

1. Greenfoot



Greenfoot (<http://www.greenfoot.org/>) est un **environnement de développement** basé sur le **langage de programmation Java**.

Il est principalement conçu pour **faciliter l'apprentissage du langage Java** et permet de facilement créer des jeux en 2D.

Greenfoot est un logiciel « libre » (gratuit) et est disponibles pour la plupart des ordinateurs sous Microsoft Windows, Mac OS X, ou Linux.

2. Java



Le langage Java est un langage de programmation « orienté objet » créé vers 1996.

C'est aujourd'hui un des langages les plus utilisés dans le monde. Il est utilisé pour développer des tas d'applications différentes (des jeux, des applications financières, des robots, les applications des téléphones Android, ...)

C'est un langage qui permet d'écrire des programmes qui fonctionnent sur plusieurs systèmes d'exploitation (Windows, Mac, Linux, ...).

Vous pouvez trouver de nombreux site web, et livres sur Java. Voici quelques liens :

<http://java.developpez.com/livres/javaEnfants/>

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

<http://www.siteduzero.com/informatique/tutoriels/apprenez-a-programmer-en-java>

3. Les classes et les objets

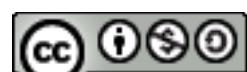
Avant de commencer avec Greenfoot, un petit mot d'explication sur les **classes** et les **objets** en Java.

Java est un langage de programmation « **orienté objet** ». Cela veut dire qu'une application Java est constituée « **d'objets** ».

Prenons un exemple concret d'objet : un personnage dans un jeu.

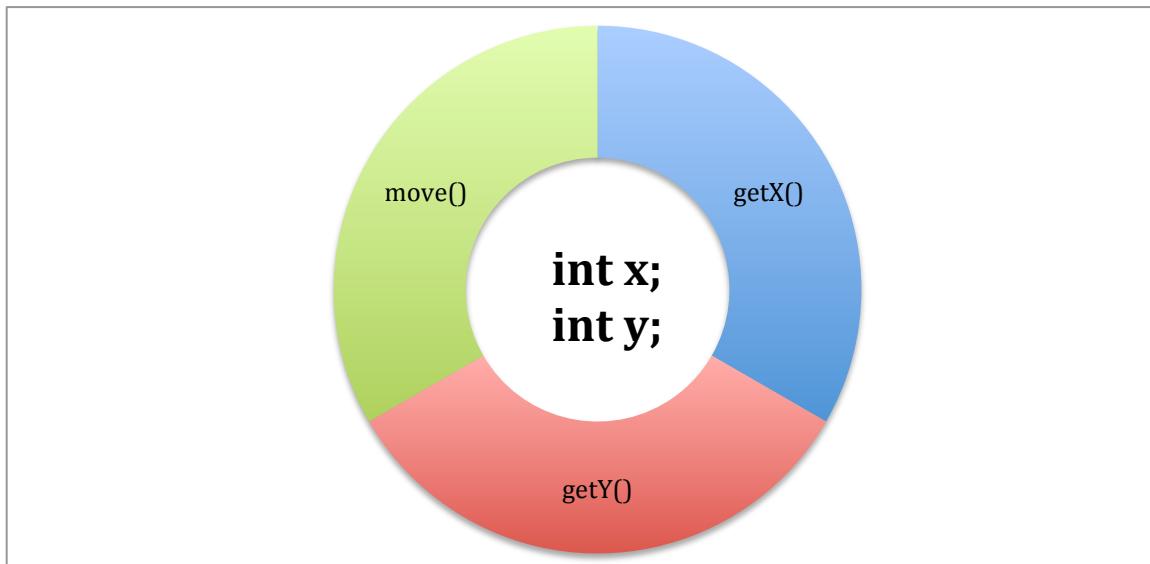
Chaque objet est constitué de deux parties importantes :

- son « état » constitué d'un ensemble de propriétés
- son « comportement », c'est-à-dire la liste des actions qu'il peut faire



Dans notre exemple d'un personnage de jeu :

- son état est constitué de sa position dans le jeu, sa direction de déplacement, ...
- son comportement est par exemple l'ensemble des mouvements qu'il peut faire.



