

1. Greenfoot



Greenfoot (<http://www.greenfoot.org/>) is een **ontwikkelingsomgeving** gebaseerd op de **Java programmeertaal**.

Greenfoot is ontworpen om te **leren programmeren met gebruik van Java** en op een eenvoudige manier **2D spellen** (in een vlakke wereld) te maken.

Greenfoot is gratis en kan gebruikt worden op zowel Windows, Mac OS X als Linux.

2. Java



De Java taal is een “object geörinteerde” programmeertaal die gemaakt werd rond 1996.

Vandaag is Java één van de meest gebruikte talen in de wereld. De taal wordt gebruikt om de meest uiteenlopende toepassingen te maken (spelletjes, bank toepassingen, robots, smartphone toepassingen(Android),...)

Het is een taal die toelaat om toepassingen te schrijven die ongewijzigd op verschillende platformen kan werken (Windows, Mac, Linux, ...).

Er zijn veel websites en boeken over Java. Hier alvast enkele linken:

<http://myflex.org/books/java4kids/java4kids.htm>

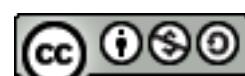
<http://www.bluej.org/>

3. Klassen en objecten

Vooraleer we starten met Greenfoot, enkele woorden over '**klassen**'(**classes**) en '**objecten**' in Java.

Java is een '**object geörinteerde taal**', dit wil zeggen dat een toepassing bestaat uit '**objecten**'.

Laat ons een concreet voorbeeld nemen om dit duidelijk te maken : een mannetje in een spel.

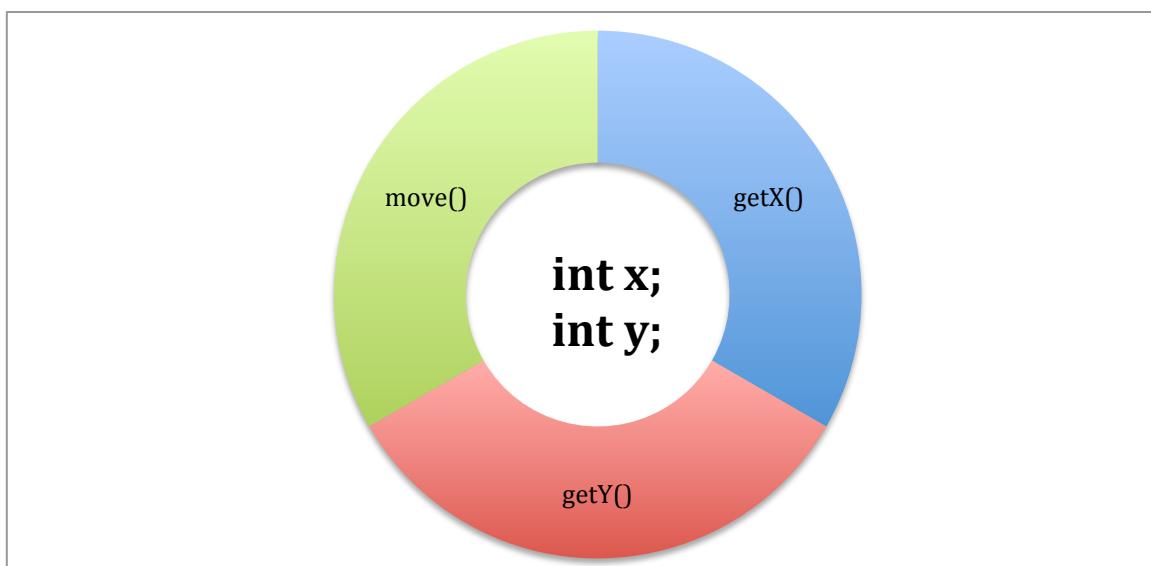


Elk object wordt gemodelleerd door twee belangrijke delen:

- de **toestand**, die bestaat uit een verzameling **eigenschappen** of **attributen** ('properties','attributes')
- het **gedrag**, dat bestaat uit een verzameling **acties** of **operaties** die het kan doen ('methods','operations')

In ons voorbeeld van het mannetje:

- de **toestand** : plaats in het spel, richting waarin het beweegt
- het **gedrag** : de bewegingen die je het kan laten doen (draaien, lopen, ...)



Om een object te beschrijven zullen we een '**klasse**'('class') definiëren. Een 'klasse' is een soort van 'malle' of 'template' om objecten mee te maken. De klasse beschrijft de eigenschappen waar alle objecten van deze klasse aan moeten voldoen.

De objecten noemen we ook realisaties van deze klasse ('**instances**').

We zullen deze concepten en de basis van Java tijdens de workshop verduidelijken met code voorbeelden in Greenfoot.

```

public class GameElement {

    private int x;
    private int y;

    public GameElement(int initX, int initY) {
        x = initX;
        y = initY;
    }

    public int getX() {
        return x;
    }

    public int getY() {
        return y;
    }

    public void setX(int newX) {
        x = newX;
    }

    public void setY(int newY) {
        y = newY;
    }

    public void move(int xMove, int yMove) {
        x = x + xMove;
        y = y + yMove;
    }
}

```

De klasse

- is als '**public**' aangekondigd. Dit betekent dat objecten die we later gaan maken realisaties van deze klasse gaan kunnen aanspreken.
- Heeft een **naam** die met een **hoofdletter** begint. Objecten ('instances') gaan we altijd met een kleine letter laten beginnen zodat we duidelijk het onderscheid kunnen zien. Als de naam uit verschillende woorden bestaat, worden ze aan elkaar 'geplakt' waarbij ze hun hoofdletter behouden.

De eigenschappen/attributen (properties):

- De toestand wordt meestal bovenaan geplaatst, daaronder hoe we 'instances' kunnen maken van de klasse en daarna de operaties die we kunnen uitvoeren.
- Elke eigenschap ('property') wordt meestal als '**private**' aangekondigd, dit betekent dat deze eigenschap enkel binnen het object gebruikt kan worden. Andere objecten kunnen niet aan deze eigenschappen. Dit wordt ook soms '**encapsulatie**' genoemd.
- Elke eigenschap begint met een kleine letter.
- Elke eigenschap heeft een '**type**', in dit geval is het een **int** (nummer), maar het zou ook een klasse kunnen zijn (vb. Een t-shirt).

