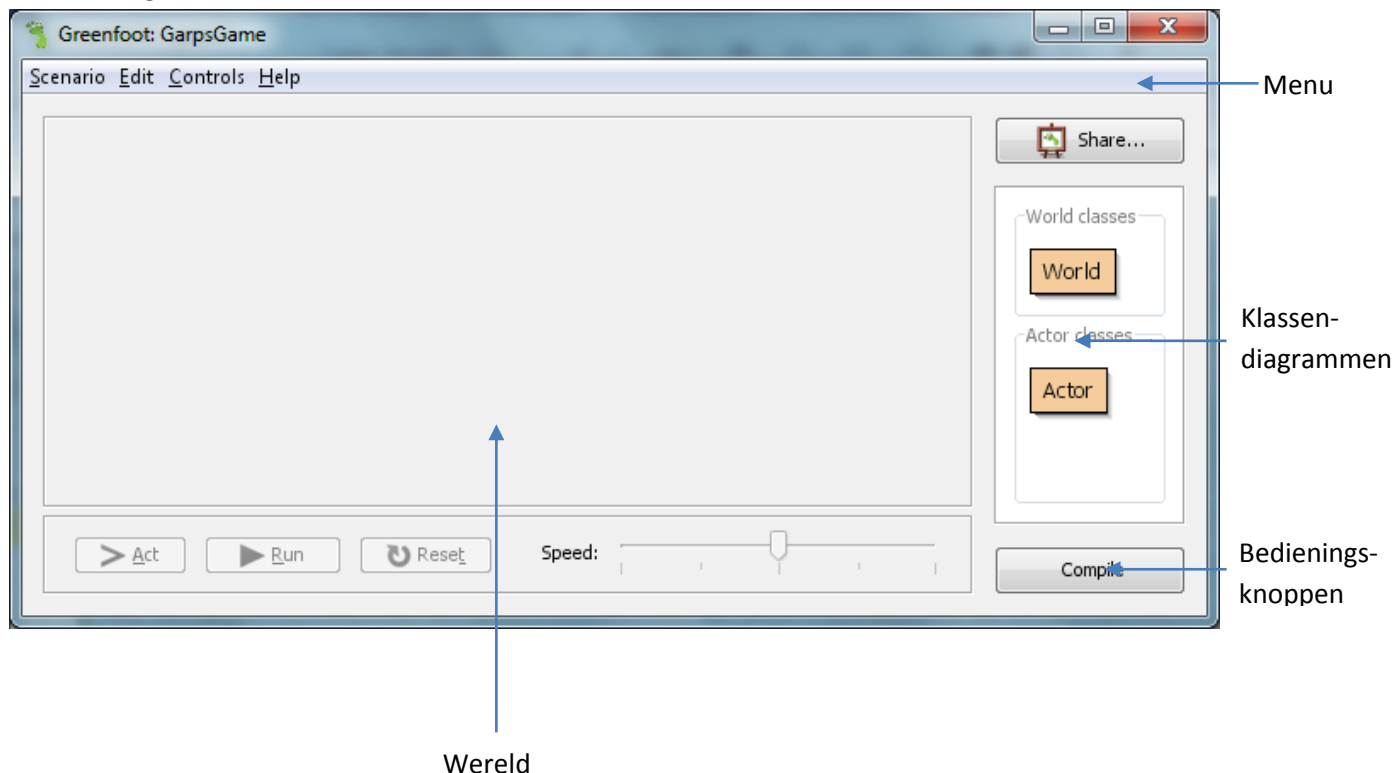
1.1 Eerste kennismaking met Greenfoot

Greenfoot is een IDE, Integrated Development Environment. Dat betekent dat Greenfoot een omgeving aanbiedt, waarin je code kunt schrijven en uitvoeren. Greenfoot zorgt er ook voor, dat de code die je schrijft, vertaald wordt in machinetaal, de taal die de computer begrijpt. Machinetaal bestaat alleen uit enen en nullen. Het vertalen van Java naar machinetaal wordt compileren genoemd.

Als je Greenfoot de allereerste keer opstart, zie je het scherm hiernaast. Kies de optie Create a new scenario en vul de naam GarpGame in.



Als je Greenfoot al eerder hebt opgestart, zie je dit scherm niet. Dan kies je voor een nieuw scenario in het menu Scenario -> New in het scherm hieronder. Vul als nieuw scenario GarpGame in. Je ziet dan het volgende venster:



Helemaal bovenaan staat het menu. Meestal heet het men item aan de linkerkant file of bestand. Nu heet het scenario. Dat is het project waarin een spel gemaakt wordt. Een aantal men items komen verderop ter sprake.

Onderaan staan de bedieningsknoppen. Deze komen later uitgebreider aan de orde:

- > **Act:** Voer een opdracht uit voor elk object in de wereld.
- ▶ **Run:** Start het spel.
- ↺ **Reset:** Herstel de beginsituatie van het spel.
- || **Pause:** Het spel wordt gepauseerd.
- Speed:** Bedieningsbalk voor de snelheid waarmee het spel uitgevoerd wordt.
- Compile:** Compileer de code zodat het spel uitgevoerd kan worden.

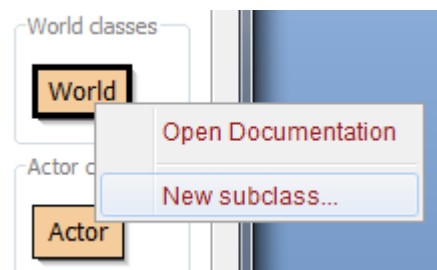
De twee belangrijkste onderdelen van Greenfoot zijn de wereld en de klassendiagrammen. In de wereld speelt het spel zich af. Hier zie je tijdens de uitvoering van het spel wat er gebeurt.

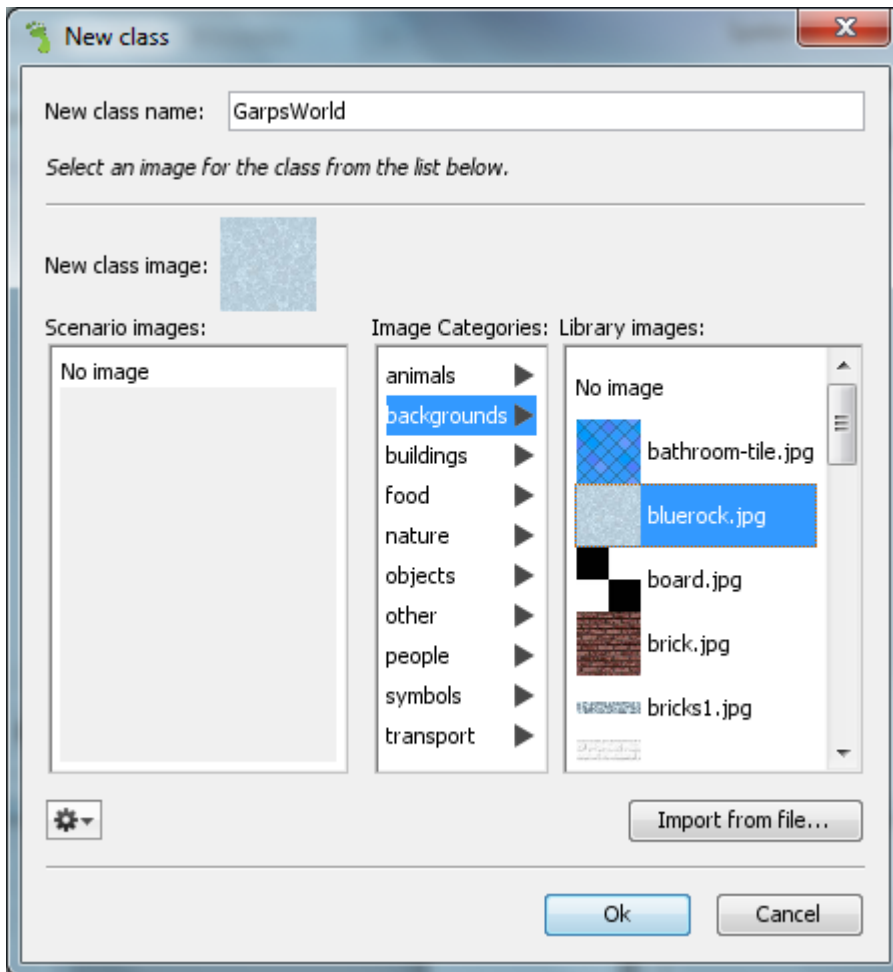
In een spel zijn tenminste twee soorten klassen nodig. De eerste soort klasse gaat over de wereld, de omgeving waarin het spel zich afspeelt. De tweede soort gaat over de actors. Dat zijn de onderdelen van het spel die aanwezig zijn in de wereld en daar iets doen of niets doen (Denk aan Garp en een obstakel). Er is nog een derde soort klasse die nu niet zichtbaar is. Dat zijn de ondersteunende klassen.

1.2 De eerste twee spelelementen

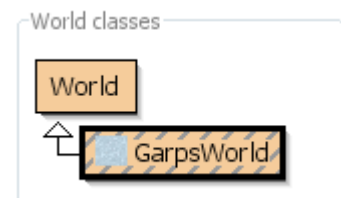
Een spel bestaat altijd uit een wereld waarin het spel zich afspeelt en uit een of meerdere actoren. Om een spel te maken hebben we dus tenminste twee elementen nodig: Een wereld en een actor. Eerst maken we de wereld en daarna de actor die we op de wereld kunnen plaatsen. De wereld is de wereld van Garp en die is nogal blauw en in die wereld leeft Garp, de actor. In Java wordt elk spelelement vastgelegd in een klasse.


Klik met je rechtermuisknop op de klasse **World**. Er verschijnt een menu. Kies voor New Subclass... Het volgende venster verschijnt:





Vul achter New class name: in: **GarpsWorld** (Let op de hoofdletters: de naam van een klasse begint volgens afspraak altijd met een hoofdletter). Kies onder Image Categories voor backgrounds en kies dan onder Library images voor bluerock.jpg (dubbelklikken). Als het goed is, is de knop Ok nu geactiveerd. Klik op de knop Ok en er verschijnt een nieuwe klasse **GarpsWorld** onder de klasse **World**. De rechthoek van de klasse GarpsWorld is gearceerd. Dat betekent dat deze klasse gecompileerd moet worden voordat de code van deze klasse uitgevoerd kan worden. Klik op de knop Compile rechts beneden en vervolgens zie je dat de arcering verdwijnt en dat de achtergrond van **GarpsWorld** in de wereld verschijnt.

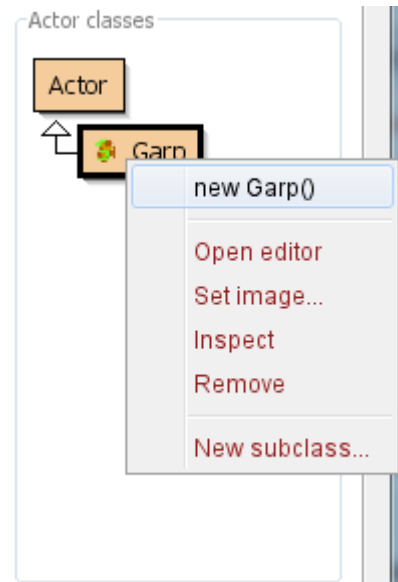


Op dezelfde manier maken we de klasse **Garp**. Alleen klikken we nu met de rechtermuisknop op de klasse **Actor**. Hetzelfde venster als bij **GarpsWorld** verschijnt. De naam voor de klasse is Garp. Alleen onder Animals is geen afbeelding van Garp niet te vinden. We maken nu gebruik van de op knop import from file... en zoeken de afbeelding zelf op. Als we die gevonden hebben, , dan wordt hij opgenomen in het scenario en verschijnt deze in het venster van de klasse. Als we nu op de knop Compile klikken, verdwijnt wel de arcering in de rechthoek van de klasse **Garp**, maar in de **GarpsWorld** verandert er niets.

1.3 Van klasse naar object

Greenfoot weet dat er altijd een wereld nodig is om een spel te spelen, maar Greenfoot weet niet welke actoren wanneer nodig zijn in het spel. Greenfoot weet dus wel dat er van de klasse `GarpsWorld` een object of een instantie gemaakt moet worden en dat die zichtbaar gemaakt moet worden. Greenfoot weet dus niet dat er van de klasse `Garp` een object gemaakt moet worden. Dat gaan we eerst met de hand doen. Later zullen we dat automatiseren.

Klik met de rechter muisknop op de klasse **Garp**. Er verschijnt nu een menu met een menu item dat we nog niet gezien hebben: `new Garp()`. Dit is eigenlijk een Java-opdracht. Het sleutelwoord `new` betekent dat je van de klasse **Garp** een object of instantie maakt. Als je op dat menu item klikt, zie je de afbeelding van de klasse **Garp** en kun je die op de wereld plaatsen. Je kunt hetzelfde nog een aantal keer doen en iedere keer kun je `Garp` op een andere plek in de wereld plaatsen. Dit kun je doen totdat het geheugen van de computer vol is. Je hebt nu een aantal objecten van de klasse **Garp** op de wereld gezet.



Opdracht 1:

Plaats tenminste tien `Garps` in de wereld.

Een klasse is een algemene beschrijving van een object. Wij gebruiken de klasse **Garp** om daarvan een object te maken. Een object wordt in de wereld geplaatst, een klasse niet. Je hebt altijd een klasse nodig om een object te maken. Je kunt meerdere objecten van dezelfde klasse maken.

Je kunt het vergelijken met de tekening van een huis die gemaakt is door een architect. Dat is dan de klasse huis. De bouwvakkers maken vervolgens dat huis. Dat huis dat gebouwd is, is het object huis. De bouwvakkers kunnen die tekening van de architect gebruiken om op een andere plek hetzelfde huis te bouwen. Dan hebben ze nog een object huis gemaakt.