Pour décrire un objet en Java, on va définir une **classe**. Une classe est une sorte de « modèle » qui permet de construire des objets.

Nous allons voir ces concepts et les bases du langage Java au travers des exemples de code, dans Greenfoot, tout au long de ce workshop. Mais commençons par un petit exemple pour illustrer les bases. Vous pourrez aussi revenir plus tard sur cette partie pour mieux comprendre les concepts de bases de Java au fur et à mesure du workshop.

```

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;
    }
}

```

La classe

- est déclarée « **public** » pour que tous les autres objets du système puissent interagir avec les objets de cette classe
- a un nom, qui par convention, commence toujours par une **majuscule**. Il n'y a jamais d'espace dans les noms en Java. Si le nom est composé de plusieurs mots, ceux-ci sont rattachés ensemble et chaque mot commence par une majuscule.

L'état de l'objet

- est déclaré au début de la classe,
- chaque propriété est habituellement déclarée « **private** » (privée), ce qui indique que seul l'objet peut y avoir accès. Les autres objets du système ne peuvent pas accéder à son état « interne ».
- chaque propriété a un **nom**, qui par convention commence par une minuscule.
- chaque propriété a un **type**. Ici, **int** représente le type « entier » (0, 1, 2, ...). On peut utiliser une autre classe pour typer une propriété.

Les méthodes de l'objet

- les méthodes sont souvent déclarées « **public** » pour pouvoir être appelée depuis d'autres objets du système. Elles peuvent aussi être « **private** » si elles ne peuvent être visibles que de l'objet lui-même.
- chaque méthode a un **nom**, qui par convention commence toujours par une minuscule.
- chaque méthode a un **type de retour** qui est le type de retour renvoyé par la méthode. Ce type peut être **void** si la méthode ne retourne aucun résultat.



- une méthode peut avoir zéro, un ou plusieurs **arguments**, déclarés entre les parenthèses. Chaque argument à un nom (qui commence par une minuscule) et un type.
- Les méthodes qui retournent un résultat (celles qui déclarent un type de retour) utilisent l'instruction **return** pour retourner ce résultat.
- Lorsque qu'un objet veut donner accès en **lecture** à son état interne aux autres objets du système, il définit généralement des « **getter** ». Ce sont des méthodes comme **getX()** et **getY()** dans notre exemple, qui
 - o sont nommées **get<Nom de la Propriété>**
 - o n'ont pas d'argument en entrée
 - o le **type de retour** est le type de la propriété
 - o font simplement **return <propriété>**
- De façon similaire, lorsque qu'un objet veut donner accès en **écriture** à son état interne aux autres objets du système, il définit généralement des « **setter** ». Ce sont des méthodes comme **setX()** et **setY()** dans notre exemple, qui
 - o sont nommées **set<Nom de la Propriété>**
 - o ont un seul argument dont le type est le **type de la propriété**
 - o le **type de retour** est **void**
 - o font simplement **<propriété> = <argument>** ;