De acties/operaties (methods)



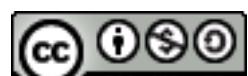
- Operaties worden meestal als '**public**' aangekondigd zodat ze kunnen aangeroepen worden door andere objecten in de toepassing. Soms zijn ze 'private', dan kunnen ze enkel binnen de klasse benadert worden (vb. Als je een hulp methode wil maken die enkel zin heeft in de klasse zelf).
- De naam start met een kleine letter
- Bij elke methode moet vermeld worden wat ze als **resultaat** heeft. Als er niks wordt teruggegeven door de methode, wordt het speciale woord '**void**' (=leeg) gebruikt.
- Het kan zijn dat je om een methode aan te roepen gegevens moet meegeven (vb. Ga 5 meter vooruit). Deze gegevens worden de '**argumenten**' van de methode genoemd. Elk argument heeft een **type** (klasse) en een **naam** (startend met kleine letter). Als je methode een resultaat teruggeeft, moet je dit doen door '**return**' met daarna het resultaat op een lijn te plaatsen.
- Daar de eigenschappen meestal 'private' zijn, worden methoden toegevoegd om deze eigenschappen op te vragen (zogenaamde '**getter**' operaties) Dit zijn de methoden **getX()** en **getY()** uit ons voorbeeld
 - o ze hebben de naam `get<Naam Van De Eigenschap>`
 - o ze hebben geen argumenten
 - o het type dat ze teruggeven is dat van de eigenschap
 - o ze geven simpelweg de eigenschap terug:
`return <argument>;`
- Gelijkwaardig, wanneer een object schrijf toegang wil geven aan zijn interne toestand(of een gedeelte daarvan), worden '**setter**' methoden gemaakt. Methoden zoals **setX()** en **setY()** in ons voorbeeld.
 - o Hun naam is `set<Naam Van De Eigenschap>`
 - o Ze hebben één argument van hetzelfde type als het type van de eigenschap
 - o Ze hebben geen resultaat
 - o Ze zetten eenvoudigweg de eigenschap gelijk aan het argument:
`<property> = <argument>;`

Constructors

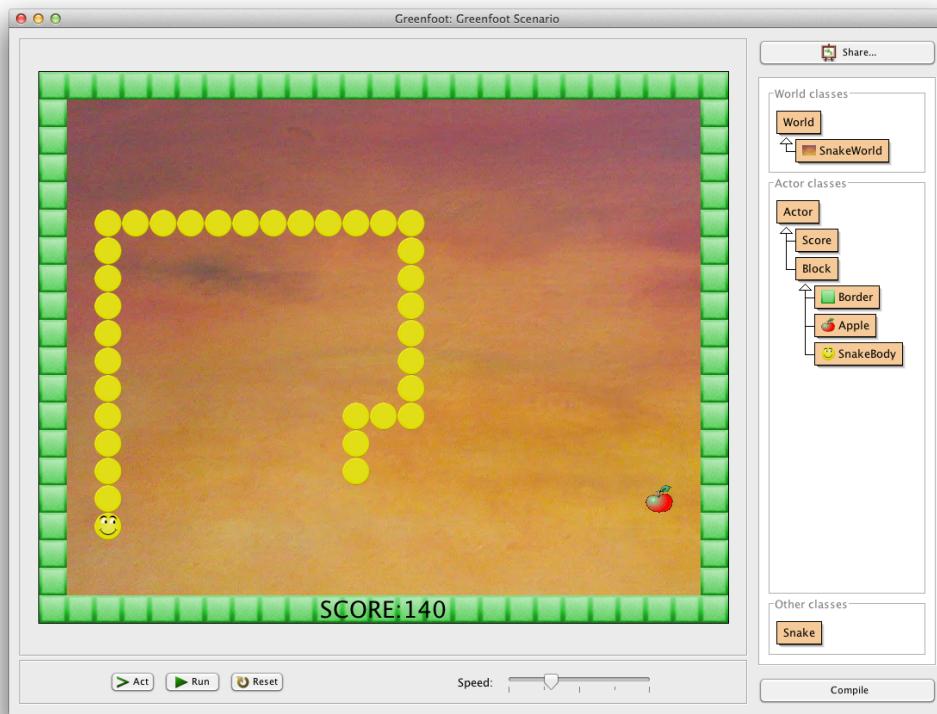
- Constructors zijn gelijkaardig aan acties, met dat verschil dat er **geen resultaat aangegeven** wordt (het resultaat is immers altijd een object ('instance') van de klasse).
- Hun naam moet identiek zijn aan de naam van de klasse waarin ze staan.
- De bedoeling van een constructor is om een object ('instance') van die klasse te maken. Je kan een constructor als volgt aanroepen (volgens het voorbeeld hierboven): `GameElement spelletje = new GameElement(2,3);`
 - o Hiermee maak je een instance met de naam 'spelletje' aan waarbij x de waarde 2 zal hebben en y de waarde 3.
- Als je zelf geen constructor maakt, dan voegt de compiler zelf een constructor toe zonder argumenten (Java voegt dit automatisch toe).

4. Voorstelling van het 'Bobby Snake' spel

Een toepassing binnen Greenfoot wordt een scenario genoemd. Je kan vele scenario's vinden op de website van Greenfoot: <http://www.greenfoot.org/scenarios>



In deze workshop gaan we een spel scenario maken die we 'Bobby Snake' noemen. Je kan het 'Bobby Snake' scenario openen door op het bestand '**project.greenfoot**' in de '**Bobby Snake**' map te dubbelklikken.



Het doel van de workshop is om dit slangenspel te maken. Je kan Bobby, de slang, met de pijltjestoetsen van het toetsenbord laten bewegen. Elke keer als Bobby een appel eet zal hij groeien, maar let op: hij mag niet tegen de rand van het spel botsen, of zichzelf opeten!

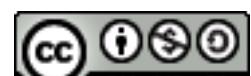
5. Een nieuw scenario starten

We gaan nu een nieuw scenario maken en alle stappen uitleggen die nodig zijn om het spel te maken. Wanneer je een nieuw scenario maakt, ziet het Greenfoot scherm er als volgt uit:

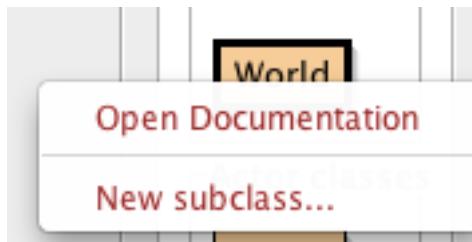


Op de rechterkant van het scherm zie je twee belangrijke klassen:

- **'World'**: dit is het spel 'wereld'. Het 'wereld' object zal het scherm tekenen.
- **'Actor'**: de verschillende spel elementen die in het spel leven zullen **'Actors'** zijn (Bobby, appels, rand blokken).



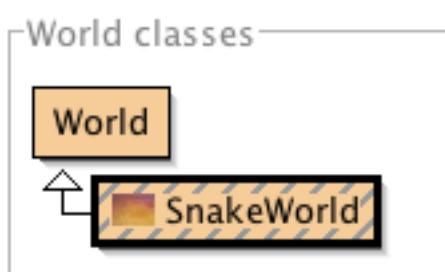
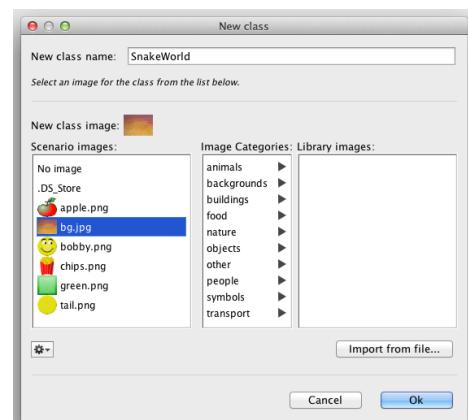
6. Aanmaken van de Wereld ('World')



Klik rechts op 'World' en kies dan voor 'New subclass...'

In het dialoog scherm, kies voor de klasse naam '**'SnakeWorld'**'.

We gaan ook een achtergrond kiezen voor onze wereld, laat ons daarvoor het plaatje met de naam '**'bg.jpg'**' kiezen.