1.4 Zij doen niets

Een aantal `Garps` staan nu in de wereld. Als we het spel activeren –klik op de knop `act` en op de knop `run`– dan zien we dat er niets gebeurt. De `Garps` blijven op hun plek staan. Dat komt, omdat we nog geen code hebben toegevoegd aan de klassen en de objecten niet weten wat zij moeten doen. In het volgende hoofdstuk gaan we daar iets aan doen. Als we op de knop `reset` klikken, verdwijnen de `Garps` van de wereld.

VAKTAAL

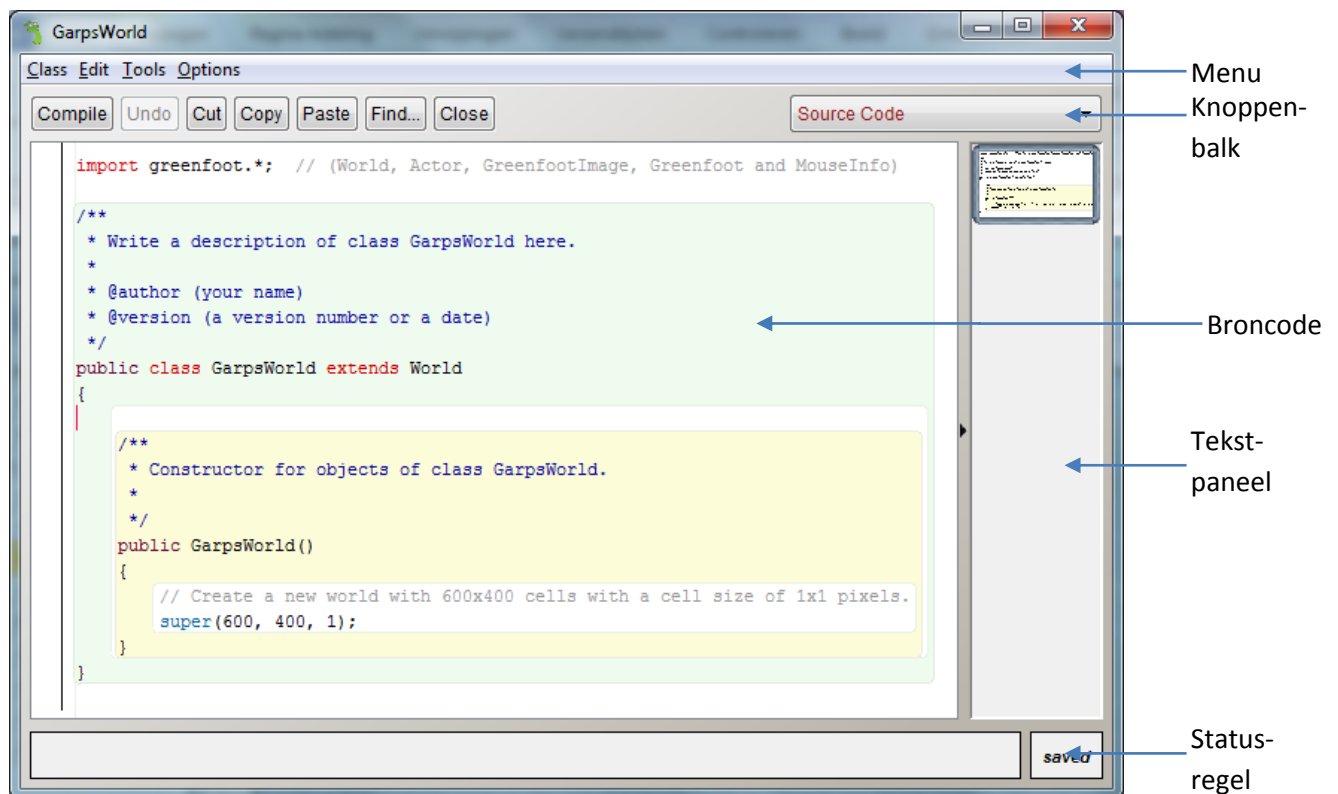
- **NEW:** Sleutelwoord in Java, waarmee een nieuw object van een klasse wordt gemaakt. Met *new Garp()* wordt een nieuw object gemaakt van de klasse **Garp**.
- **KLASSE:** Een klasse is een algemene beschrijving van iets uit de werkelijkheid, bijvoorbeeld van een patiënt.
- **SUBKLASSE:** een klasse die de eigenschappen en methoden van een andere klasse erft en dus kan gebruiken.
- **OBJECT:** een object is een exemplaar van een klasse, bijvoorbeeld patiënt Jan is een object van patiënt.
- **INSTANTIE:** Een instantie is hetzelfde als een object.
- **INTEGRATED DEVELOPMENT ENVIRONMENT:** Een omgeving waarin de programmeur code kan schrijven en bewerken, waarin de code wordt gecompileerd en uitgevoerd. (Afkorting: IDE)
- **EDITOR:** een tekstverwerker waarin de programmeur code schrijft of bewerkt
- **COMPILER:** software die door de programmeur geschreven code vertaalt in machinetaal.
- **MACHINETAAL:** de taal in enen en nullen die de computer begrijpt en die de computer kan uitvoeren.

2. Besturing en beweging

Garp wordt bestuurd door de speler. Gnomus die achter Garp aan zit, beweegt vanzelf. In dit hoofdstuk gaan we eerst de code bekijken die Greenfoot al gegenereerd heeft en daarna gaan we code toevoegen waardoor Garp en Gnomus gaan bewegen. Gnomus laten we vanzelf bewegen, Garp wordt door de speler aangestuurd. Tot slot leren we ook, hoe Greenfoot een spel aanstuurt.

2.1 De code van Greenfoot

De code van een klasse kun je zichtbaar maken door dubbel te klikken op de rechthoek van de klasse. Ook kun je met de rechter muisknop het menu zichtbaar maken en voor het item Open editor kiezen. In beide gevallen opent de editor zich met de code die bij de klasse hoort. Open nu de editor met de code van de klasse **GarpsWorld**.



De knoppen en de menu items komen verderop vanzelf aan de orde. Voor ons is het deel van het venster waarin de broncode staat van belang. Daarin kunnen we de tekst van de code bewerken. Daarnaast staat het tekstpaneel, waarin door de afgeronde rechthoek wordt aangegeven welk deel van de broncode zichtbaar is. Op de statusregel wordt in het rechter deel verteld of de code gewijzigd is en of de code opgeslagen is. In de lange rechthoek links op de statusregel komen foutmeldingen te staan.

De eerste regel in de code bevat een import:

```
import greenfoot.*; // (World, Actor, GreenfootImage, Greenfoot and MouseInfo)
```

Dit betekent dat alle klassen die in de directory `greenfoot` staan, onderdeel worden van het scenario. Het sterretje betekent alle klassen. Deze klassen kunnen we op dezelfde manier gebruiken als klassen die we zelf hebben gemaakt. Welke klassen dat zijn staat erachter in het commentaar. Achter twee forward slashes staat altijd commentaar. Dat is eigenlijk geen code, maar is een toelichting op de code voor de programmeur. Java slaat commentaar over.

Opdracht 2:
Bekijk de javadocs van de klassen `World` en `Actor`.

Meer informatie over de klassen van Greenfoot kun je vinden op:

<http://www.greenfoot.org/files/javadoc/>. Dat zijn zogenaamde Javadocs die gebruikt worden om de klassen in Java te documenteren.

De regels 3 tot en met 8 bevatten ook commentaar. Daaronder staat de klassedefinitie:

```
public class GarpsWorld extends World
```

`public` betekent dat de klasse altijd aangeroepen kan worden. Daarachter staat dat het om een klasse gaat (Er zijn andere mogelijkheden die hier niet besproken worden). Dan volgt de naam van de klasse: **GarpsWorld**. De naam van een klasse begint standaard met een hoofdletter. Dat wordt gedaan om een onderscheid te maken tussen klassen, objecten en andere variabelen. Na de naam volgt `extends World`: De klasse **GarpsWorld** is een uitbreiding van de klasse **World**. We kunnen alles van de klasse **World** gebruiken alsof het in de klasse **GarpsWorld** staat. Anders gezegd: De subklasse **GarpsWorld** erft alle eigenschappen van de superklasse **World**.

Op de volgende regel staat alleen een openingsaccolade. Als je daarachter gaat staan met je cursor, zie je dat de onderste sluitaccolade geselecteerd wordt weergegeven. Deze twee accolades horen dus bij elkaar. Tussen deze twee accolades staat de body of het codeblok van de klasse **GarpsWorld**. Alle regels tussen deze twee accolades horen bij de klasse **GarpsWorld**.

In deze klasse staat een methode:

```
public GarpsWorld()
{
    // Create a new world with 600x400 cells with a cell size of 1x1 pixels.
    super(600, 400, 1);
}
```

Dit is een bijzondere methode, want hij heeft dezelfde naam, inclusief hoofdletters, als de klasse waartoe hij behoort. Deze methode wordt een constructor genoemd. Een constructor wordt aangeroepen als van een klasse een object gemaakt wordt, bijvoorbeeld met `new GarpsWorld()`; Hier betekent `public` dat deze constructor van buiten de klasse aangeroepen kan worden. Een constructor zorgt ervoor dat het object of instantie begint met de juiste instellingen.

Ook onder deze regel staat een openingsaccolade. Daar hoort een sluitaccolade bij. Als je achter een van deze accolades gaat staan, zie je welke andere accolade erbij hoort. Tussen deze accolades staat

de body of codeblok van de constructor. In de codeblok van de klasse staat een andere codeblok van de constructor.

In het codeblok staat maar een opdracht: `super(600, 400, 1);` De methode `super` roept de constructor van de klasse erboven op, in dit geval de constructor van de klasse **World**. Deze klasse is de superklasse van de subklasse **GrapsWorld**. De methode kent drie parameters. Een parameter is een waarde die meegegeven wordt aan een methode of constructor en waarmee die methode of constructor iets doet. De eerste twee parameters, 600 en 400, zijn de breedte en de hoogte van de wereld. De derde parameter is de grootte van de cellen op de wereld. In dit geval is een cel gelijk aan een pixel. Bij bordspelen kan een cel 10 pixels groot zijn.

Opdracht 3:

Maak de breedte en de hoogte van de wereld 100 pixels groter. Pas ook het commentaar erboven aan.

De code van de klasse **Garp** –dubbelklik op de rechthoek van de klasse **Garp**- lijkt heel veel op de code van de klasse **GarpsWorld**. Er zijn twee verschillen.

- In de code van de klasse **Garp** staat geen constructor. Die kunnen we altijd nog maken.
- Er staat in de code van **Garp** de methode `act()`:

```
public void act()
{
    // Add your action code here.
}
```

In deze methode staan geen opdrachten. Er staat wel in het commentaar dat je hier de code moet toevoegen. De methode wordt begrensd door een begin- en een eindaccolade. De opdrachten die we daartussen gaan zetten, worden het codeblok van de methode genoemd. Telkens wanneer de methode wordt aangeroepen, worden opdrachten in het codeblok van de methode uitgevoerd.

De methode `act()` is in Greenfoot een bijzondere methode. Zodra je op de knop **> Act** klikt, zal Greenfoot de code in `act()` een keer uitvoeren. Greenfoot roept de methode `act()` aan om de code in het codeblok uit te voeren. Als je op de knop **► Run** klikt, zal Greenfoot steeds weer de code in de methode `act()` uitvoeren totdat het spel stopt of totdat je op de knop **↺ Reset** klikt. Nu is ook duidelijk waarom de Garps op de wereld niets deden: Er staat geen code in de methode `act()`. Het wordt dus tijd om code aan de methode `act()` toe te voegen.

2.2 Compileren

De code is gewijzigd, maar er gebeurt niets. Alleen zie je dat het rechthoek van de klasse **GarpsWorld** gearceerd weergegeven wordt. De broncode moet nog gecompileerd worden.

De computer begrijpt niets van Java. Een computer begrijpt alleen machinetaal die uit enen en nullen bestaat. Dat betekent dat jouw broncode eerst vertaald moet worden voordat de code uitgevoerd kan worden. Dat doe je door op de knop **Compile** te drukken. Dan wordt de code vertaald en kan

Greenfoot de code uitvoeren. Dat zie je aan de wereld die groter wordt en aan de arcering die verdwijnt.

2.3 Het rechte pad van Garp

We gaan ons eerst richten op de aansturing van Garp door middel van de toetsen op het toetsenbord. Daarvoor gaan we nu de code van Garp aanpassen in de methode *act()*. Met *Greenfoot.isKeyDown()* kan opgevraagd worden welke toetsen zijn ingedrukt. Het gaat om de pijltjestoetsen. Dus van deze moeten we de namen weten: “left”, “right”, “up” en “down”. We beginnen met een toets, bijvoorbeeld die naar rechts. De volgende code kunnen we dan schrijven in de methode *act()*:

```
if(Greenfoot.isKeyDown("right")) {
    move(5);
}
```

Opdracht 4:

Pas eventueel de code aan en voer deze uit.

Garp loopt nu naar rechts en stopt aan de rand van de wereld.

Het if-statement voert de opdrachten in het codeblok (Alle opdrachten tussen de beginaccolade en eindaccolade die bij elkaar horen) alleen uit als de voorwaarde tussen de ronde haken achter het if-statement waar is. Dus in dit geval wordt de opdracht *move(5)* alleen uitgevoerd als je op de toets → drukt. De methode *Greenfoot.isKeyDown("right")* geeft dan *true* terug. Als je op een andere toets drukt, doet Garp niets. De methode *Greenfoot.isKeyDown("right")* geeft dan *false* terug.

Opdracht 5:

Zet Garp aan de linkerkant van de wereld en druk niet allen op de toets →, maar druk ook op andere toetsen en kijk hoe Garp reageert.