Les constructeurs

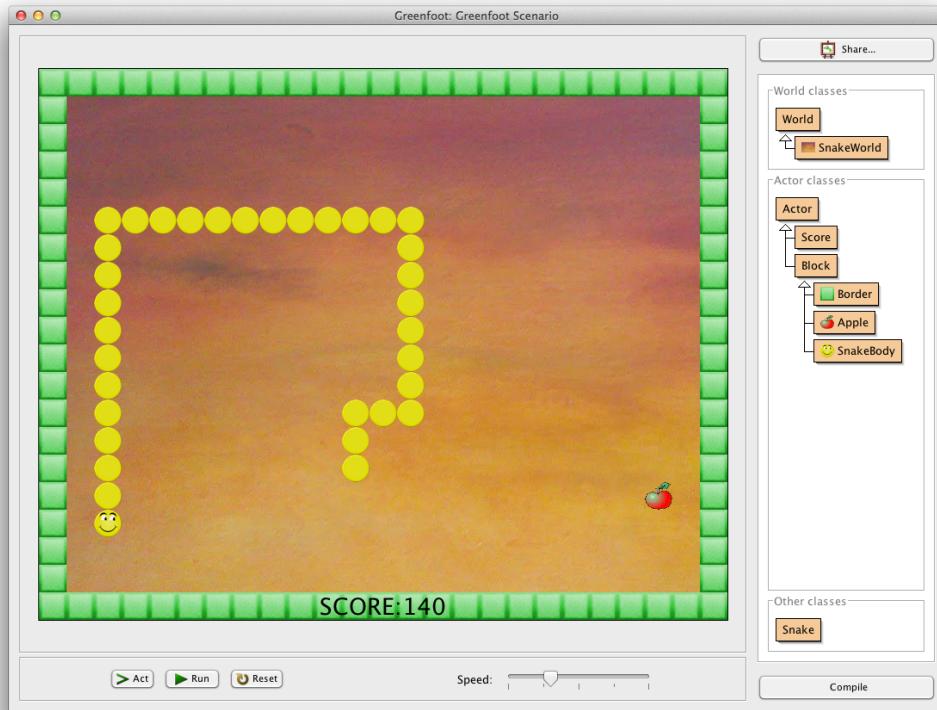
- Les constructeurs ressemblent à des méthodes, mais ils n'ont **pas de type de retour** et le **nom** d'un constructeur est toujours le même que **celui de la classe**.
- Un constructeur permet de créer un objet (on dit « instancier » la classe) avec une expression telle que **new GameElement(2, 3)** ;
 - o Dans ce cas, sur base de notre exemple, cela va créer un objet avec 2 et 3 comme valeur pour les propriétés x et y de notre objet.
- Il n'est pas nécessaire d'avoir un constructeur dans une classe. Dans ce cas, il existe un constructeur « par défaut » qui ne prend aucun argument et qui permet d'instancier la classe avec une expression telle que **new GameElement()** ;
- Notez que ce constructeur « par défaut » n'existe plus dès qu'un autre constructeur est défini dans la classe.

4. Présentation du jeu « Bobby Snake »

Une application dans Greenfoot s'appelle un scénario. On peut trouver plein de scénarios sur le site de Greenfoot <http://www.greenfoot.org/scenarios>

Dans ce workshop, nous allons créer un scénario de jeu que nous appelons « **Bobby Snake** ». Vous pouvez ouvrir le Scénario « **Bobby Snake** » en double-cliquant sur le fichier **project.greenfoot** dans le dossier **Bobby Snake**.

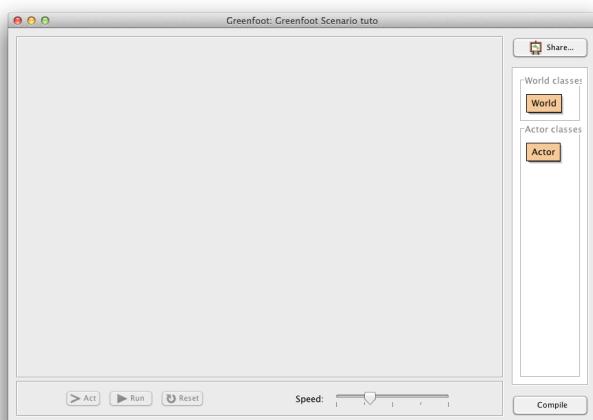




L'objectif de notre workshop va être de développer ce jeu de Snake (serpent). Bobby se déplace à l'aide des 4 flèches du clavier et s'agrandit à chaque fois qu'il mange une pomme. Attention, il ne peut pas aller sur les bords ou sur sa propre queue.

5. Démarrer un nouveau scénario

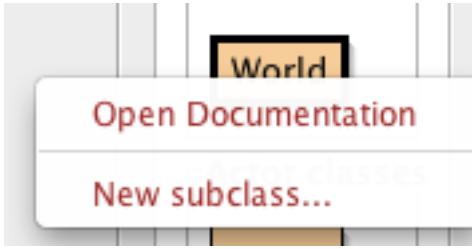
Nous allons donc commencer avec un scénario « vide » et nous allons expliquer comment développer Bobby Snake, étape par étape. Lorsque l'on démarre un scénario vide, l'écran de Greenfoot se présente comme suit :



Sur le côté droit de l'écran, GreenFoot nous montre deux classes importantes dans Greenfoot :

- **World** : c'est le « monde » de notre jeu. L'objet World va dessiner l'écran.
- **Actor** : ensuite, les différents éléments qui vont « vivre » dans le jeu (Bobby, les pommes, les bords) sont des « Actor » (des acteurs).