Nadat we op 'Ok' geklikt hebben, maakt Greenfoot een nieuwe klasse voor ons aan die een onderklasse(subclass) is van 'World'

Als we nu dubbel-klikken op onze nieuwe klasse, zal Greenfoot een scherm openen waarin we de Java code kunnen bewerken:

```

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

/**
 * Write a description of class SnakeWorld here.
 *
 * @author (your name)
 * @version (a version number or a date)
 */
public class SnakeWorld extends World
{
    /**
     * Constructor for objects of class SnakeWorld.
     *
     */
    public SnakeWorld()
    {
        // Create a new world with 600x400 cells with a cell size of 1x1 pixels.
        super(600, 400, 1);
    }
}

```

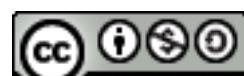
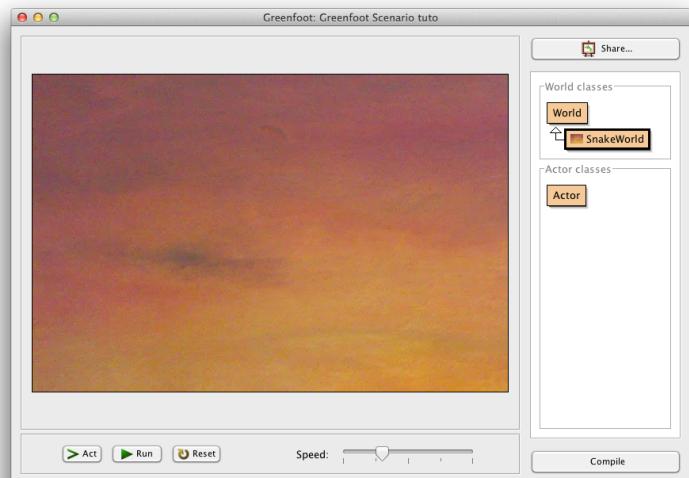
We komen straks hierop terug, maar we zullen nu eerst deze code eens uitvoeren.

Om het spelletje te starten moet je eerst **'compileren'** (de java code omzetten naar iets wat Greenfoot kan uitvoeren). Klik daartoe rechts onderaan op de 'Compile' knop. In ons voorbeeld wordt enkel een achtergrond getoond.

In Greenfoot worden klassen die opnieuw moeten gecompileerd worden als **gearceerd** getoond. Vooraleer je iets kan uitvoeren zal je dus altijd eerst moeten compileren door op de '**Compile**' knop onderaan rechts te klikken.

Laat ons nu terugkeren naar onze code in de klasse '**SnakeWorld**':

```
public class SnakeWorld extends World
```



Het gedeelte '**extends World**' geeft aan dat onze 'SnakeWorld' klasse de karakteristieken '**overerfd**' van de **World** klasse.

De 'World' klasse is ingebakken in Greenfoot en dient om het hoofdscherm van het spel te maken.

We noemen 'SnakeWorld' ook wel eens een 'onder-klasse' (sub-class) of 'kind-klasse' (child-class) en 'World' een ouder-klasse (parent-class).

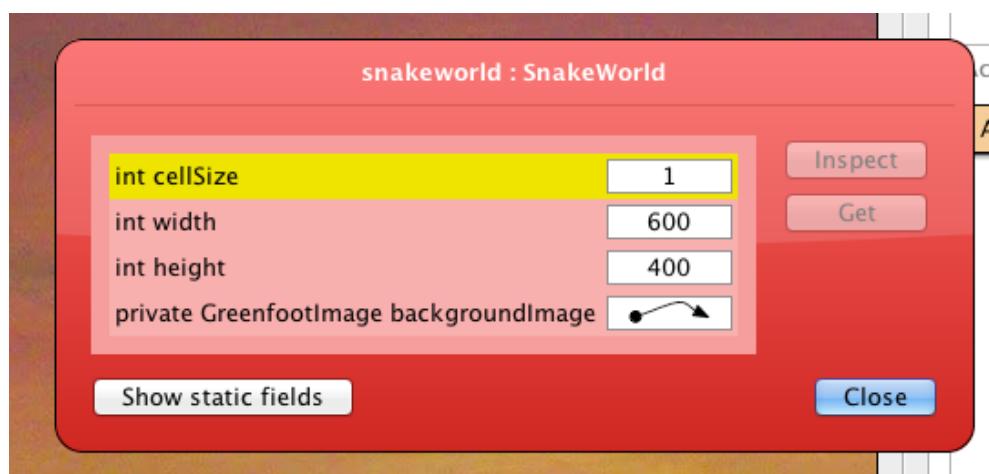
Hierna hebben we een '**constructor**'. Je herkent dit, doordat er geen resultaat vermeld wordt en de naam van de methode is identiek als de naam van de klasse.

```
public SnakeWorld() {  
    super(600, 400, 1);  
}
```

De uitdrukking (of 'statement') **super(600, 400, 1)** is een aanroep naar de **constructor** die in onze ouder-klasse staat, en die ons toelaat een Wereld object te maken door de volgende argumenten mee te geven:

- De eerste parameter is de **breedte** van het scherm (in puntjes of 'pixels')
- De tweede parameter is de **hoogte** van het scherm (in puntjes of 'pixels')
- De derde parameter geeft de **grootte** aan van de blokken.

Door rechts te klikken op het spel scherm kunnen we ook het wereld object inspecteren (klik op '**Inspect**').



Daar zien we dat het object inderdaad de meegeven parameters aan de constructor als toestand bevat.

Laat ons nu de code wijzigen, om met een scherm van 25 blokken breed en 20 blokken hoog te werken waarbij de blokken een grootte hebben van 32 bij 32 puntjes:

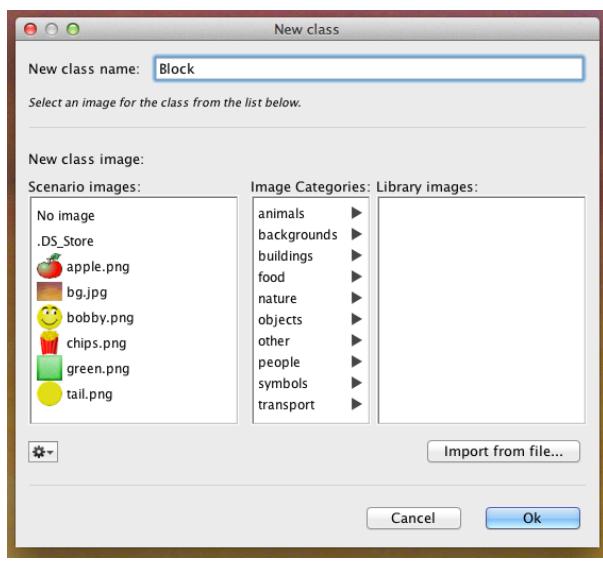
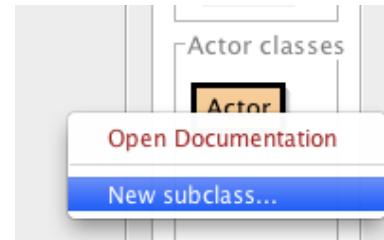
```
public SnakeWorld() {  
    super(25, 20, 32);  
}
```

Compileer je wijzigingen en controleer het resultaat door je wereld te inspecteren.