Teruglopen kan dus met het pijltje naar links:

```
if(Greenfoot.isKeyDown("left")) {
    setRotation(180);
    move(5);
}
```

Garp loopt nu inderdaad terug, maar wel op zijn kop. Dat komt door de opdracht *setRotation(180);*. Door deze opdracht wordt de afbeelding die geassocieerd is aan het object, 180 graden gedraaid en dus staat Garp op zijn kop. Dat gaan we later aanpassen. Door die rotatie wordt ook de richting waarin Garp zich beweegt, aangepast, nu van rechts naar links. Daarvan gaan we gebruik maken om Garp naar boven en beneden te laten lopen:

```

if(Greenfoot.isKeyDown("up")) {
    setRotation(-90);
    move(5);
}
if(Greenfoot.isKeyDown("down")) {
    setRotation(90);
    move(5);
}

```

Alleen als we Garp naar rechts willen laten lopen, dan gaat dat niet en loopt hij dezelfde kant op als daarvoor. We moeten nog `setRotation(0);` toevoegen om te vertellen dat Garp naar rechts gaat. Nu gaat Garp alle kanten op.

Opdracht 6:

Pas de code aan zoals boven beschreven en voer de code uit. Kijk wat Garp doet.

2.4 Garp rechtop

Om Garp naar links niet op zijn kop te laten lopen hebben we twee afbeeldingen van Garp nodig: Eentje waarin Garp naar rechts kijkt en eentje waarin Garp naar Links kijkt. De eerste afbeelding gebruiken we als Garp naar rechts, naar boven en naar beneden loopt. De tweede gebruiken we als Garp naar links loopt.

In de constructor gaan we de afbeeldingen klaar zetten voor gebruik in de methode `act()`. Een constructor heeft dezelfde naam als de klasse, dus in dit geval `Garp()`. Om de afbeeldingen klaar te kunnen zetten, hebben we twee variabelen nodig van het type `GreenfootImage`. Dat doen we met behulp van globale of instantievariabelen die in alle methodes van de klasse gebruikt kunnen worden. De declaratie van deze variabelen staan dan ook niet in een methode maar juist daarbuiten, standaard bovenaan in de klasse:

```

public class Garp extends Actor
{
    private GreenfootImage imageLeft;
    private GreenfootImage imageRight;

    public void act()
    {
        ...
    }
}

```

Private betekent dat de variabele alleen benaderd kan worden vanuit alle methodes binnen deze klasse en niet vanuit methodes van een andere klasse. `GreenfootImage` is het type van de variabele. Dan weet de compiler hoeveel geheugen er gereserveerd moet worden voor de variabele. Hier is het type van de variabele dus de klasse `GreenfootImage`. De naam van de eerste variabele is `imageLeft`, de tweede heet `imageRight`.

Nu gaan we de constructor maken:

```
public class Garp extends Actor
{
    private GreenfootImage imageLeft;
    private GreenfootImage imageRight;

    public Garp()
    {
        ...
    }

    public void act()
    {
        ...
    }
}
```

De constructor wordt maar een keer uitgevoerd, namelijk als er een object gemaakt wordt, dus als de klasse geïnstancieerd wordt. We gebruiken daarvoor het sleutelwoord *new*. In de constructor gaan we de afbeeldingen van schijf lezen en in het geheugen klaar zetten voor gebruik. Dat doen we als volgt:

```
imageLeft = new GreenfootImage("GarpLeft.png");
imageRight = new GreenfootImage("GarpRight.png");
```



Vervolgens kunnen we een van de twee afbeeldingen associëren met de klasse door `setImage()` te gebruiken:

```
setImage(imageRight);
```

Lezen van schijf kost veel tijd. Vandaar dat we de twee afbeeldingen klaar zetten zodat we dat niet telkens in de methode `act()` hoeven te doen. De methode `act()` wordt steeds opnieuw door Greenfoot aangeroepen en daarom moet deze methode snel zijn. Nu hoeft `act()` alleen in het geheugen de afbeeldingen te verwisselen en dat is sneller dan steeds weer inlezen van een afbeelding.

We gaan nu de methode `act()` aanpassen, eerst voor de toets ←:

```
if(Greenfoot.isKeyDown("left")) {
    if(getImage() == imageRight) {
        setImage(imageLeft);
    }
    setRotation(0);
    move(-5);
}
```

```
}

```

Met `getImage()` achter het if-statement wordt opgevraagd welke afbeelding geassocieerd is aan het object. Dan wordt met behulp van een operator (`==`) vergeleken of `imageRight` met het object geassocieerd is. Als dat zo is, dan wordt `imageLeft` aan het object geassocieerd.

Achter het if-statement staat altijd een controle of voorwaarde. Daarin wordt gecontroleerd of een vergelijking waar is of onwaar (*true* of *false*). Java heeft een aantal logische operatoren waarmee de inhoud van twee variabelen of de waarden zelf vergeleken kunnen worden achter het if-statement:

<code>==</code>	is gelijk aan	<code>!=</code>	is ongelijk aan
<code><</code>	kleiner dan	<code>></code>	groter dan
<code><=</code>	kleiner dan of gelijk aan	<code>>=</code>	groter dan of gelijk aan

Als de vergelijking waar is, dan wordt het codeblok onder *if-statement* uitgevoerd. Als de vergelijking niet waar is, wordt het codeblok onder *if* overgeslagen en gaat het programma verder met het eerste statement na het codeblok.

Na het if-statement zijn er nog twee wijzigingen aangebracht. De parameter die met `setRotation()` wordt meegegeven is van 180 verandert in 0, want Garp staat al de juiste kant op. Alleen om van rechts naar links te bewegen, is de parameter die met `move()` wordt meegegeven, gewijzigd in -5 in plaats van 5.

Opdracht 7:

Pas de code aan zoals boven is beschreven en kijk wat Garp doet. Stuur hem met de pijltjestoetsen maar eens alle kanten uit.

Je ziet: het gaat nog niet helemaal goed. Als je eenmaal op de toets ← hebt gedrukt, gaat Garp steeds achteruit en is hij niet meer vooruit te branden totdat je weer op de toets ← hebt gedrukt. Dat komt omdat alleen de afbeelding van Garp naar links wordt gebruikt. Bij de andere toetsen moet nu alleen nog de afbeelding naar rechts met het object geassocieerd worden. Dat doen we door bij elke toets de volgende code te zetten:

```
if(getImage() == imageLeft) {
    setImage(imageRight);
}
```

Opdracht 8:

Pas de code aan zoals boven is beschreven en voer de code uit. Kijk wat Garp doet.

Opdracht 9:

Zet nu tenminste vijf Garps op de wereld en laat de code uitvoeren. Kijk hoe de vijf Garps reageren en probeer van de vijf Garps een Garp te maken door ze over elkaar heen te zetten.

2.5 De eerste stappen op het dievenpad

We gaan nu Gnomus de eerste stappen op de wereld zetten. Daarvoor moeten we eerst de klasse **Gnomus** aanmaken.



Opdracht 10:

Maak nu de klasse **Gnomus** aan op dezelfde wijze als je de klasse **Garp** hebt gemaakt. Gebruik de daarbij behorende afbeelding.

De code van de klasse **Gnomus** ziet er hetzelfde uit als die van de klasse **Garp**.

We gaan Gnomus zijn eerste stappen laten doen. Daarvoor gebruiken we de methode `move()`. We wijzigen de methode `act()` in de klasse **Gnomus** aldus:

```
public void act()
{
    move(5);
}
```

De methode `move` laat Gnomus naar rechts lopen. De methode `move` kent een parameter. Het getal geeft het aantal pixels aan waarmee Gnomus verplaatst wordt. Aan het einde van de wereld stopt Gnomus met bewegen.

Opdracht 11:

Wijzig de code van de klasse **Gnomus** zoals hierboven beschreven. Zet een dief op de wereld en druk op de knop ► **Run**.

Opdracht 12:

Druk op de knop ↺ **Reset**. Zet weer een dief op de wereld en druk nu op de knop > **Act** en kijk wat er gebeurt. Druk nog een paar keer op de knop > **Act**.

Opdracht 13:

Laat Gnomus heel langzaam van links naar rechts lopen door de code aan te passen.

Opdracht 14:

Laat Gnomus een sprintje trekken van links naar rechts door de code aan te passen.

Gnomus kan ook van rechts naar links lopen. Dat doe je door een negatief getal als parameter aan `move` mee te geven.

Opdracht 15:

Laat Gnomus van rechts naar links lopen, eerst gewoon, dan heel langzaam en daarna laat je Gnomus een sprintje van rechts naar links trekken.

Gnomus loopt nu achteruit. Dat laten we nu zo, daar gaan we later iets aan doen. Het gaat nu om de beweging zelf.

Als Gnomus aan de rand van de wereld komt, staat hij stil. Eerst laten we Gnomus draaien. Dat doen we op de volgende manier:

```
public void act()
{
    setRotation(90);
}
```

Nu staat in de methode *act()* de opdracht *setRotation*. Als parameter geef je mee het aantal graden dat de afbeelding moet draaien, in dit geval 90 graden.

Opdracht 16:

Pas de code van de klasse Gnomus aan en plaats een dief op de wereld. Laat vervolgens de code uitvoeren.

Je ziet dat Gnomus in een keer 90 graden draait en daarna stil blijft staan. Dus hij rekent met het beginpunt 0. We moeten bij het draaien van de huidige situatie uitgaan, namelijk dat Gnomus al 90 graden is gedraaid. Die huidige situatie vragen we op met *getRotation()*. In plaats van *setRotation(90)*; zetten we in de code *setRotation(getRotation() + 90)*; We tellen bij die 90 graden de huidige situatie op.

Opdracht 17:

Pas de code van de klasse Gnomus aan en plaats een dief op de wereld. Laat vervolgens de code uitvoeren. Pas eventueel de snelheid van de uitvoering aan als je niet goed kunt zien wat er gebeurt.

Je ziet dat Gnomus nu om zijn as draait. We kunnen de opdrachten *move()* en *setRotation()* natuurlijk ook combineren:

```
public void act()
{
    setRotation(getRotation() + 90);
    move(5);
}
```

Opdracht 18:
Pas de code aan en voer hem uit.

Ook nu draait Gnomus om zijn as. Dat komt omdat met de opdracht *setRotation()* ook de richting waarin Gnomus zich beweegt, verandert. Als we het aantal graden verminderen en daar bijvoorbeeld 10 van maken, dan zien we dat Gnomus om zijn linker onderpunt draait. Maken we het aantal graden nog kleiner dan draait Gnomus in een rondje, hoe lager het aantal graden, hoe groter de cirkel, waarin Gnomus draait.

Opdracht 19:
Pas de code aan en laat Gnomus draaien met 10, 5 graden en 1 graad. Zorg er in het laatste geval voor dat Gnomus een rand van de wereld raakt en kijk wat er dan gebeurt.

Opdracht 20:
Wijzig de code van de klasse **Gnomus** dusdanig dat Gnomus naar links loopt en bij de rand van de wereld stil blijft staan.

2.6 Het einde van de wereld

Nu Gnomus van richting kan veranderen, moeten we bepalen wanneer Gnomus bij de rand van de wereld is en hem daar laten draaien, zodat hij verder kan gaan. Greenfoot rekt vanuit het midden van de afbeelding. De afbeelding van Gnomus is 80x80, vierkant dus. We moeten dus met de helft, 40 dus, gaan rekenen.

Boven en links zijn de randen gemakkelijk te vinden, namelijk in beide gevallen de waarde 0. Dat is in elke wereld zo. Om het probleem van de randen van de wereld goed te kunnen oplossen maken we daar een aparte methode van met de naam *atWorldEdge*. (Let op: de naam van een methode begint met een kleine letter) De methode kent geen parameters, vandaar dat er niets tussen de ronde haken staat. Deze methode vertelt ons of Gnomus bij een rand van de wereld is of niet. Er zijn dus twee mogelijkheden: *true* (Gnomus is bij de rand van de wereld) of *false* (Gnomus is niet bij een rand van de wereld). Als er twee mogelijkheden zijn, spreken we van een *boolean*. Dus geeft deze methode als antwoord een boolean:

```
protected boolean atWorldEdge()
{
    return false;
}
```

Tussen de accolades komen de opdrachten te staan. De enige opdracht is nu: *return false;*. We hebben namelijk nog niet geconstateerd dat Gnomus aan een rand van de wereld staat. Boven dit eerste statement gaan we kijken of Gnomus bij een rand van de wereld is. Eerst de linker en de bovenkant. Met *getX()* kunnen we de plek opvragen waar Gnomus is ten opzichte van de linkerrand. De methode geeft een geheel getal of een integer terug, een *int* in Javatermen. Deze waarde

bewaren we in variabele `x` die ook van het type `int` is. Deze variabele hebben we boven aan in de methode gedeclareerd, dat wil zeggen we hebben aan de compiler verteld dat we van plan zijn een variabele van het type `int` te gebruiken met de naam `x`. De compiler kan dan ruimte in het geheugen reserveren. Zo'n variabele is alleen in deze methode bereikbaar, daarbuiten niet. Daarom heet zo'n variabele een lokale variabele. Daarna controleren we of de waarde 0 is, zo ja, dan is Gnomus aan de linkerkant van de wereld.

```
public boolean atWorldEdge()
{
    int x;           //declaratie

    x = getX();     //Vraag op waar Gnomus is ten opzichte van de linkerrand
    if(x == 0) {    // Is x gelijk aan 0?
        return true; //Geef true terug
    }
    return false;   //Dief staat niet aan een rand van de wereld dus false teruggeven
}
```

Toch blijft Gnomus aan de linkerkant nog steeds staan. Dat komt omdat in `act()` alleen de opdracht `move(-5)` staat en nog geen aanroep staat van de methode `atWorldEdge()` en zo ja, wat Gnomus dan moet doen, namelijk omkeren:

```
public void act()
{
    move(-5);
    if(atWorldEdge()) { //Aan de rand van de wereld?
        setRotation(getRotation() + 180); //zo ja, ga op je schreden terug
    }
}
```

Gnomus verdwijnt aan de linkerkant nog steeds voor de helft. Dat komt omdat Greenfoot rekent met het midden van de afbeelding. Dus we moeten de breedte van de afbeelding opvragen, die door twee delen en dat als de rand van de wereld beschouwen. Als Gnomus dus op een plek staat die voor de helft van de breedte van de afbeelding van de rand van de wereld is of minder dan de helft, dan moet de methode `true` teruggeven.

```
public boolean atWorldEdge()
{
    int x, i;           //declaratie van i toegevoegd voor de helft van breedte van de afbeelding

    x = getX();
    i = getImage().getWidth() / 2; //Vraag de breedte van de afbeelding op en deel deze door twee
    if(x <= i) {       //Aan de linkerkant van de wereld
        return true;
    }
}
```

```

    return false;
}

```

Opdracht 21:

Vul de methode aan met de code die kijkt of Gnomus aan de bovenrand van de wereld staat. Aanwijzing: gebruik *getY()* om de positie op te vragen en bewaar die in de variabele *y*.