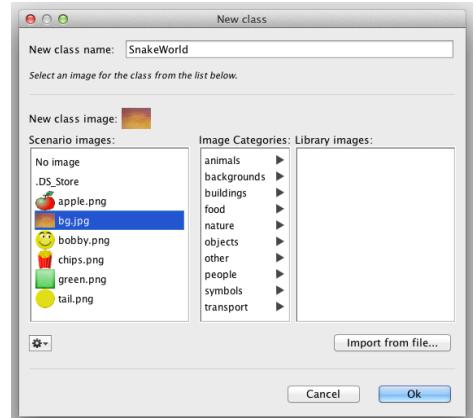
6. Création du World



En faisant un « clique-droit » sur World, nous allons choisir « New subclass... » (nouvelle sous-classe)

Dans la fenêtre de dialogue, nous allons définir le nom de notre classe : **SnakeWorld**

Nous pouvons également choisir une image de fond pour notre monde, prenez l'image **bg.jpg**



Après avoir cliqué sur OK, Greenfoot a créé une nouvelle classe, qui est une « sous-classe » de World.

En double-cliquant sur cette nouvelle classe, on peut ouvrir un « éditeur » de code Java.

```

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);
    }
}

```

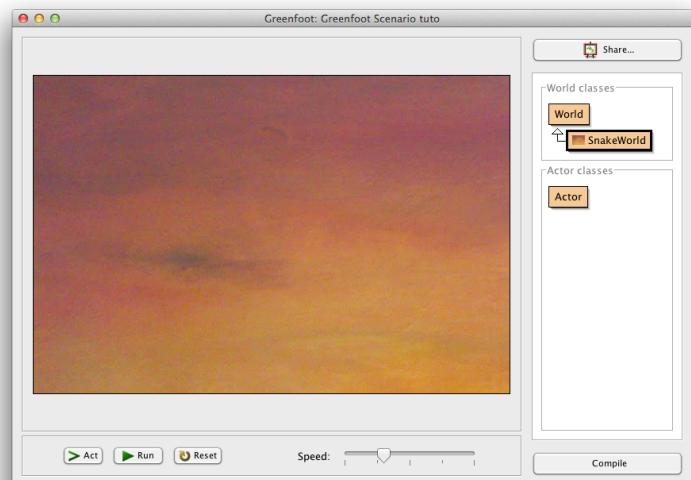
Nous reviendrons sur ce code dans un moment, mais nous allons d'abord l'exécuter.

Pour cela, il faut d'abord

« compiler » le code, en cliquant sur le bouton « compile », en bas à droite de la fenêtre Greenfoot.

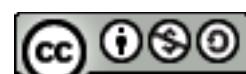
La **compilation** du code, va analyser le code, identifier les erreurs s'il y en a, et produire un « exécutable » qui permet alors d'exécuter le code.

Dans notre cas, l'exécution crée simplement une image.



Dans Greenfoot, les classes affichées en hachuré indiquent qu'elles nécessitent d'être (re)compilées. Si vous voulez exécuter le code, il faut alors d'abord faire une compilation en cliquant le bouton « **compile** »

Revenons sur le code :



```
public class SnakeWorld extends World
```

La partie « **extends World** » indique que c'est une classe qui « **hérite** » des caractéristiques (les propriétés, les méthodes) de la classe **World**. Cette classe est fournie par Greenfoot et permet de créer l'écran du jeu.

Ensuite, nous avons un « constructeur ». En effet, ce n'est pas une méthode, car il n'y a pas de type de retour et le nom est celui de la classe.