7. Aanmaken van een Blok

We gaan nu de '**Actors**' aanmaken die deel gaan uitmaken van ons spel.

Om dit te doen, klik rechts op de '**Actor**' klasse en selecteer 'New subclass...'



We gaan verschillende types blokken maken die we in ons spel zullen gebruiken. Met dat doel gaan we een klasse '**Block**' maken als kind van de ingebakken Greenfoot klasse '**Actor**' (daarmee krijgt het meteen alle karakteristieken van de 'Actor' klasse).

Voorlopig gaan we klasse laten zoals hij is.

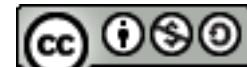
Dubbel klik op de klasse en verwijder de 'act()' methode (we hebben die niet nodig).



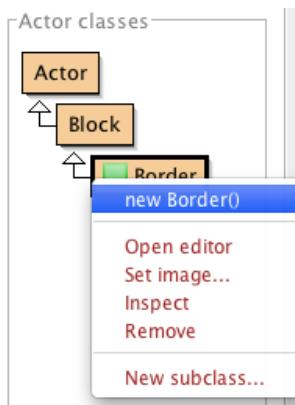
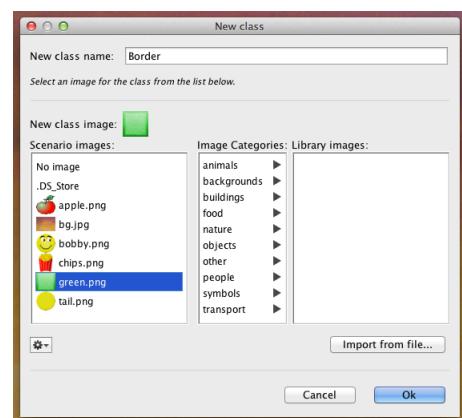
8. Aanmaak van een rand blok('Border')



Laat ons nu een onder-klasse(sub-class) maken van onze 'Blok' klasse die we daarnet hebben gemaakt.



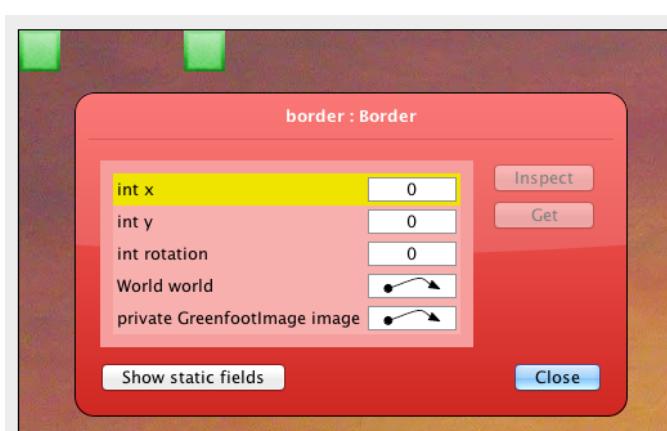
Geef de naam '**Border**' en kies als plaatje '**green.png**'.



Er verschijnt nu een nieuwe klasse 'Border' onder de 'Block' klasse (die zelf weer onder de 'Actor' klasse hangt).

Door rechts te klikken op de 'Border' klasse, kunnen we op 'new Border()' klikken, waardoor een nieuw object (instance) van onze klasse zal aangemaakt worden.

Met behulp van de muis kan je de blok (object) overal op het scherm plaatsen.

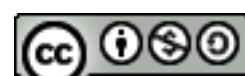


Je kan dit herhalen zoveel je wil om meerdere blokken aan te maken. Je kan elke blok inspecteren (via 'inspect') om de toestand van elke blok te bekijken.

Voor elke blok zal je de positie zien (x en y).
Je kan zien dat als een blok helemaal linksboven geplaatst is, zowel x als y nul zijn (0,0).

Als je blokken naar rechts verplaatst wordt de **x positie** groter en analoog als je blokken naar beneden verplaatst wordt de **y positie** groter.

We kunnen dit als volgt voorstellen:



(0,0)	(1,0)	(2,0)	...
(0,1)	(1,1)	(2,1)	...
(0,2)	(1,2)	(2,2)	...
...

9. Tekenen van de Rand(Border)

We gaan nu een rand te tekenen in onze wereld.

We beginnen met het plaatsen van één blokje en zullen dit daarna uitbreiden om de volledige rand te vullen.

Klik op de 'SnakeWorld' klasse om de editor te openen en voeg volgende code toe:

```
public SnakeWorld()
{
    super(25, 20, 32);
    addObject(new Border(), 0, 0);
}
```

Nadat je de code gecompileerd hebt verschijnt er een blok in de linker boven hoek.

De **addObject** methode is een methode die geërfd is van de wereld klasse en laat toe om een 'Actor' te tonen op de aangegeven positie.

De '**new Border()**' is een aanroep naar de constructor van de Border klasse om een nieuw Border object aan te maken.

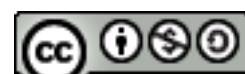
Om nu op deze manier blokken te tonen rondom het hele scherm zou dit nogal langdradig zijn, daar we voor elke blok een lijn met 'addObject' zouden moeten toevoegen.

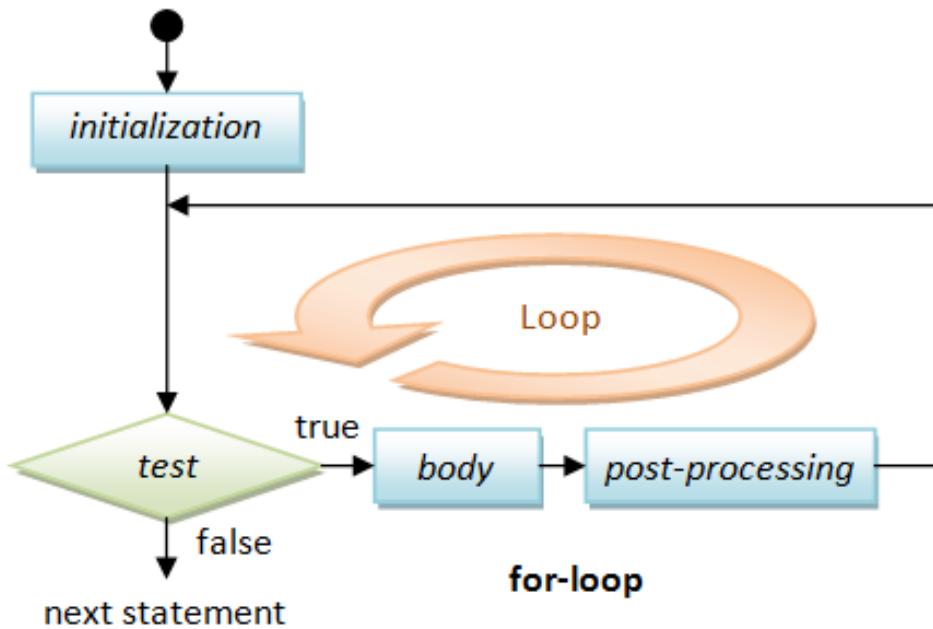
We gaan hier natuurlijk de computer het werk laten doen, door iets te gebruiken (**'for loop'**) zodat we dezelfde code kunnen herhalen.