De rechter en de onderkant zijn moeilijker, want die kunnen per wereld anders zijn. In dit geval zijn de waarde 700 en 500, maar als we deze waarden wijzigen, dan klopt ons programma niet meer en moeten we opeens veel meer aanpassen. Daarom vragen we met *getWorld()* de wereld op en met *getWidth()* de breedte van de wereld. Vervolgens moet daarvan de helft van de breedte van de afbeelding afgetrokken worden. Voeg onderstaande code aan de methode *atWorldEdge()* toe:

```

wx = getWorld().getWidth() - i;
if(x >= wx) {
    return true;
}

```

Uiteraard moet *wx* ook gedeclareerd worden boven aan in de methode.

Opdracht 22:

Test je code door de code te compileren en uit te voeren. Als het goed is gaat Gnomus heen en weer.

Opdracht 23:

Zet een aantal dieven op de wereld op willekeurige plekken en druk op ► **Run**.

Je ziet dat de dieven van richting veranderen op het moment dat zij aan de linker- of rechterraand van de wereld zijn. Dat gebeurt bij elke dief op een ander moment. Objecten passen zich aan aan de situatie van dat moment maar blijven reageren op de manier zoals in de klasse is beschreven.

Opdracht 24:

Vul de methode aan met de code die kijkt of Gnomus aan de onderrand van de wereld staat. Aanwijzing: gebruik *getHeight()* om de hoogte van de wereld op te vragen en bewaar die in de variabele *wy*.

Opdracht 25:

Test je code maar laat in plaats van 180 graden te draaien Gnomus 225 graden draaien.

2.7 Het geheim van het dievenpad

Het pad dat Gnomus nu volgt, is erg voorspelbaar. Daarin gaan we nu verandering in aan brengen door gebruik te maken van de randomgenerator van de computer. Met de randomgenerator kan op elk moment een willekeurig getal worden opgeroepen. Dat doen we door gebruik te maken van `Greenfoot.getRandomNumber()`. Daarmee is de kans op elk getal even groot. Het gaat erom Gnomus willekeurig verschillende richtingen op te sturen. Dat kunnen we allereerst doen door het getal 225 te vervangen in de code door `Greenfoot.getRandomNumber(180)` en kijken wat er gebeurt. Het betekent dat we een willekeurig getal terugkrijgen vanaf 0 tot 180.

Opdracht 26:

Pas de code aan door 225 te vervangen door `Greenfoot.getRandomNumber(180)`.

Nu gaat Gnomus steeds dezelfde kant uit. Dat veranderen we door de plus-teken te vervangen door min-teken. In 50% van de gevallen een plus en in 50% van de gevallen een minteken. Dat lossen we als volgt op:

```
if(atWorldEdge()) {
    if(Greenfoot.getRandomNumber(100) < 50) {                //50% kans
        setRotation(getRotation() + Greenfoot.getRandomNumber(180));
    }
    else {
        setRotation(getRotation() - Greenfoot.getRandomNumber(180));
    }
}
```

Opdracht 27:

Pas de code aan en voer deze uit. Kijk goed wat er gebeurt.

Door een willekeurig getal tussen de 0 en de 100 op te vragen, is er 50% kans dat het getal kleiner dan 50 is. Als het getal kleiner dan 50 is, gaat Gnomus naar rechts, een willekeurig aantal graden tussen 0 en 180. In het andere geval (*else*) dus als het getal groter dan 49 is, dan gaat Gnomus een willekeurig aantal graden naar links.

Gnomus is aan de rand van de wereld soms erg onzeker en blijft maar rondjes draaien totdat hij uiteindelijk weet welke kant hij op wil. Dat kunnen we voorkomen door Gnomus eerst een stap terug te laten doen als hij bij een rand van de wereld is: `move(-5)`. Deze regel voegen we toe onder de regel `if(atWorldEdge())` {.

Tussen de randen gaat Gnomus nog steeds rechtdoor. We willen in 2 procent van de gevallen dat Gnomus naar rechts gaat en in 2 procent van de gevallen dat hij naar links gaat. Dat kunnen we op dezelfde manier doen, alleen dan is het getal kleiner dan 2 of groter dan 98. Als het getal kleiner dan 2 is, gaat Gnomus naar rechts en als het getal groter is dan 98 gaat Gnomus naar links. Ook het aantal graden dat Gnomus draait, is willekeurig. Er zijn twee mogelijkheden: Gnomus is aan een rand van de

wereld of Gnomus is ergens anders in de wereld. In het eerste geval weet Gnomus wat hij moet doen, in het tweede nog niet. De structuur van de code ziet er als volgt uit:

```
if(atWorldEdge()) {
    ... //eerste codeblok
}
else {
    ... //tweede codeblok
}
```

Het eerste codeblok hebben we al gevuld met code en daarom weet Gnomus wat hij moet doen als hij aan een rand van de wereld is. In het tweede codeblok komt dus de code te staan als Gnomus niet aan de rand van de wereld staat maar zich ergens anders op de wereld bevindt:

```
random = Greenfoot.getRandomNumber(100);
if(random < 2) {
    setRotation(getRotation() + Greenfoot.getRandomNumber(180));
}
if(random > 98) {
    setRotation(getRotation() - Greenfoot.getRandomNumber(180));
}
```

Omdat de kans 2% naar rechts of 2% naar links is, slaan we het willekeurig getal op in de variabele *random* van het type *int*. Daarna controleren we of het getal kleiner is dan 2 en zo ja dan draait Gnomus een willekeurig aantal graden naar rechts met een maximum van 180. Uiteraard moeten we boven in het codeblok van de methode *act()* de variabele *random* declareren.

De code in de methode *act()* is dan:

```
public void act()
{
    int random;

    move(5); //ga 5 pixels vooruit
    if(atWorldEdge()) { //Sta je aan de rand van de wereld
        move(-5); //Doe een stap terug
        if(Greenfoot.getRandomNumber(100) < 50) { //Welke kant op: rechts of links
            setRotation(getRotation() + Greenfoot.getRandomNumber(180));
        }
        else {
            setRotation(getRotation() - Greenfoot.getRandomNumber(180));
        }
    }
    else { //Als je niet aan de rand van de wereld bent
        random = Greenfoot.getRandomNumber(100); //getal tussen 0 en 100
    }
}
```

```
if(random < 2) {                               //2% kans naar rechts
    setRotation(getRotation() + Greenfoot.getRand omNumber(180));
}
if(random > 98) {                               //2% kans naar links
    setRotation(getRotation() - Greenfoot.getRandomNumber(180));
}
}
}
```

Opdracht 28:

Pas eventueel de code aan en voer deze uit.

Opdracht 29:

Voer nogmaals opdracht 9 uit: Zet nu tenminste vijf Garps op de wereld en laat de code uitvoeren. Kijk hoe de vijf Garps reageren en probeer van de vijf Garps een Garp te maken door ze over elkaar heen te zetten.

Opdracht 30:

Maak nu een stuk of vijf, zes objecten van de klasse **Gnomus** en voer de code uit. Kijk goed wat er gebeurt.

Bij opdracht 29 gingen de vijf Garps in eerste instantie dezelfde kant op totdat een van hun het einde van de wereld bereikte. Het lijkt alsof elk object van de klasse **Gnomus** zich daarentegen anders gedraagt. Dat is natuurlijk niet waar. Doordat elk object andere willekeurige getallen krijgt dan een ander object, gaat ieder object een eigen weg. Verder houden zij zich aan de beschrijving zoals die in de klasse **Gnomus** staat.

VAKTAAL

- **INSTANCIËREN:** Van een klasse een object maken.
- **BRONCODE:** broncode is de code zoals die is geschreven door de programmeur.
- **CODEBLOK:** opdrachten die tussen een begin- en een eindaccolade staan, die bij elkaar horen. Een codeblok kan in een andere codeblok voorkomen.
- **BODY:** hetzelfde als een codeblok.
- **METHODE:** een groep bij elkaar horende opdrachten die gezamenlijk een bepaalde taak uitvoeren. Een methode heeft een naam waarmee hij aangeroepen wordt.
- **CONSTRUCTOR:** een methode die aangeroepen wordt als van de klasse een object gemaakt wordt. Een constructor wordt een keer aangeroepen en heeft dezelfde naam als de klasse. In de constructor worden de beginwaarden ingesteld.
- **PARAMETER:** een parameter is een waarde of een variabele die meegegeven wordt aan een methode of constructor en waarmee die methode of constructor iets doet.
- **DECLARATIE:** Bekendmaking van een variabele met het typeaanduiding aan de compiler zodat deze geheugen kan reserveren voor deze variabele.
- **VARIABELE:** een variabele is een plaats in het interne geheugen van een bepaalde grootte, waar een waarde kan worden opgeslagen
- **LOGISCHE OPERATOR:** Met een logische operator kun je de inhoud van twee variabelen of waarden vergelijken.
- **Globale variabele:** Een variabele die buiten de methodes gedeclareerd wordt en daardoor in elke methode aangeroepen kan worden. Volgens afspraak zetten we globale variabelen meteen onder de klassedefinitie.
- **Lokale variabele:** Een variabele die gedeclareerd wordt in een methode of constructor. Volgens afspraak zetten we de declaratie van een lokale variabele onder de definitie van een methode of constructor.
- **INSTANTIE- OF KLASSENVARIABELE:** Een instantie- of klassenvariabele is hetzelfde als een globale variabele.
- **COMMENTAAR:** commentaar dient als uitleg van de broncode. Commentaar wordt door de compiler overgeslagen.
- **BOOLEAN:** type variabele waarin twee waarden kunnen worden opgeslagen: **true** of **false**.

3. De wereld van Garp

In dit hoofdstuk gaan we de wereld van Garp in orde maken. We zorgen dat als het scenario wordt gestart, Garp en Gnomus automatisch op de wereld geplaatst worden. We zetten andere obstakels op de wereld. En tenslotte blijft het niet stil in de wereld van Garp: We gaan de achtergrondmuziek maken. Intussen leren we ook meer over het systeem van Greenfoot en we leren gebruik te maken van een tellende lus.

3.1 Garp en Gnomus samen in de wereld

De constructor van de wereld wordt aangeroepen zodra het scenario geopend wordt. Van deze eigenschap gaan we gebruik maken om vooraf objecten op de wereld te plaatsen. We gaan eerst Garp in het midden van de wereld plaatsen. Om een object toe te voegen aan de wereld gebruiken we de methode `addObject()`; van de superklasse **World**. Het codeblok van de constructor van `GarpsWorld` bestaat uit een regel:

```
public GarpsWorld()
{
    // Create a new world with 700x500 cells with a cell size of 1x1 pixels.
    super(700, 500, 1);
}
```

Op p. 10 hebben we de methode `super()` al besproken. Boven de aanroep van `super` mag alleen commentaar staan verder niets, dus ook geen declaratie. We gaan een nieuwe methode maken die we `populateTheWorld()` noemen, waarin we de objecten op de wereld gaan plaatsen. De aanroep plaatsen we onder `super()`:

```
public GarpsWorld()
{
    // Create a new world with 700x500 cells with a cell size of 1x1 pixels.
    super(700, 500, 1);
    populateTheWorld();
}
```

Vervolgens maken we een nieuwe methode met dezelfde naam en daarin kunnen we Garp met `addObject()` op de wereld plaatsen:

```
protected void populateTheWorld()
{
    addObject(new Garp(), 350, 250);
}
```

Opdracht 31:

Pas de code aan en zie dat Garp midden in de wereld staat. Druk op een van de pijltjestoetsen en er gebeurt niets. Klik nu eerst op ► **Run** en druk daarna op de pijltjestoetsen en Garp gaat daarheen waar jij wilt.

De methode `addObject()` verwacht drie parameters. De eerste parameter is een object. Zoals je weet, maken we met het sleutelwoord `new` een nieuw object, in dit geval van de klasse **Garp**. Daarna volgen twee parameters van het type `int`. `int` (afkorting van integer) is een geheel getal, dus een getal zonder cijfers achter de komma. Deze twee getallen bepalen de plek (de `x` en de `y`) waar Garp in de wereld geplaatst wordt.

Het is nu heel makkelijk Gnomus in de wereld te zetten. Dat doen we op dezelfde manier, alleen de vraag is op welke plek. Dat laten we aan het toeval over, net als het pad dat Gnomus volgt. We voegen de volgende opdracht toe aan de methode `populateTheWorld()`:

```
protected void populateTheWorld()
{
    addObject(new Garp(), 350, 250);
    addObject(new Gnomus(), Greenfoot.getRandomNumber(700),
        Greenfoot.getRandomNumber(500));
}
```

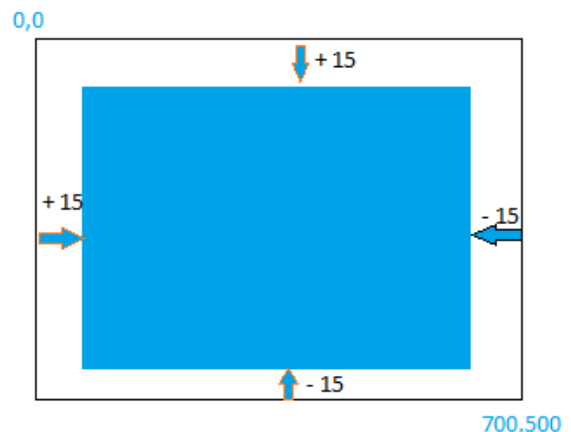
De getallen die als parameter aan `Greenfoot.getRandomNumber()` worden meegegeven, zijn de breedte en de hoogte van de wereld.

Opdracht 32:



Pas de code aan. Garp staat nog steeds in het midden van de wereld. Druk nu een paar maal achter elkaar op de knop  **Reset** en zie dat Gnomus steeds op een andere plek staat.

Als we op de knop **Run** klikken en Gnomus staat aan de rand van de wereld, dan gaat het niet goed met Gnomus. Hij draait dan rond als een gek. We moeten hem dus weghouden van de rand van de wereld. Nu wordt Gnomus geplaatst vanaf 0 (de boven- en de linkerkant) tot 700 of 500. Dus de parameters moeten verlaagd worden zodat Gnomus niet aan de rechterkant of aan de onderkant van de wereld start en dan moet er bijvoorbeeld 15 bij opgeteld worden zodat Gnomus niet aan de linker- of de bovenkant van de wereld begint. Dat doen we als volgt:

```
protected void populateTheWorld()
{
    addObject(new Garp(), 350, 250);
    addObject(new Gnomus(), Greenfoot.getRandomNumber(670) + 15,
        Greenfoot.getRandomNumber(470) + 15);
}
```



Opdracht 33:

Pas de code aan. Druk nu een paar maal achter elkaar op de knoppen  **Reset** en  **Run**.
Gnomus start steeds op een andere plek maar gaat altijd naar rechts.

Om ervoor te zorgen dat Gnomus een willekeurige richting kiest, hebben we de constructor in de klasse **Gnomus** nodig, waarin we Gnomus een willekeurige draai laten maken, zodat onzeker wordt welke kant Gnomus opgaat. Eigenlijk hebben we die code al in de methode *act()* staan. Die code halen we uit de methode *act()* en plaatsen we in een andere nieuwe methode met de naam *setRotation()*:

```
protected void setRotation()
{
    if(Greenfoot.getRandomNumber(100) < 50) {
        setRotation(getRotation() + Greenfoot.getRandomNumber(180));
    }
    else {
        setRotation(getRotation() - Greenfoot.getRandomNumber(180));
    }
}
```

In de methode *act()* roepen we *setRotation()* aan op de plek waar we de code hebben weggehaald, namelijk uit het codeblok van van if-statement:

```
if(atWorldEdge()) {
    move(-5);
    setRotation();
}
```

Opdracht 34:

Doe opdracht 33 nog een keer maar dan om te testen of je de code goed hebt aangepast.

Er is dus nog geen verandering in het gedrag van Gnomus. Nu kunnen we de constructor aanmaken en de code op dezelfde manier aanroepen als in de methode *act()*:

```
public Gnomus()
{
    setRotation();
}
```

Het voordeel van deze manier van werken, code die je meermalen nodig hebt in een aparte methode zetten, is dat je broncode korter is en dat je de code op een plek en niet op meerdere plekken hoeft te veranderen bij wijzigingen. Nog een voordeel is, dat de methode een naam krijgt, waarmee je kunt aanduiden wat de code doet.

Opdracht 35:

Pas je code aan en test of Gnomus nu wel steeds een andere richting kiest

3.2 De wereld verrijken

Er zijn nog drie andere spelelementen die we aan de wereld gaan toevoegen: de diamanten, de obstakels en de hinderlagen in de vorm van bommen. En net als bij Gnomus willen we ze op willekeurige plekken in de wereld zetten. Eerst gaan we 10 diamanten op de wereld plaatsen. We hebben de klasse **Diamond** nodig.

**Opdracht 36:**

Maak een klasse **Diamond** aan en kies een afbeelding bij deze klasse.

Nu kunnen we de diamanten met de volgende opdracht aan **GarpsWorld** toevoegen:

```
addObject(new Diamond(), Greenfoot.getRandomNumber(700), Greenfoot.getRandomNumber(500));
```

Maar om tien diamanten te plaatsen moeten we tien maal deze opdracht in **GarpsWorld** zetten. Stel je voor dat het gaat om 1000 diamanten. Dan betekent dat 1000 opdrachten. Daar bestaat gelukkig een andere oplossing voor.

De computer is heel goed in steeds weer dezelfde opdracht uit te voeren. Dat gaan we doen door een lus. Omdat we al weten hoeveel diamanten er in de wereld komen, kunnen we een tellende lus maken. Allereerst moeten we bovenaan in de methode een teller declareren:

```
int teller;
```

Vervolgens kunnen we de lus maken:

```
for(teller = 0; teller < 10; teller++) {
    addObject(new Diamond(), Greenfoot.getRandomNumber(700),
        Greenfoot.getRandomNumber(500));
}
```

For geeft aan dat het om een tellende lus gaat. De teller begint bij 0 en telt door tot 10. Telkens als de lus doorlopen is, wordt er 1 bij teller opgeteld. Dus de opdracht in het codeblok wordt uitgevoerd als de teller 0, 1, 2 . . . en 9 is. Dat is tien keer. Als de waarde in teller 10 is, gaat het programma verder met de eerst volgende opdracht na de lus.

**Opdracht 37:**

Maak de klasse **Rock** aan en kies een afbeelding bij deze klasse. Plaats vervolgens 6 rotsen in de wereld.

Opdracht 38:

Maak de klasse **Bomb** aan en kies een afbeelding bij deze klasse. Plaats vervolgens 4 bommen in de wereld.

Als je alles goed hebt gedaan, zijn er in de wereld van Garp 4 bommen, 6 rotsen en 10 diamanten bij gekomen en ziet de wereld er ongeveer als volgt uit:



3.3 De stilte voorbij

Tot nu toe is het erg stil in de wereld van Garp. Die stilte gaan we nu doorbreken. Hier houden we ons bezig met de achtergrondmuziek. Speciale geluidseffecten komen later aan de orde. Om achtergrondmuziek te laten horen wordt eerst de muziek van schijf in het intern geheugen gelezen en vervolgens afgespeeld. Als de muziek afgelopen is en het spel duurt nog voort, dan wordt de muziek opnieuw vanaf het begin afgespeeld.

Om te beginnen hebben we een klassenvariabele nodig om de muziek in te bewaren voor later gebruik. Deze variabele declareren we boven in de klasse **GrapsWorld**:

```
private GreenfootSound sound;
```

GreenfootSound is een klasse van Greenfoot zelf. Met deze klasse kunnen we op eenvoudige wijze de achtergrondmuziek laten horen. *sound* is de naam van de variabele waarin we de muziek bewaren. Vervolgens maken we van deze variabele een object en we lezen meteen de muziek van schijf:

```
sound = new GreenfootSound("Zelda.mp3");
```


Zorg er wel voor, dat het bestand in de directory sounds van je scenario staat. Anders krijg je een foutmelding. En Java is hoofdlettergevoelig, dus let daar ook op. Dan kunnen we de muziek starten. Dat doen we met:

```
sound.playLoop();
```

De punt tussen *sound* en *playLoop()* betekent dat de methode *playLoop()* behoort tot het object *sound*. Dit wordt de puntnotatie genoemd. Een punt gebruik je als je een methode uit een andere klasse of object aanroept.

De methode *playLoop()* van de klasse **GreenfootSound** zorgt ervoor dat de muziek opnieuw gestart wordt als de muziek afgelopen is. We hebben dus verder geen omkijken naar de achtergrondmuziek tijdens het spel.

De twee laatste regels voegen we toe aan de methode *populateTheWorld()*.

Opdracht 38:

Maak de klasse **Bomb** aan en kies een afbeelding bij deze klasse. Plaats vervolgens 4 bommen in de wereld.

De muziek begint meteen te spelen en wacht niet totdat het spel begint, met andere woorden de muziek wacht niet totdat je op de knop ► **Run** hebt geklikt. De klasse **World** van **Greenfoot** kent twee methodes die we kunnen gebruiken. De eerste methode is *started()* die wordt aangeroepen als er op ► **Run** wordt geklikt. De tweede is de methode *stopped()* die aangeroepen wordt als er op de knop ↺ **Reset** wordt geklikt. Daarvan gaan we gebruik maken om de muziek te starten en te stoppen. De opdracht *sound.playLoop();* verplaatsen we naar de methode *started()*:

```
public void started()
{
    sound.playLoop();
}
```

Deze methode is public omdat hij van buiten de klasse aangeroepen wordt door het systeem van **Greenfoot**. Nu begint de muziek te spelen als we op de knop ► **Run** klikken.

Om de muziek te laten stoppen, gebruiken we de methode *stopped()*:

Opdracht 40:

Pas de code aan. Hoor wat er gebeurt als je op de knop ► **Run** klikt. Klik nu ook op || **Pause** en je hoort dat de muziek doorgaat. Klik op de knop ↺ **Reset** en de muziek stopt wel.

Om de muziek te laten stoppen als we op de knop || **Pause** drukken, gebruiken we de methode *stopped()* met daarin de opdracht om de muziek te stoppen:

```
public void stopped()
```

```
{
    sound.stop();
}
```

Opdracht 41:

Pas de code aan en hoor wat er gebeurt als je op de knoppen ► **Run**, || **Pause** en ↺ **Reset** drukt.

3.4 Erop of eronder

Als je het spel nu draait dan beweegt Gnomus zich onder de rotsen, diamanten en bommen door. Hetzelfde doet Garp als je hem laat bewegen. Dat heeft te maken met de volgorde waarin Greenfoot tekent. Die tekenvolgorde kunnen we wijzigen met de methode `setPaintOrder()`. Die methode gebruiken we nadat alle spelelementen aan de wereld zijn toegevoegd. We voegen deze opdracht toe aan de methode `populateTheWorld()` onder de opdrachten met `addObject()`:

```
setPaintOrder(Garp.class, Gnomus.class, Diamond.class, Bomb.class, Rock.class);
```

De laatste klasse (de rotsen) worden het eerst getekend in de wereld. Garp wordt als laatste getekend en is dus altijd zichtbaar.

Opdracht 42:

Pas de code aan en kijk wat er gebeurt als Garp en Gnomus bewegen.

VAKTAAL

- **INTEGER:** Geheel getal, dus een getal zonder cijfers achter de komma. In Java wordt een integer met `int` aangeduid bij een declaratie. Met een `int` kun je rekenen.
- **LUS:** een lus is een opdracht waardoor een codeblok die bij de lus hoort, meerdere malen wordt uitgevoerd.
- **TELLENDE LUS:** een lus waarvan het aantal maal dat het codeblok van de lus wordt uitgevoerd, van tevoren vast staat.
- **PUNTNOTATIE:** de punt wordt gebruikt om aan te duiden dat een methode of een variabele bij een object of klasse hoort: `klasse.variabelenaam` of `object.methodenaam`

4. Leven en niet laten leven

Garp en Gnomus leven nog langs elkaar heen. Ze gaan nog over de rotsen en Garp verzamelt nog steeds geen diamanten. We gaan nu eerst Garp diamanten laten verzamelen en daarna leren we Gnomus hoe hij Garp moet vermoorden. En verder zullen ze niet over de rotsen heen kunnen, maar moeten zij er omheen.

4.1 Diamanten verzamelen

Garp moet zo veel mogelijk diamanten verzamelen, voordat hij door Gnomus om zeep wordt geholpen. Als Garp alle diamanten heeft verzameld, heeft de speler gewonnen. Dat verzamelen doet Garp door op een diamant te komen. Garp moet dus eerst weten of hij een diamant tegen komt. Daarvoor gebruiken we de methode *getOneObjectAtOffset()* van de klasse **Actor**. Deze methode gebruikt drie parameters. De eerste twee zijn de relatieve afstand tot Garp. De laatste parameter is de klasse waar het om gaat, in dit geval de diamanten. Als Garp een diamant 'ziet', dan wordt dat teruggegeven en anders geeft deze methode null terug. null in computertaal betekent, dat iets niet bestaat. Eerst declareren we een variabele van de klasse die we willen zien en dan kijken we of we er een tegenkomen:

```
Actor diamond;
diamond = getOneObjectAtOffset(0, 0, Diamond.class);
```

Deze regels zetten we in de methode *act()* van de klasse **Garp** onder de code die er al staat. Dan wordt bij elke aanroep gecontroleerd of Garp een diamant te pakken heeft. Vervolgens moet er iets gebeuren als er een diamand is. Eerst controleren we of Garp een diamant ziet, zo ja, dan wordt de diamant van de wereld verwijderd.

```
if(diamond != null) {
    world = getWorld();
    world.removeObject(diamond);
}
//Als Garp een diamant ziet
//In welke wereld leeft Grap
//Verwijder de diamant van de wereld
```

Eerst wordt gecontroleerd of de afbeelding van Garp een afbeelding van een diamant raakt. Als dat zo is, dan wordt het object wereld opgevraagd met *getWorld()*, een methode van de klasse **Actor**. Daarna wordt de diamant van de wereld verwijderd met de methode *removeObject()* van het object World. Wel moet er nog de variabele World gedeclareerd worden, voordat de code uitgevoerd kan worden:

```
World World;
```

4.2 Orde op zaken stellen

In de methode *act()* van de klasse **Garp** staan nu opdrachten die te maken hebben met de besturing van Garp door de speler en opdrachten die te maken hebben met het verzamelen van diamanten. Dat maakt de code onleesbaar en die opdrachten moeten we van elkaar scheiden in methodes. We gaan dus de code aanpassen zonder de functionaliteit te veranderen. De functionaliteit is het gedrag van in dit geval Garp. Zijn gedrag verandert niet. Een wijziging in de code zonder wijziging van functionaliteit heet refactoring.