Een '**for loop**' ziet er in Java als volgt uit (**syntax**):

```
for ( initialization ; test ; post-processing ) {
    body ;
}
```

Het volgende schema illustreert wat hiermee bereikt wordt:





Om de randen van ons spel te tekenen gaan we twee 'for loop'-s gebruiken. Eentje om de horizontale rijen te tekenen en eentje om de verticale rij te tekenen. Hiervoor laten we de x waarde veranderen van 0 tot de breedte van het scherm (merk op dat we bij 0 beginnen en dus eindigen bij de breedte min één). Bij elke stap gaan we zowel een blokje bovenaan als onderaan tekenen.

```

public SnakeWorld()
{
    super(25, 20, 32);

    for (int x = 0; x < getWidth(); x++) {
        addObject(new Border(), x, 0);
        addObject(new Border(), x, getHeight() - 1);
    }

    for (int y = 0; y < getHeight(); y++) {
        addObject(new Border(), 0, y);
        addObject(new Border(), getWidth() - 1, y);
    }
}

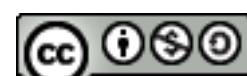
```

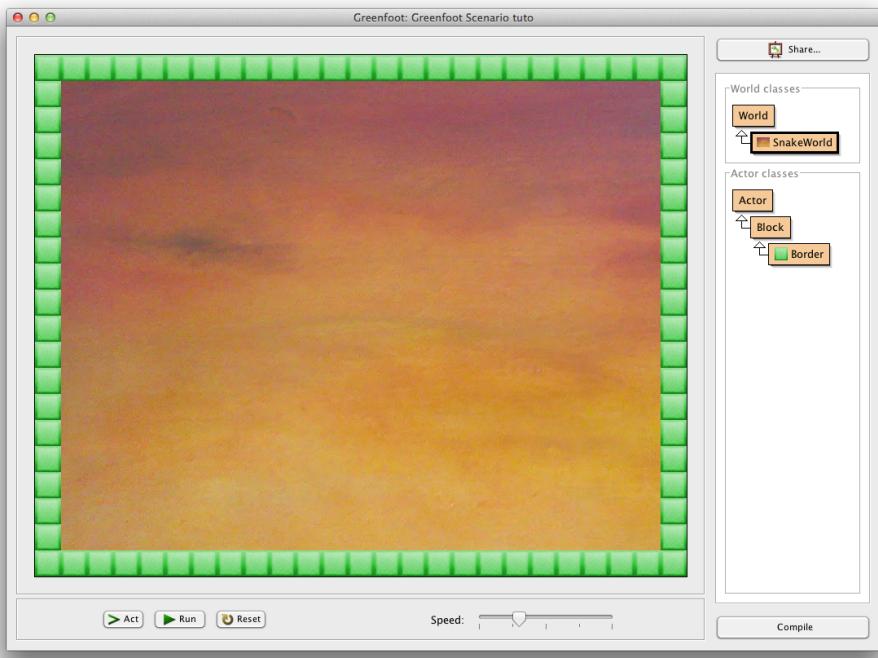
De tweede 'for loop' is analoog, maar nu gaan we de y waarde laten veranderen en tekenen we telkens links en rechts een blokje.

Een aantal bijkomende opmerkingen:

- variabelen in Java zijn altijd getypeerd, vandaar dat we **int x = 0** moeten schrijven om het type van x aan de compiler duidelijk te maken.
- Een lijn als '**variable = expression**' betekent het toekennen van een waarde aan een variabele. Bijvoorbeeld **x=0** betekent ken aan de variabele 'x' de waarde '0' toe.
- De verschillende opdrachten ('statements'), moeten in Java door een ';' gescheiden worden.
- De uitdrukking '**x++**' is een korter manier om te schrijven **x = x + 1** (dus verhoog x met 1)
- De methoden **getWidth()** and **getHeight()** worden geërfd van de '**World**' klasse en laten toe om de breedte en hoogte van de wereld op te vragen.

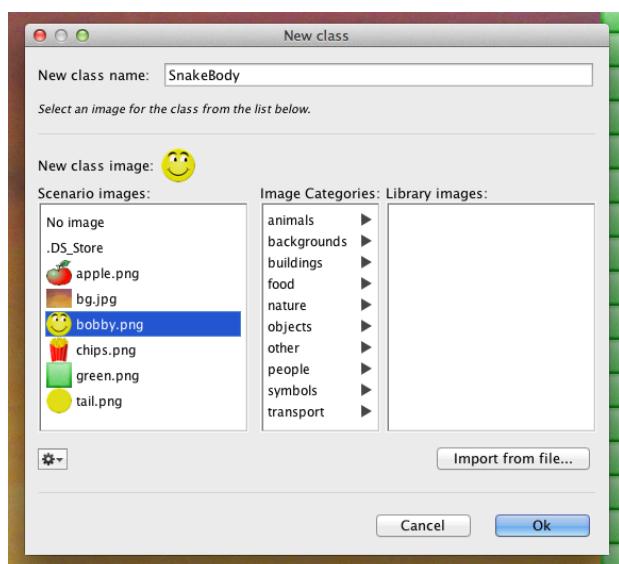
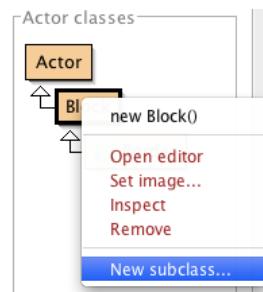
Compileer je code. Daarna moet je volgend resultaat hebben:





10. Display Bobby's head

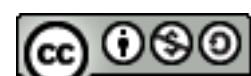
Maak een nieuwe sub-klasse van onze 'Block' klasse.



Geef het de naam '**SnakeBody**' en kies het plaatje van Bobby ('**bobby.png**').

De slang in ons spel zal een lijst (list) van 'SnakeBody'-s zijn. Daarom gaan we in '**SnakeWorld**' een eigenschap definiëren zodat de lijst van 'SnakeBody'-s een onderdeel van de toestand wordt.

Open de 'SnakeWorld' klasse en voeg de volgende lijn toe:



```
public class SnakeWorld extends World
{
    private LinkedList<SnakeBody> snake = new LinkedList<SnakeBody>();
```

Deze lijn maakt een nieuwe lege lijst aan die '**SnakeBody**' objecten kan bevatten. Deze lijst kunnen we nu aanspreken in de code die we gaan toevoegen in de 'SnakeWorld' klasse.

Om de klasse '**LinkedList**' te kunnen gebruiken moeten we bovenaan in '**SnakeWorld**' nog volgende code toevoegen:

```
import greenfoot.*;
import java.util.*;
```

Laat ons nu het eerste element in de lijst toevoegen. We gaan dit doen in de constructor van de klasse 'SnakeWorld' zodat als de wereld aangemaakt wordt, de kop van 'Bobby' verschijnt..

De eerste lijn maakt een object aan van het type '**SnakeBody**'.