We maken twee methodes: *movingAndTurning()* en *collectingDiamonds()*. De namen van de twee methodes geven de functionaliteit van de opdrachten in het codeblok. De eerste methode zorgt voor de besturing van Garp door de speler en de tweede verzamelt de diamanten.

We kunnen met copy / paste het codeblok met opdrachten vullen. De twee methodes zien er als volgt uit:

```
protected void collectingDiamonds()
{
    World world;
    Actor diamond;

    diamond = getObjectAtOffset(0, 0, Diamond.class);
    if(diamond != null) {
        world = getWorld();
        world.removeObject(diamond);
    }
}
```

```
protected void movingAndTurning()
{
    if(Greenfoot.isKeyDown("left")) {
        if(getImage() == imageRight) {
            setImage(imageLeft);
        }
        setRotation(0);
        move(-5);
    }
    if(Greenfoot.isKeyDown("right")) {
        if(getImage() == imageLeft) {
            setImage(imageRight);
        }
        setRotation(0);
        move(5);
    }
    if(Greenfoot.isKeyDown("up")) {
        if(getImage() == imageLeft) {
            setImage(imageRight);
        }
        setRotation(-90);
        move(5);
    }
    if(Greenfoot.isKeyDown("down")) {
        if(getImage() == imageLeft) {
```

```

        setImage(imageRight);
    }
    setRotation(90);
    move(5);
}
}

```

Nu kunnen we de code uit de methode *act()* verwijderen en daarvoor in de plaats zetten we de volgende twee regels:

```

    movingAndTurning();
    collectingDiamonds();

```

Opdracht 43:

Pas de code aan en zorg ervoor dat alles werkt als voor de wijziging.

4.3 Als een rots in de branding

Tot nu toe kan Garp gewoon over de rotsen heen lopen. Nu gaan we ervoor zorgen dat Garp tegen een rots aanloopt. Net zoals bij de diamanten moet Garp eerst weten of hij tegen een rots aanloopt:

```

Protected boolean foundRock() {
    Actor rock;

    rock = getObjectAtOffset(0, 0, Rock.class);
    if(rock != null) {
        return true;
    }
    Return false;
}

```

De methode *foundRock()* controleert alleen maar of Garp tegen een rots is aangelopen. De methode geeft *true* terug als Garp tegen een rots aanloopt en geeft *false* terug als Garp niet tegen een rots aanloopt. Nu kunnen we in de methode *movingAndTurning()* bepalen wat er moet gebeuren als Garp tegen een rots aanloopt:

```

if(foundRock()) {
    move(-5);
}

```

Als Garp tegen een rots aanloopt, dan doet Garp een stap terug, zodat hij verder kan. Deze code komt te staan onder elke *move()* opdracht. Pas wel de waarde van de parameter van *move()* aan, zodat Garp een stap terug doet, en de speler Garp kan aansturen met de pijltjestoetsen.

Opdracht 44:

Pas de code aan en zorg ervoor dat alles werkt als voor de wijziging.

4.3 De ontploffing in beeld

Als Garp tegen een bom aanloopt, ontploft deze. Daarvoor verschijnt er een nieuw object ten tonele: de ontploffing. Als de ontploffing plaats heeft gevonden, is het spel geëindigd. Bij een explosie zijn twee dingen van belang: beeld en geluid.

We maken een klasse **Explosion** waarin die twee elementen beeld en geluid gecombineerd worden. Er wordt pas een object van die klasse gemaakt als de ontploffing moet plaatsvinden, dus niet zoals bij de andere objecten aan het begin van het spel, maar pas als het object nodig is.

Om de explosie zichtbaar te maken maken we gebruik van acht afbeeldingen. Die acht afbeeldingen maken we van de afbeelding zoals die hiernaast staat. We laten eerst de acht afbeeldingen van klein naar groot zien en daarna van groot naar klein. Om dat te kunnen doen maken we een array waarin acht afbeeldingen passen. Een array is een bepaald aantal variabelen van dezelfde type. In dit geval maken we een array van acht variabelen die allemaal van het type *GreenfootImage* zijn. Dat doen we met de code:

```
images = new GreenfootImage[8];
```

Er worden acht afbeeldingen gemaakt die allemaal dezelfde naam hebben: *images*. Iedere afbeelding afzonderlijk krijgt een nummer (van 0 tot en met 7), een index in vaktermen, waaraan de verschillende afbeeldingen herkend worden. Dus *images[0]* is de eerste afbeelding en *images[7]* is de laatste afbeelding.

Voordat deze array van afbeeldingen gebruikt kan worden, moet hij eerst gedeclareerd worden:

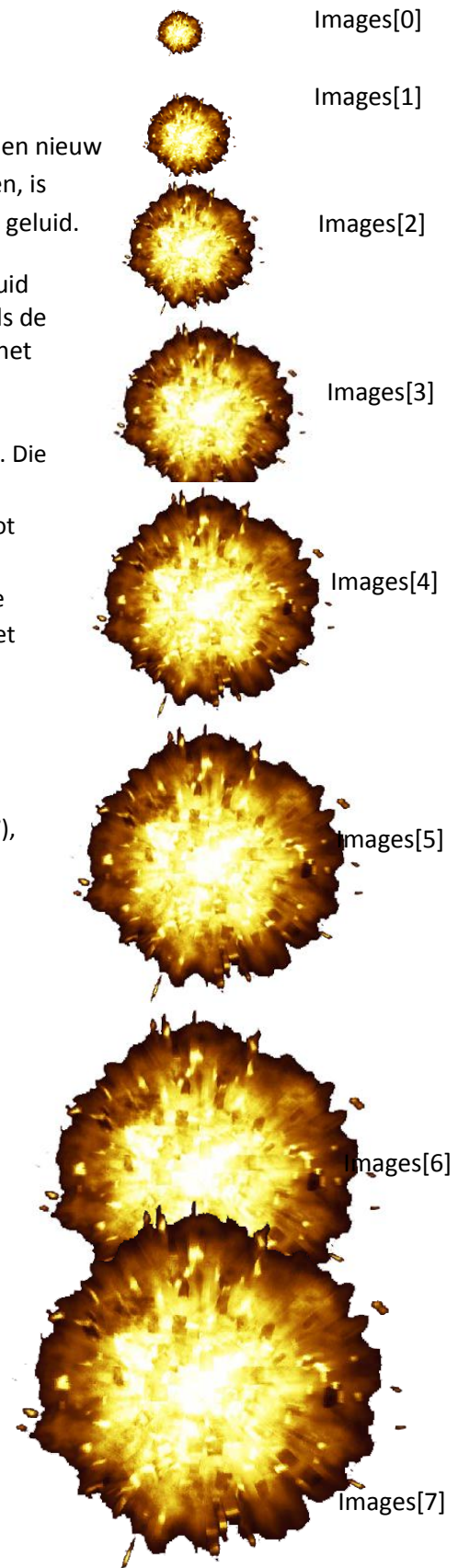
```
private GreenfootImage[] images;
```

De basis is de oorspronkelijke afbeelding. Dat is de laatste die is afgebeeld. Deze gaan we schalen, met andere woorden we geven deze afbeeldingen acht verschillende groottes.

We noemen de afbeelding die op schijf staat: *baseImage*:

```
baseImage = new GreenfootImage("explosion.png");
baseSize = baseImage.getWidth();
```

En we vragen van deze afbeelding de grootte op die in *baseSize* komt te staan. Dan berekenen we het verschil in grootte van de acht afbeeldingen:



```
verschil = baseSize / 8;
size = 0;
```

en door de laatste regel stellen we de beginwaarde op 0. Nu kunnen we een voor een de acht afbeeldingen gaan maken uiteraard in een tellende lus (zie pagina 28):

```
for(teller = 0; teller < 8; teller++) {
    size += verschil;
    images[teller] = new GreenfootImage(baseImage);
    images[teller].scale(size, size);
}
```

In de eerste regel tellen we het verschil op bij de grootte, die de eerste keer 0 is. Daarna kennen we de basisafbeelding toe aan de afbeelding in de array en in de derde regel schalen we de afbeelding (de grootte van de afbeelding wordt gewijzigd). De volgorde is van klein naar groot, dus de eerste afbeelding (*images[0]*) is de kleinste afbeelding en de laatste afbeelding (*images[7]*) is de grootste afbeelding. Dat komt doordat steeds opnieuw het verschil bij de variabele *size* wordt opgeteld. Dat komt doordat steeds opnieuw het verschil bij de variabele *size* wordt opgeteld. De gehele constructor ziet er nu als volgt uit:

```
public Explosion(){
    int size = 0, baseSize, verschil, teller;

    baseImage = new GreenfootImage("Explosion.png");
    baseSize = baseImage.getWidth();
    images = new GreenfootImage[8];
    verschil = baseSize / 8;
    increment = 1;
    size = 0;
    for(teller = 0; teller < 8; teller++){
        size += verschil;
        images[teller] = new GreenfootImage(baseImage);
        images[teller].scale(size, size);
    }
}
```

Nu moeten we ervoor zorgen dat eerst de afbeeldingen van klein naar groot worden getoond en daarna in omgekeerde volgorde: van groot naar klein. Daarvoor gebruiken we de methode *act* van de klasse **Explosion**. We moeten dus de eerste keer dat de methode *act()* door Greenfoot wordt aangeroepen, de kleinste afbeelding (*images[0]*), daarna de wat grotere (*images[1]*) tot en met de grootste afbeelding *images[7]* laten zien. Dat doen we door de variabele *action* van 0 tot en met 7 te laten tellen, dus steeds als de methode *act()* wordt aangeroepen met 1 te verhogen.

```
setImage(images[action]);
action += increment;
```

Met *setImage(images[action])* zetten we de afbeelding van de klasse. Dat is dus de eerste keer *images[0]*, de kleinste afbeelding. Daarna verhogen we *action* met de waarde in *increment*. Aan de variabele

increment hebben we in de constructor de waarde 1 toegekend, dus de waarde in *action* wordt steeds met 1 verhoogd. Als de methode *act()* van **Explosion** opnieuw wordt aangeroepen, wordt de volgende afbeelding aan het object van de klasse **Explosion** geassocieerd en getoond.

Als de waarde in *action* 8 is, komt *action* buiten de grenzen van de array. Dus moeten we controleren of de waarde in *action* 8 is en zo ja dan zorgen we ervoor dat de waarde in *action* met 1 verlaagd wordt:

```
if(action > 7) {
    increment = -1;
    action += increment;
}
```

Als de waarde in *action* groter is dan 7, dan wordt de waarde van -1 aan *increment* toegekend en vervolgens wordt dus de waarde van *action* met -1 verhoogd, dus met 1 verlaagd. Nu wordt dus eerst de grootste afbeelding (*images[7]*) getoond en daarna steeds een kleinere. Als de waarde in *action* kleiner dan 0 wordt, dan komt *action* weer buiten de grenzen van de array. Daarop moet ook gecontroleerd worden:

```
if(action < 0) {
    getWorld().removeObject(this);
    Greenfoot.stop();
}
```

Als de waarde kleiner in *action* kleiner dan 0 is, dan is de ontploffing voorbij en moet deze van de wereld worden verwijderd. De Garp is dood door de ontploffing, dus moet het spel beëindigd worden. Dat gebeurt door de opdracht *Greenfoot.stop()*;

4.3 De ontploffing met geluid

De ontploffing vindt tot nu toe in stilte plaats. Een ontploffing zonder geluid is geen ontploffing. Dus we gaan er geluid aan toevoegen. Dat doen we door de regel *Greenfoot.playSound("Explosion.wav");* aan de methode *act()* toe te voegen. Maar de methode *act()* wordt verschillende malen door *Greenfoot* aangeroepen en een ontploffing hoor je maar een keer. Dus we moeten ervoor zorgen dat het geluid maar een keer afgespeeld kan worden. Dat doen we door de boolean *geluid*. Deze declareren we boven in de klasse:

```
private boolean geluid;
```

In een boolean kunnen twee waarden staan: true of false, meer niet. De eerste keer moet het geluid afgespeeld worden, dus de eerste waarde die aan variabele *geluid* toegekend wordt, is true. Dat doen we in de constructor:

```
geluid = true;
```

Als het geluid eenmaal is afgespeeld, wordt de waarde false aan *geluid* toegekend. Nu kunnen we de volgende regels aan de methode *act()* toevoegen:

```
if(geluid) {
    Greenfoot.playSound("Explosion.wav");
}
```



```

    geluid = false;
}

```

Nu wordt alleen de eerste keer dat de methode `act()` aangeroepen wordt, het geluid afgespeeld, daarna niet meer. De methode `act()` ziet er nu als volgt uit:

```

public void act() {
    setImage(images[action]);
    action += increment;

    if (geluid){
        Greenfoot.playSound("Explosion.wav");
        geluid = false;
    }

    if (action > 7){
        increment = -1;
        action += increment;
    }

    if (action < 0){
        getWorld().removeObject(this);
        Greenfoot.stop();
    }
}

```

Opdracht 45:

Schrijf de klasse `Explosion` en zorg ervoor dat deze gecompileerd wordt zonder fouten.

4.4 Garp ontploft

Als we nu het spel spelen, zal Garp vrolijk fluitend over de bommen heen lopen. Dat komt omdat we Garp moeten vertellen, wat hij moet doen, als hij een bom tegenkomt. Dat doen we in de methode met de naam `foundBomb()`. De eerste regels zijn hetzelfde als de regels van de methodes `collectingDiamonds()` en `foundRock()` die we al eerder hebben gemaakt (zie pagina 32 - 34):

```

public void foundBomb() {

    Actor bomb;

    bomb = getOneObjectAtOffset(0, 0, Bomb.class);
    if(bomb != null) {
        ...
    }
}

```

Door deze code weet Garp of hij wel of niet tegen een bom aanloopt. Als hij tegen een bom aanloopt, dan worden de opdrachten in het codeblok van het if-statement uitgevoerd (. . .). Allereerst gaan we de bom verwijderen van de wereld: `getWorld().removeObject(bomb);`

Vervolgens gaan we de ontploffing aan de wereld toevoegen: `getWorld().addObject(new Explosion(), getX(), getY());` Met de methoden `getX()` en `getY()` vragen we op waar in de wereld Garp zich bevindt. Daar moet namelijk de ontploffing plaatsvinden. Met `new Explosion()` maken we een object van de klasse **Explosion**. Met de methode `addObject()` kunnen we objecten aan de wereld toevoegen. Tot slot vragen we met `GetWorld()` op aan welke wereld de explosie wordt toegevoegd. Dat is dus de wereld waarin Garp tegen de bom aanliep.

Tot slot moet Garp van de wereld verwijderd worden. Dat doen we met de opdracht `getWorld().removeObject(this);` `this` verwijst naar het object Garp dat op dat moment op de wereld is. De methode `removeObject()` verwijdert een object van de wereld, die weer met `getWorld()` wordt opgevraagd.

Opdracht 46:

Schrijf de methode `foundBomb()` en zorg ervoor dat de klasse **Garp** zonder fouten wordt gecompileerd

Het enige wat we nu nog moeten doen om de explosie te laten plaatsvinden is een verwijzing maken in de methode `act()` van de klasse **Garp**. Dat doen we door de volgende regel aan de methode `act()` toe te voegen: `foundBomb();`

Opdracht 47:

Voeg de regel `foundBomb()` toe aan de methode `act()` van de klasse **Garp** en laat Garp tegen een bom aanlopen. Kijk wat er gebeurt.

4.5 Gnomus is aan de beurt

Gnomus gaat nog steeds over de rotsen heen. Ook hij moet van richting veranderen.

```
protected void lookForRock() {
    Actor rock;

    rock = getObjectAtOffset(0, 0, Rock.class);
    if((rock != null))
        turn(45);
}
```

De methode heet `lookForRock()` en heeft geen parameters. Dat zie je aan de haakjes waar niets tussen staat. In de eerste regel van het codeblok van de methode wordt gecontroleerd of Gnomus tegen een rots aanloopt, en zo ja, dan draait Gnomus 40 graden. Nu moeten we in de methode `act()` deze methode aanroepen. Dat doen we door de opdracht `lookForRock();` toe te voegen aan de methode `act()`.

Opdracht 48:

Pas de code van de klasse **Gnomus** aan, compileer de code en kijk wat er gebeurt als Gnomus tegen een rots aanloopt.

Gnomus gaat nog steeds over de diamanten en de bommen heen. We moeten de methode abstracter maken: De methode moet voor alle objecten in de wereld gelden. Alle objecten in de wereld zijn van het type **Actor**. Dus moeten we **Rock.class** als parameter van `getOneObjectAtOffset()` veranderen in **Actor.class**:

```
rock = getOneObjectAtOffset(0, 0, Actor.class);
```

Als we deze regel wijzigen, gaat Gnomus opzij voor alle objecten in de wereld, maar nu is de naamgeving niet logisch: De naam van de methode suggereert dat de methode alleen voor rotsen geldt en niet voor de andere objecten op de wereld. Dus moeten we deze naam veranderen in `lookForActor()`.

Hetzelfde geldt voor de declaratie: `Actor rock`; Deze wijzigen we in `Actor actor`; We moeten dan ook overall `rock` in de methode in `actor` veranderen. Tot slot moet de aanroep in de methode `act()` aangepast worden in `lookForActor()`. Als we later iets aan de code moeten veranderen, dan lezen we door de logische naamgeving meteen wat de code doet.

Opdracht 49:

Pas de code van de klasse **Gnomus** aan, compileer de code en kijk wat er gebeurt als Gnomus tegen een rots, een diamant of een bom aanloopt. Probeer met Garp eens tegen Gnomus aan te lopen en kijk wat er dan gebeurt.

4.6 Het einde van Garp

Zoals je ziet, gaat Gnomus nu opzij voor Garp. Dat is natuurlijk niet de bedoeling. Als Gnomus Garp ontmoet, betekent dat het einde van Garp. Dus we moeten een methode maken waarin Gnomus weet, dat hij Garp ontmoet en Garp van de wereld wordt verwijderd. De methode ziet er niet veel anders uit als de methodes die we tot nu toe geschreven hebben:

```
protected void lookForGarp() {
    Actor garp;

    garp = getOneObjectAtOffset(0, 0, Garp.class);
    if(garp != null) {
        Greenfoot.playSound("scream.mp3");
        getWorld().removeObject(garp);
        Greenfoot.stop();
    }
}
```

Uiteraard moet deze methode opgeroepen worden in de methode *act()*.

Als garp wordt gevonden, wordt er eerst een schreeuw van wanhoop ten gehore gebracht, daarna wordt Garp van de wereld verwijderd en wordt het spel gestopt.

Opdracht 50:

Voeg de methode *lookForGarp()* toe aan de code van de klasse **Gnomus** en zorg voor een aanroep in de methode *act()*. Probeer met Garp eens tegen Gnomus aan te lopen en kijk wat er dan gebeurt.

VAKTAAL

- **NULL:** de waarde null betekent dat iets niet bestaat.
- **REFACTORING:** de code wordt gewijzigd zonder dat de functionaliteit van de code verandert. Dit gebeurt om de code leesbaarder en logischer te maken.
- **ARRAY:** een bepaald aantal variabelen van hetzelfde type die onder dezelfde naam benaderbaar zijn. Aan de hand van een index is een variabele benaderbaar.
- **INDEX:** een nummer waarmee een variabele in een array benaderbaar is.
- **SCHALEN:** de grootte van een afbeelding wijzigen.
- **THIS:** sleutelwoord in Java, waarmee naar het object verwezen wordt waarin het woord wordt gebruikt.
- **LOGISCHE NAAM:** de naam van een variabele of methode vertelt iets over de inhoud van de variabele of vertelt wat een methode doet.

5. De buit verdelen

Tot slot moet de score nog worden bijgehouden tijdens het spel en moet aan het eind van het spel een venster verschijnen met daarin de aanduiding of de speler gewonnen of verloren heeft. Verder wordt de score zichtbaar gemaakt en de duur van het spel. De score is het aantal diamanten dat Garp heeft verzameld. Om het aantal diamanten bij te houden dat Garp heeft verzameld, maken we een nieuwe klasse aan: **Counter**. Deze klasse bewaart ook de begintijd van het spel. Deze tijd hebben we aan het einde van het spel nodig.

5.1 De diamanten tellen

De klasse **Counter** houdt tijdens het spel het aantal diamanten dat door Garp is verzameld. Dat aantal maken we zichtbaar links onder in het scherm. Om dit te bereiken maken we een nieuwe klasse **Counter**. Let op: Deze klasse heeft geen afbeelding!



Er zijn twee regels nodig voor het importeren van de bibliotheken:

```
import greenfoot.*; // (World, Actor, GreenfootImage, Greenfoot and MouseInfo)
import java.awt.*;
```

Deze klasse is een uitbreiding van de klasse **Score** (zie bijlage 2). Daarmee wordt deze klasse niet alleen een uitbreiding van de klasse **Score** maar tegelijk een uitbreiding van de klasse **Actor** omdat **Score** een uitbreiding is van **Actor**:

```
public class Counter extends Score {
```

Omdat we de score willen bijhouden wordt de score een klasse of globale variabele die overal in de klasse bereikbaar is. De scope van deze variabele is dus de gehele klasse:

```
Private int score;
```

Vervolgens maken we de constructor van deze klasse en zetten de variabele score op 0. Garp heeft tot nu toe nog geen diamanten verzameld want het spel is nog niet begonnen:

```
public Counter() {
    String scoreText;

    score = 0;
    scoreText = "Aantal diamanten: " + score;
```

Tevens declareren we `scoreText` als lokale variabele, zodat we dadelijk de score op de wereld kunnen zetten en kennen we een String toe aan deze variabele. Het plusteken op deze regel betekent niet dat er opgeteld wordt, maar dat de waarde die in de int score staat, geconverteerd wordt naar een string en vervolgens toegevoegd wordt aan de string ervoor.

Nu hebben we een image nodig waarop we de tekst kunnen plaatsen en die we vervolgens op de wereld kunnen zetten. We declareren een `GreenfootImage` onder de declaratie van `scoreText`:

```
GreenfootImage image;
```

En maken een image aan van willekeurige grootte:

```
image = new GreenfootImage(10, 100);
```

Nu moeten we ervoor zorgen dat de tekst "Aantal diamanten: 00" op de afbeelding past. "00" is de plek waar later een of twee cijfers komen te staan. Zo houden we er nu al rekening mee dat er meer dan negen diamanten door Garp kunnen worden verzameld:

```
scoreText = "Aantal diamanten: 00";
```

Om de grootte van de afbeelding te bepalen hebben we de hoogte en de breedte in pixels nodig van de string. Om die hoogte en breedte te kunnen op te slaan gebruiken we de klasse **Dimension** van Java. Daarvoor moeten we die klasse eerst importeren. Dat doen we boven in de code:

```
import java.awt.*;
```

Deze klasse van Java bevat twee variabelen die we rechtstreeks kunnen benaderen: *width* en *height*. In deze klasse krijgen we de hoogte en de breedte van de string terug. Daarom schrijven we eerst een declaratie: `Dimension dim`; die we bij de andere declaraties zetten. Dan maken we gebruik van de methode `getTextDimensions` van de klasse **Score** die de hoogte en de breedte uitrekent:

```
dim = getTextDimensions(image, scoreText);
```

```
image.scale(dim.width, dim.height);
```

In het object `dim` staan de hoogte en de breedte en in de tweede regel passen we de hoogte en de breedte van de afbeelding aan aan de hoogte en breedte van de string. We schalen de afbeelding. Een `GreenfootImage` is doorzichtig, dus daar hoeven we niets aan te doen. We associëren de afbeelding met het object:

```
setImage(image);
```

Tot slot zetten we de tekst op de afbeelding. De letters maken we zwart en vervolgens tekenen we de tekst op de afbeelding:

```
image.setColor(new Color(0, 0, 0));
```

```
image.drawString(scoreText, 0, dim.height);
```

Door `new Color(0, 0, 0)` als parameter mee te geven aan `setColor()` wordt verteld dat met de kleur zwart getekend wordt (Zie bijlage 1). Vervolgens wordt met de methode `drawString()` de tekst op de afbeelding getekend. Let daarbij op dat het uitgangspunt de linker benedenhoek is, vandaar `dim.height`.

Opdracht 51:

Schrijf de code van de klasse **Counter** met alleen de constructor. Compileer de code en verbeter eventuele fouten.

Nu moet de score nog op de wereld zichtbaar gemaakt worden. Dat doen we in de klasse **GarpsWorld** in de methode *populateTheWorld()*. Dat is de methode waarin we andere objecten of spelelementen ook op de wereld hebben gezet. In de methode declareren we eerst drie lokale variabelen:

```
int regel, kolom;
Counter counter;
```

De eerste twee variabelen hebben we nodig om de counter linksonder op de wereld te plaatsen, in de laatste variabele wordt het object van de klasse **Counter** opgeslagen:

```
counter = new Counter();
regel = getHeight() - counter.getImage().getHeight() / 2 - 1;
kolom = counter.getImage().getWidth() / 2 + 1;
```

De laatste twee regels rekenen uit op waar de counter op de wereld geplaatst wordt. Eerst wordt de hoogte uitgerekend door met *getHeight()* de hoogte van de wereld op te vragen. Daarna wordt de hoogte van de afbeelding van de counter opgevraagd: *counter.getImage().getHeight()*. Omdat Greenfoot met de helft van de afbeelding rekent (zie p.18), delen we de hoogte van de afbeelding door twee. We trekken er 1 pixel vanaf (De afbeelding komt dus een pixel hoger te staan) zodat de tekst van de counter vrij komt van de rand aan de onderkant van de wereld. Omdat de counter aan de linkerkant staat en dus in de eerste kolom (dus kolom 0), hoeven we alleen de breedte van de afbeelding van de counter op te vragen met *getWidth()* en deze door twee te delen. Om de tekst niet tegen de linkerkant van de wereld te laten komen, tellen we er 1 bij op. Nu kunnen we met de methode *addObject* de afbeelding van de counter op de wereld plaatsen:

```
addObject(new Counter(), kolom, regel);
```

Tot slot zorgen we ervoor dat de counter altijd zichtbaar blijft (zie p. 31):

```
setPaintOrder(Counter.class, Garp.class, Gnomus.class, Diamond.class, Bomb.class, Rock.class);
```

Opdracht 52:

Vul de code in de methode *populateTheWorld()* van de klasse **GarpsWorld** aan en compileer de code en laat Garp een paar diamanten vangen.

De counter telt nog niet, zoals je ziet. Om dat voor elkaar te krijgen maken we de methode *addScore()* in de klasse **Counter**, waarin de score met een wordt opgehoogd en de afbeelding van nieuwe informatie wordt voorzien. Dat laatste doen we in een andere methode die we *updateImage()* noemen.

```
public void addScore() {
    score++;
    updateImage();
}
```

In de eerste regel wordt de score met een verhoogd. Daarna wordt de methode *updateImage()* aangeroepen. Deze ziet er als volgt uit:

```
protected void updateImage() {
    String scoreText;
    Dimension dim;
    GreenfootImage image;

    image = getImage();
    image.clear();
    scoreText = "Aantal diamanten: " + score;
    dim = getTextDimensions(image, scoreText);
    image.setColor(new Color(0, 0, 0));
    image.drawString(scoreText, 0, dim.height);
}
```

De declaraties zijn dezelfde als in de constructor van deze klasse. De eerste opdracht vraagt de *image* van het object op en slaat deze op in de lokale variabele *image*. Daarna wordt de afbeelding leeg gemaakt met *image.clear()*; De volgende vier opdrachten staan ook in de constructor (zie p.43). Door die overeenkomst kunnen we de drie laatste opdrachten van de constructor vervangen door een opdracht: *updateImage()*; Dat maakt de code korter en leesbaarder.