Op de tweede lijn voegen we dit object toe aan de snake lijst (op het einde van de lijst) door de methode '**add()**' aan te roepen

```
public SnakeWorld()
{
    super(25, 20, 32);

    SnakeBody body = new SnakeBody();
    snake.add(body);
    addObject(body, 2, 2);
```

Op de derde lijn voegen we de 'SnakeBody' toe aan de wereld op positie (2,2). We gebruiken hiervoor de methode '**addObject()**' die we overgeerfd hebben van de 'World' klasse.

Compileer je wijzigingen en je zou nu 'Bobby' moeten zien verschijnen op positie (2,2)!



11. Bobby's Bewegingen

Om te weten hoe Bobby beweegt (in welke richting), gaan we 2 nieuwe eigenschappen toevoegen, één voor de beweging in de richting van de x-as en één voor de beweging in de richting van de y-as.

```
public class SnakeWorld extends World
{
    private LinkedList<SnakeBody> snake = new LinkedList<SnakeBody>();
    private int dx = 1;
    private int dy = 0;
```

Om met een beweging van Bobby naar rechts te starten hebben we dx op 1 ingesteld (geïnitialiseerd) en dy op 0.

Laat ons nu een methode **act()** aan de '**SnakeWorld**' klasse toevoegen. Deze methode zal door Greenfoot voordurend worden opgeroepen om het spel te laten vooruitgaan. Greenfoot roept deze methode aan op alle object van de '**World**' en op alle '**Actors**' die aan '**World**' zijn toegevoegd.

Hier volgt een eerste versie van de **act()** methode:

```
public void act()
{
    //on remplace l'image de la tête
    SnakeBody head = snake.getLast();
    head.setImage("tail.png");

    //crée une nouvelle tête
    SnakeBody newHead = new SnakeBody();
    int newHeadX = head.getX() + dx;
    int newHeadY = head.getY() + dy;

    //ajoute la nouvelle tête à la liste et au world
    addObject(newHead, newHeadX, newHeadY);
    snake.add(newHead);
}
```

In deze versie vervangen we de kop (het laatste element in de lijst) door het plaatje 'tail.png'. We kunnen het laatste element in een lijst opvragen door aan het **lijst** object de methode **getLast()** aan te roepen.

Vervolgens maken we een nieuwe kop aan (**new SnakeBody()**) en we berekenen de plaats waar de nieuwe kop moet komen, gebaseerd op de huidige plaats van de kop en de richting van de beweging (dx en dy).

Finaal voegen we de kop toe op de berekende plaats in de wereld, en we voegen ook de kop toe aan de lijst.



Compileer je wijzigingen en click daarna op 'Run'. Dit heeft tot gevolg dat Greenfoot de 'act()' methode voortdurend zal oproepen op alle actors.

We zien inderdaad dat de slang naar rechts gaat (we hadden immers dx=1 gesteld) en groeit met elke stap die het maakt tot we aan de rechter rand komen.



12. Beperken van de grootte van de slang

Om de grootte van de slang te beperken, zullen we bijhouden hoeveel elementen nog mogen toegevoegd worden aan de staart.

We gaan dit doen door een eigenschap '**tailCounter**' toe te voegen aan de klasse 'SnakeWorld'

```
public class SnakeWorld extends World
{
    private LinkedList<SnakeBody> snake = new LinkedList<SnakeBody>();
    private int dx = 1;
    private int dy = 0;
    private int tailCounter = 5;
```

We gaan nu deze waarde gebruiken in de **act()** methode:

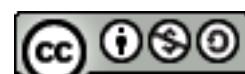
- Als de waarde van tailCounter gelijk is aan nul, verwijderen we het uiterste van de staart (dit is eigenlijk het eerste element in onze lijst).
- Anders verlagen we de tailCounter met 1

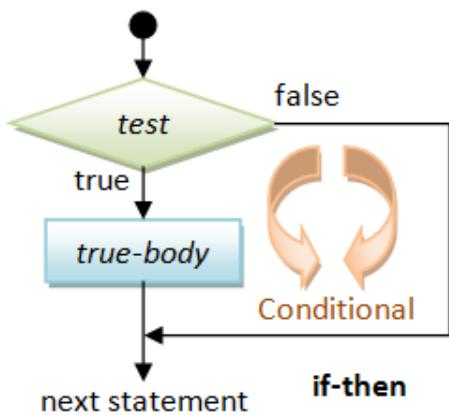
Om een dergelijke test te maken gaan we een nieuwe constructie gebruiken in Java: de '**if constructie ('if statement')**'

Een '**if statement**' ziet er als volgt uit:

```
if ( condition ) {
    true-body ;
}
```

Het volgende schema maakt zijn werking duidelijk:



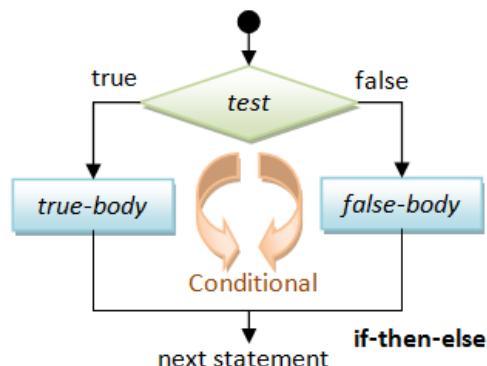


Een uitgebreidere vorm van de 'if statement' is de '**if-then-else**'. Dit gebruik je als je eveneens iets anders wil doen indien de voorwaarde niet voldaan is:

```

if ( condition ) {
    true-body ;
} else {
    false-body ;
}
  
```

Volgend schema maakt duidelijk hoe dit uitgevoerd wordt:



```

public void act()
{
    //on remplace l'image de la tête
    SnakeBody head = snake.getLast();
    head.setImage("tail.png");

    //crée une nouvelle tête
    SnakeBody newHead = new SnakeBody();
    int newHeadX = head.getX() + dx;
    int newHeadY = head.getY() + dy;

    //ajoute la nouvelle tête à la liste et au world
    addObject(newHead, newHeadX, newHeadY);
    snake.add(newHead);

    if (tailCounter == 0) {
        SnakeBody tail = snake.removeFirst();
        removeObject(tail);
    }
}
  
```

Deze kennis kunnen we nu gebruiken om onze `act()` methode aan te vullen.

De **removeFirst()** methode op een **LinkedList** verwijdert het eerste element uit de lijst en geeft het terug als resultaat van de aanroep.



De **removeObject()** methode, overgeerfd van World, verwijdert het meegegeven 'Actor' object (en verwijdert het dus effectief van het scherm).

Noot: '**tailCounter--;**' is een verkorte notatie voor
'tailCounter = tailCounter - 1;' (analoog aan de '**++**' notatie)

De eerste 5 keer dat Greenfoot de **act()** methode aanroeft zal de slang groeien en zal **'tailCounter'** telkens met 1 vermindert worden.

Van zodra de waarde van **'tailCounter'** gelijk is aan 0, zal de slang stoppen met groeien omdat we dan telkens we een nieuwe kop toevoegen, ook het laatste deel van de staart verwijderen.



13. De richting veranderen

We gaan er nu voor zorgen dat we de slang in een andere richting kunnen laten bewegen.

We gaan dit doen door te reageren op de pijltjes toetsen van ons toetsenbord.