Opdracht 53:

Schrijf de twee methodes *addScore()* en *updateImage()* en wijzig de code in de constructor van de klasse **Counter**.

Nu moeten we er nog voor zorgen dat de counter telt als Garp een diamant verzamelt. Daarvoor gaan we naar de methode *collectingDiamonds()* van de klasse **Garp** en we voegen drie regels aan deze methode toe in het codeblok van *if(diamond != null)*

```
lijst = world.getObjects(Counter.class);
counter = (Counter)lijst.get(0);
counter.addScore();
```

De eerste opdracht maakt een lijst van alle counters die zich op dat moment op de wereld bevinden. In die lijst staat een counter, de enige op de wereld. Daarom kunnen we in de volgende opdracht met *lijst.get(0)* de eerste en de enige counter uit de lijst halen. In de lijst staan objecten van de klasse **Object** en we moeten aan Java vertellen, dat dit een object is van de klasse **Counter**. We moeten dus casten: het object van klasse veranderen. Daarom staat er *(Counter)*. Vervolgens wordt dit object toegerekend aan de variabele *counter*. Nu kunnen we aan het object counter de opdracht geven om de score te verhogen door de methode *addScore()* aan te roepen.

Om deze code werkend te maken moeten we nog twee zaken in orde brengen. Ten eerste moeten er twee declaraties aan de methode *collectingDiamonds()* worden toegevoegd:

```
Counter counter;
List lijst;
```


Ten tweede moet er een import boven in de code worden toegevoegd. List is namelijk een klasse van Java:

```
import java.util.List;
```

Opdracht 54:

Voeg de bovenstaande codes aan de klasse Garp toe en speel het spel. Kijk of de counter linksonder nu wel telt. En wat gebeurt er als Garp alle diamanten heeft verzameld?

Als Garp de laatste diamant heeft verzameld, gaat het spel gewoon door. Dat is niet de bedoeling. Het spel moet dan stoppen. Dat is eenvoudig te coderen door nog de volgende regels toe te voegen:

```
if(counter.getScore() == 12) {
    Greenfoot.stop();
}
```

Met de opdracht `Greenfoot.stop();` eindigt het spel (zie p. 37).

Als Garp twaalf diamanten heeft verzameld, stopt het spel. In counter moet dan wel een getter gemaakt worden die de waarde in score teruggeeft:

```
public int getScore() {
    return score;
}
```

Opdracht 55:

Voeg de bovenstaande codes aan de klasse Garp en Counter toe en speel het spel. En wat gebeurt er nu als Garp alle diamanten heeft verzameld?

5.2 Het einde nadert

Op het eindscherm moeten drie zaken vermeld worden: Of de speler gewonnen of verloren heeft, hoeveel diamanten Garp heeft verzameld en hoe lang het spel heeft geduurd. We beginnen met het laatste.

Om het eindscherm zichtbaar te maken hebben we een nieuwe klasse nodig

onder de klasse **Score**. Die klasse noemen we **EndScore**. Deze klasse moet de tijd weten waarop het spel gestart is. We maken een constructor voor deze klasse en vragen daarin de tijd op:



```
public EndScore() {
    startTime = System.currentTimeMillis();
}
```

Verder is er een globale variabele nodig om de begintijd in op te slaan:

```
private long startTime;
```

Het betekent wel dat deze constructor aangeroepen moet worden als het spel start. De tijd die we opgeslagen hebben in *startTime* is van het type *long*. Dat is een integer maar met een groter bereik dan *int*. In *startTime* is de tijd opgeslagen in de vorm van milliseconden. Dat is het aantal milliseconden sinds 1 januari 1970 middernacht. Dat laten we nu zo, later gaan we daarmee rekenen.

Aan het eind van het spel moeten we wederom op dezelfde manier de tijd opvragen en daarvan de starttijd aftrekken. We houden dan de duur van het spel in milliseconden over. We maken een methode met de naam *getElapsedTime()*:

```
public String getElapsedTime() {
    long duration;

    duration = System.currentTimeMillis() - startTime;
}
```

In *duration* staat nu het aantal milliseconden dat het spel geduurd heeft.

De volgende stap is het aantal milliseconden om te rekenen naar seconden, minuten en uren. Door het aantal milliseconden door 1000 te delen verkrijgen we het aantal seconden. Om het aantal uren te berekenen delen we het aantal seconden door 3600. Dat is het aantal seconden per minuut maal het aantal minuten per uur: $60 * 60$. Vervolgens moeten we de rest weten van de deling, want dat zijn de minuten en seconden die overblijven. Om de rest van een deling te weten te komen maken we gebruik van de modulus-operator: `%`. Door de rest door 60 te delen, komen we het aantal minuten te weten en de rest van de laatste deling is het aantal seconden dat overblijft. In code ziet dat er zo uit:

```
int uren, minuten, seconden;

seconden = (int)(duration / 1000);
uren = seconden / 3600;
seconden %= 3600;
minuten = (int) (seconden / 60);
seconden %= 60;
```

Omdat de variabele *duration* van het type *long* is en *seconden* van het type *int* is, moet er gecast worden. Java gaat ervan uit dat een berekening met een *long* een *long* oplevert en van die *long* moet een *int* gemaakt worden, voordat de waarde toegekend kan worden aan *seconden*. Het wijzigen van de ene type variabele naar de andere heet casten.

De laatste stap is de duur van het spel in een format te zetten, zodat deze begrijpelijk is voor de speler: uu:mm:ss. Een formatstring ziet er als volgt uit: “%02d”. In Java staat een string altijd tussen dubbele aanhalingstekens. Het %-teken betekent dat er een formatString begint. De d betekent een decimale weergave van een getal. Het getal heeft voorloopnullen en de lengte is twee tekens. We kunnen de formatstring uitbreiden met een tekst die de betekenis van het getal weergeeft: “Duur van het spel: %02d”. Na de d eindigt de format en daarachter worden tekens weer letterlijk weergegeven. Dus we kunnen er twee keer “:%02d” aan toe voegen. Dus wordt de formatstring: “Duur van het spel: %02d:%02d:%02d”. Deze formatstring kunnen we met de methode format uit de klasse String gebruiken:

```
duur = String.format("Duur: %02d:%02d:%02d", uren, minuten, seconden);
```

De uren, minuten en seconden zijn de integers die we net uitgerekend hebben en de inhoud van die variabelen wordt geplaatst op de plek van het %-teken. Aan *duur* wordt de string toegekend die de methode String.format teruggeeft. De variabele *duur* moet wel nog gedeclareerd worden. Tot slot wordt duur door de methode teruggeven, zodat deze string bijvoorbeeld op de afbeelding van de klasse **EndScore** kan worden geplaatst.

Opdracht 56:

Schrijf de methode *getElapsedTime()* in de klasse **EndScore**. Zorg ervoor dat de klasse zonder fouten gecompileerd wordt.

We gaan nu eerst de afbeelding maken en die op de wereld zetten, zodat we kunnen zien of alles goed gaat:

```
public void setEndImage() {
    GreenfootImage image;
    Dimension dim;
    String duur;

    image = new GreenfootImage(10, 10);
    duur = getElapsedTime();
    dim = getTextDimensions(image, duur);
    image.scale(dim.width, dim.height);
    setImage(image);
    image.drawString(duur, 0, dim.height);
}
```

De methode noemen we `setEndImage()`. De code die in deze methode staat, is al eerder besproken (zie p. 45). Daarom gaan we daar niet verder op in. De volgende stap is de afbeelding zichtbaar te maken op de wereld. Dat doen we in de klasse **GarpsWorld** in de methode `stopped()`. Dat is de methode die aangeroepen wordt als de opdracht `Greenfoot.stop()` wordt uitgevoerd, dus aan het einde van het spel. In die methode zetten we twee opdrachten: De eerste is een aanroep van de methode `setEndImage()` in de klasse `EndScore` en de tweede is het toevoegen van het object `endScore` aan de wereld:

```
endScore.setEndImage();
addObject(endScore, getWidth() / 2, getHeight() / 2);
```



In de tweede opdracht wordt de breedte en de hoogte van de wereld door 2 gedeeld, waardoor het midden van de afbeelding in het midden van de wereld geplaatst wordt.

Opdracht 56:

Schrijf de methode `setEndImage()` in de klasse **EndScore**. Zorg ervoor dat de klasse zonder fouten gecompileerd wordt. Voeg daarna de twee regels aan de methode `stopped()` van de klasse **GarpsWorld** toe. Speel het spel en kijk wat er aan het einde van het spel gebeurt.

Op de afbeelding moeten nog twee zaken worden gemeld. Ten eerste of de speler gewonnen of verloren heeft en het aantal verzamelde diamanten. Beide gegevens zijn afhankelijk van het aantal verzamelde diamanten. In `GarpsWorld` in de methode `stopped()` declareren we een variabele van het type `int` met de naam `score`. Vervolgens kunnen we met `counter.getScore()` het aantal diamanten opvragen, dat Garp verzameld heeft en dat aantal toekennen aan `score`:

```
score = counter.getScore();
```

Nu moeten we alleen de score nog doorgeven aan de klasse `EndScore`. Dat kunnen we eenvoudig doen door de score als parameter mee te geven aan de oproep van `setEndImage()`:

```
endScore.setEndImage(score);
```

En deze parameter ook aan de methode zelf toe te voegen:

```
public void setEndImage(int score) {
```

Vervolgens kunnen we bepalen of de speler gewonnen of verloren heeft:

```
if(score == 12) {
    resultaat = "Je hebt gewonnen..!";
}
else {
    resultaat = "Je hebt verloren..!";
}
```


Extra opdrachten

Gnomus kan al in de verte zien waar Garp is en vervolgens op Garp afgaan om hem te doden. Dat kun je programmeren met behulp van de methodes *int getX()* en *int getY()* van het object Garp. Het object Garp moet dan wel bekend zijn aan Gnomus: `getObjectsInRange(100, "Garp.class");`

Je kunt het spel ook met zijn twee spelers spelen, waarbij een speler Garp aanstuurt en de andere speler Gnomus. Gnomus wordt op dezelfde manier aangestuurd als Garp maar dan met bijvoorbeeld de toetsen: Z, X, C en S, aan de linkerkant van het toetsenbord. De pijltjestoetsen zitten namelijk aan de rechterkant van het toetsenbord. Als Gnomus een bom raakt, ontploft de bom en heeft Garp gewonnen.

Je kunt het spel ook zo veranderen dat er twee Garps zijn die beide door een speler worden aangestuurd. De Garp die de meeste diamanten heeft verzameld als alle diamanten uit het spel zijn, heeft gewonnen. Als de twee Garps tegen elkaar botsen, dan heeft de Garp met de meeste diamanten gewonnen. Als een van de Garps eerder dood gaat door Gnomus of door een bom, dan gaat de andere Garp door totdat hij of ook dood gaat of alle diamanten die nog in het spel zijn, heeft verzameld. In het laatste geval heeft deze Garp gewonnen. Het bijhouden van de scores moet dan ook aangepast worden.

Bijlage 1: Kleuren

Er zijn in Java standaard 13 kleuren:

Color.black	zwart
Color.blue	blauw
Color.cyan	groenblauw
Color.darkGray	donkergrijs
Color.gray	grijs
Color.green	groen
Color.lightGray	lichtgrijs
Color.magenta	paars
Color.orange	oranje
Color.pink	roze
Color.red	rood
Color.white	wit
Color.yellow	geel

Deze kleuren kun je met `setColor()` en `setBackground()` gebruiken.

Je kunt ook met het RGB-systeem kleuren maken. RGB staat voor rood, groen en blauw. Pixels bestaan uit drie puntjes die ieder één van de drie kleuren weergeven. Door de intensiteit van zo'n puntje te wijzigen, krijg je een andere kleur. Een rode achtergrond krijg je door alleen het rode puntje aan te zetten. Dat doe je door bijvoorbeeld de volgende opdracht:

```
g.setColor( new Color( 255, 0, 0 ) );
```

Het groene en het blauwe puntje staan uit door daar 0 mee te geven. Het rode puntje staat aan. Een hogere waarde dan 255 geeft een foutmelding bij de uitvoering van de applet. Met dit systeem heb je ongeveer 16 miljoen mogelijkheden.

Bijlage 2: De klasse Score

Als je de klasse Score niet hebt, vind je hieronder de code. Maak een nieuwe klasse aan met de naam **Score** onder **Actor** zonder afbeelding en neem de onderstaande code over:

```
import greenfoot.GreenfootImage;
import greenfoot.Actor;
import java.awt.Font;
import java.awt.FontMetrics;
import java.awt.Dimension;

/**
 * Deze klasse is een hulpklasse om al te ingewikkelde code te voorkomen.
 * Er is een methode die de hoogte en de lengte van een string berekent.
 * Vooraf moet het juiste font voor de image gezet zijn zodat de juiste maten van de string
 * berekend kunnen worden.
 *
 * @author Ton van Beuningen
 * @version 19-10-2012
 */
public abstract class Score extends Actor
{
    /**
     * De methode getTextDimensions berekent de breedte en de hoogte van de string in
     * pixels.
     *
     * @param image De Greenfootimage waarin de string wordt geplaatst.
     * @param tekst de String waarvan de breedte en de hoogte wordt berekend.
     * @return Dimension met daarin de breedte en de hoogte van de string
     */
    public Dimension getTextDimensions(GreenfootImage image, String tekst) {
        int height, width;
        Font font;

        font = image.getFont();
        FontMetrics metrics = image.getAwtImage().getGraphics().getFontMetrics(font);
        width = metrics.stringWidth(tekst);
        height = metrics.getLeading() + metrics.getAscent() + metrics.getDescent();
        return new Dimension(width, height);
    }
}
```