Greenfoot heeft hiervoor een ingebakken methode om te testen of een bepaalde toets ingedrukt is of niet:

Greenfoot.isKeyDown(key)

Wanneer we deze methode aanroepen, krijgen we ofwel 'true' (wanneer 'key' is ingedrukt) ofwel 'false' terug(wanneer 'key' niet is ingedrukt). Het argument 'key' is de naam van de toets (in ons geval 'left', 'right', 'up', 'down')

```
private void changeDirection() {
    if (Greenfoot.isKeyDown("left") && dx == 0 ) {
        dx = -1;
        dy = 0;
    } else if (Greenfoot.isKeyDown("right") && dx == 0 ) {
        dx = 1;
        dy = 0;
    } else if (Greenfoot.isKeyDown("down") && dy == 0 ) {
        dx = 0;
        dy = 1;
    } else if (Greenfoot.isKeyDown("up") && dy == 0 ) {
        dx = 0;
        dy = -1;
    }
}
```

We zullen dit gedrag nu groeperen in een aparte methode **'changeDirection()** die we daarna gaan aanroepen vanuit de **act()** methode.



Let op: als de slang naar links beweegt, mag je de slang niet naar rechts laten bewegen (Bobby mag niet naar zichzelf terugkeren)!

Vandaar dat bij elke test nog een bijkomende voorwaarde staat vb.

Greenfoot.isKeyDown("left") && dx == 0

- De uitdrukking '**&&**' betekent 'en'.
- De uitdrukking '**==**' test voor **gelijkheid**.
- Dus : 'als' je het linkerpijltje indrukt 'en' **dx==0** 'dan' ...

Het enige wat ons rest is deze methode aanroepen vanuit de **act()** methode.

```
public void act()
{
    changeDirection();
    // een normale lijn moet da
```



Compileer je wijzigingen en test het resultaat.

Je moet Bobby nu kunnen van richting laten veranderen met de pijltjes toetsen.

Op dit moment kan Bobby nog overal naartoe gaan, en zelfs zichzelf kruisen.

Dit ongewenst gedrag gaan we later verbeteren

Vraagjes:

- Was het echt nodig om die code in een aparte methode 'changeDirection()' te plaatsen?
- Waarom denk je dat we dit zo gedaan hebben?

14. Botsingen vermijden

Vooraleer we botsingen aanpakken, gaan we een eigenschap '**dead**' toevoegen in onze '**SnakeWorld**' klasse.



```

public class SnakeWorld extends World
{
    private LinkedList<SnakeBody> snake = new LinkedList<SnakeBody>();
    private int dx = 1;
    private int dy = 0;
    private int tailCounter = 5;
    private boolean dead = false;
}

```

Deze eigenschap zal ons toelaten het spel te stoppen wanneer Bobby botst met een obstakel.

De 'dood' eigenschap heeft als type '**boolean**'. Dit is een **type** die slechts twee waarden kan aannemen '**true**' (**waar**) en '**false**' (**vals**).

We kunnen nu de **act()** methode aanpassen zodat we niks meer doen als Bobby dood is.

Als **dead** waar is, keert de methode direct terug en het spel stopt dus.

```

public void act()
{
    if (dead) {
        return;
    }
    changeDirection();
}

```

We gaan nu ook een **dead()** methode toevoegen aan de 'SnakeWorld' om de waarde van de '**dead**' eigenschap aan te passen:

```

public void dead() {
    dead = true;
}

```

We gaan nu bekijken hoe we botsingen ('**collisions**') kunnen afhandelen. In de **act()** methode, net voor we de nieuwe kop tonen, zullen we testen of er al andere 'Block'-s zijn op de plaats waar we de nieuwe kop willen tonen. Op die 'Block'-s zullen we de '**collision**' methode aanroepen.

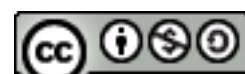
```

//crée une nouvelle tête
SnakeBody newHead = new SnakeBody();
int newHeadX = head.getX() + dx;
int newHeadY = head.getY() + dy;

List<Block> blocks = getObjectsAt(newHeadX, newHeadY, Block.class);
for(Block block : blocks) {
    block.collision(this);
}

//ajoute la nouvelle tête à la liste et au world
addObject(newHead, newHeadX, newHeadY);
snake.add(newHead);

```



De **getObjectsAt(x,y,type)** methode, overgeerfd van de World klasse, geeft een lijst van blokken op de (x,y) positie die van het meegegeven type zijn (wij gaan hier naar alle blokken vragen, we kunnen hiervoor de uitdrukking '**Block.class**' gebruiken, deze uitdrukking geeft ons het type van 'Block' terug).

We gaan nu over deze lijst van blokken lopen, en voor elke 'Block' in de lijst gaan we de '**collision**' methode aanroepen:

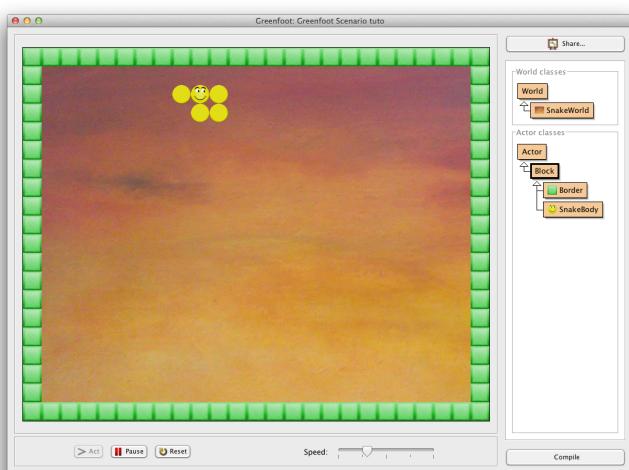
- Het argument dat we meegeven is onszelf (hier dus het huidig 'SnakeWorld' object).
- Het speciale woord 'this' is een verwijzing naar huidig object. Als 'this' vb. Gebruikt wordt in een methode van 'SnakeWorld' dan staat 'this' voor het huidig 'SnakeWorld' object.

Als je nu probeert te compileren zal er een fout gerapporteerd worden. Dit komt uiteraard omdat we 'collision()' nog niet geschreven hebben in de 'Block' klasse (en 'collision' wordt ook niet overgeerfd).

Open nu de 'Block' klasse en voeg de 'collision' methode toe:

Deze methode roept gewoon de methode `dead()` aan waardoor het spel stopt.

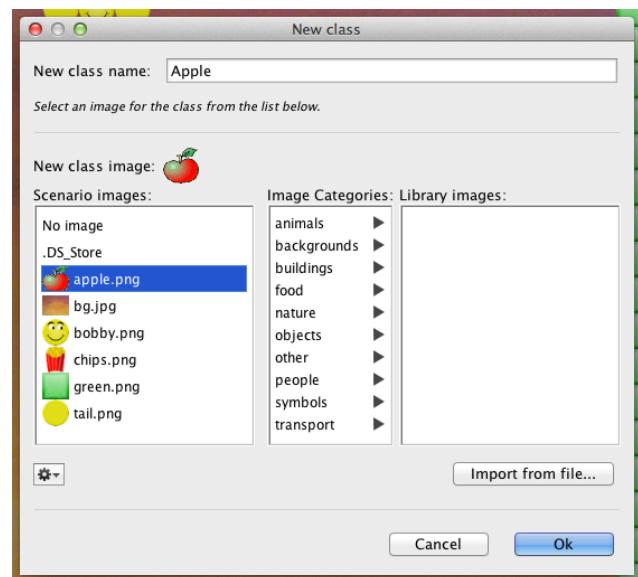
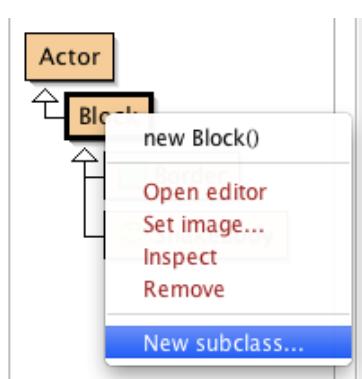
```
public class Block extends Actor
{
    public void collision(SnakeWorld world) {
        world.dead();
    }
}
```



Compileer de wijziging en test ze. Het spel stopt van zodra Bobby met de rand of met zichzelf botst.

15. Een appel toevoegen

We gaan een nieuwe kind-klasse maken van 'Block'. We geven ze de naam '**Apple**' en kiezen het plaatje '**apple.png**'.



Open de nieuwe '**Apple**' klasse, en verwijder de **act()** methode.

```
public class Apple extends Block
{
}
```

Keer nu terug naar de '**SnakeWorld**' klasse. Tijdens de initialisatie van het spel (dus in de 'constructor', nadat we Bobby gemaakt hebben), gaan we nu ook een 'Apple' object maken en het op een **willekeurige** plaats op het scherm plaatsen.

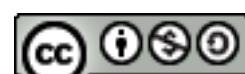
```
public SnakeWorld()
{
    super(25, 20, 32);

    SnakeBody body = new SnakeBody();
    snake.add(body);
    addObject(body, 2, 2);

    Apple apple = new Apple();
    addObject(apple,
              Greenfoot.getRandomNumber(getWidth()-2)+1,
              Greenfoot.getRandomNumber(getHeight()-2)+1);
}
```

De appel wordt gemaakt met de uitdrukking '**new Apple()**'. Vervolgens voegen we de appel toe aan de wereld met de methode ' **addObject()**' ingebakken in Greenfoot.

De plaats waar de appel moet komen wordt bepaald door een andere ingebakken Greenfoot methode: **Greenfoot.getRandomNumber()**. Deze methode neemt als argument een getal en geeft een willekeurig getal tussen 0 en het gegeven nummer



terug. Om de rand uit te sluiten moeten we als argument de breedte;hoogte - 2 meegeven. Bij het resultaat tellen we dan 1 op om binnen de rand te blijven.

Compileer je wijzigingen en test. Probeer een appel te pakken.
Wat gebeurt er?

...inderdaad het spel stopt... we hebben immers gezegd dat bij een botsing met een 'Block' het spel moet stoppen, en een appel is ook een blok!

We gaan dit in volgende paragraaf aanpakken.

16. Botsing met een appel

We gaan nu de 'collision' methode **overschrijven ('override')** in de **Apple** klasse.

In Java kunnen we **overgeërfde** methodes **overschrijven** om het gedrag in een onderklasse aan te passen.

```
public class Apple extends Block
{
    public void collision(SnakeWorld world) {
        world.grow(2);
        setLocation(
            Greenfoot.getRandomNumber(getWorld().getWidth()-2)+1,
            Greenfoot.getRandomNumber(getWorld().getHeight()-2)+1);
    }
}
```

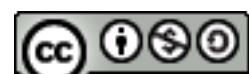
Wanneer de slang botst met een appel, mogen we de '**dead()**' methode niet aanroepen, maar moeten we aan de wereld vragen om de slang te laten groeien (vb. Met 2 of meer stukjes), daarbij moet de opgegeten appel ook verplaatst worden naar een andere plaats in het spel.

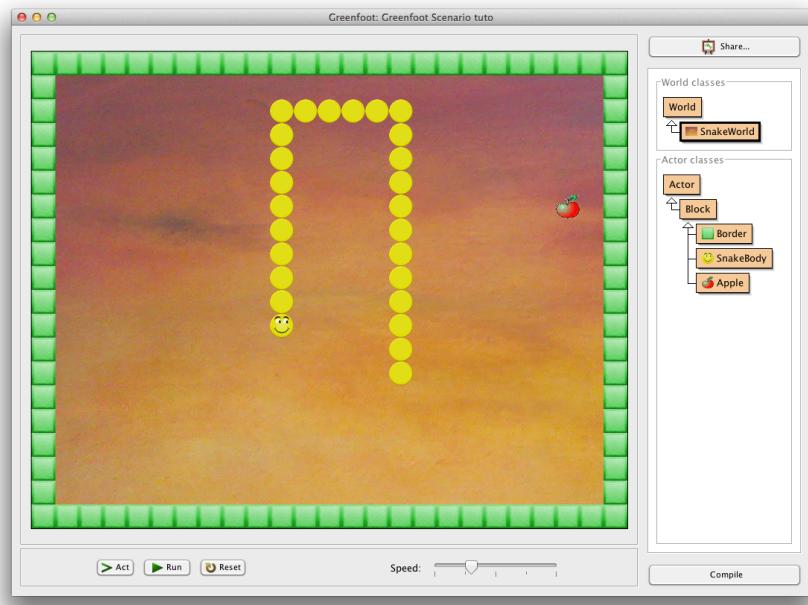
We moeten nu enkel nog de '**grow()**' methode toevoegen in de '**SnakeWorld**' klasse:

```
public void grow(int i) {
    tailCounter = tailCounter + i;
}
```

Deze methode moet enkel het aangegeven aantal elementen toevoegen aan onze slang (dmv **tailCounter**)

Compileer en test opnieuw. De slang groeit nu telkens hij een appel eet!





17. Geluid toevoegen

Om het spel verder te verbeteren gaan we geluid toevoegen telkens Bobby een appel opeet of sterft.

Greenfoot heeft een ingebakken methode om geluid af te spelen (fragment is de naam van het geluidsfragment wat moet afgespeeld worden):

Greenfoot.playSound(fragment);

Om geluid te laten spelen wanneer Bobby een appel eet, kan je volgende code toevoegen in de 'collision' methode van de '**Apple**' klasse:

```
public void collision(SnakeWorld world) {
    Greenfoot.playSound("slurp.mp3");
    world.grow(2);
    setLocation(
        Greenfoot.getRandomNumber(getWorld().getWidth()-2)+1,
        Greenfoot.getRandomNumber(getWorld().getHeight()-2)+1);
}
```

Compileer en test het resultaat!

De **playSound()** methode neemt dus de naam van een geluidsbestand als invoer parameter. Greenfoot zoekt dit bestand in de '**sounds**' map van je Greenfoot project.

Analoog kan je nu ook een geluid afspelen wanneer Bobby sterft, door volgend stuk code in de 'collision' methode van de '**Block**' klasse toe te voegen:

```
public void collision(SnakeWorld world) {  
    Greenfoot.playSound("dead.mp3");  
    world.dead();  
}
```

Compileer en test!

18. Einde

Proficiat! Je hebt net je eerste Greenfoot spel gemaakt!

Je kan nu nieuwe features toevoegen aan het spel of nieuwe spellen maken!