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

L'instruction « **super(600, 400, 1)** » permet en fait d'appeler un constructeur qui est défini sur la classe parent (**World**) et qui permet de « construire » le **World** en lui passant des paramètres pour définir les dimensions du jeu :

- Le premier paramètre correspond à la **largeur** de l'écran à créer
- Le second paramètre correspond à la **hauteur** de l'écran à créer
- Le troisième correspond à la taille d'un « bloc » si l'on veut avoir un écran composer d'un quadrillage

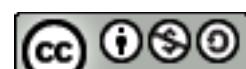
En faisant un « clique-droit » sur l'écran du jeu, on peut aussi « inspecter » l'objet qui a été créé (cliquez sur « **inspect** »). On voit que cet objet contient bien les paramètres qui ont été donnés pour le construire.



Changeons maintenant ce code, pour travailler sur base d'un quadrillage composé de blocs de 32 pixels, avec 25 blocs de large et 20 blocs de haut :

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

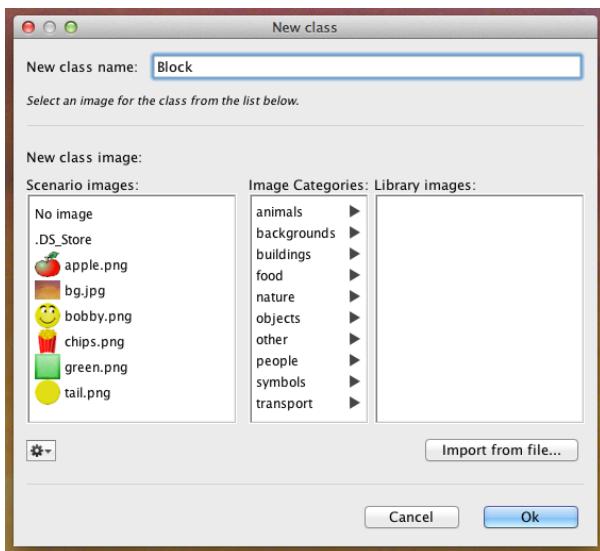
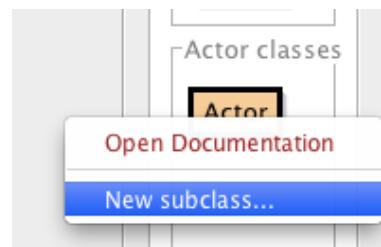
Compilez votre modification et vérifier en faisant un « **inspect** » du world.



7. Création d'un bloc

Nous allons maintenant commencer à créer les « Actors » (acteurs) qui vont faire partie de notre jeu.

Pour cela, faites un « clique-droit » sur la classe Actor et sélectionnez « New subclass... ».

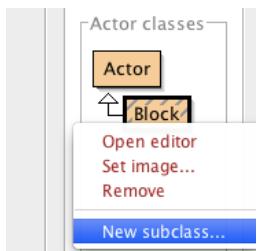


Nous allons créer plusieurs types de blocs qui seront utilisés dans notre jeu. Pour cela, nous allons d'abord créer une classe **Block**, qui sera une sous-classe de **Actor** (elle héritera donc des caractéristiques de la classe Actor de Greenfoot).

Pour l'instant, nous laissons cette classe vide : double-cliquez dessus pour voir le code et supprimez la méthode act().

```
public class Block extends Actor
{
}
```

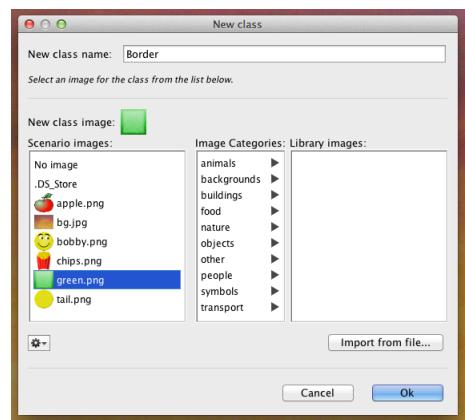
8. Création d'un bloc « border »



Créons maintenant une sous-classe de la classe « Block » que nous venons de créer (clique-droit sur Block, puis « New subclass... »).

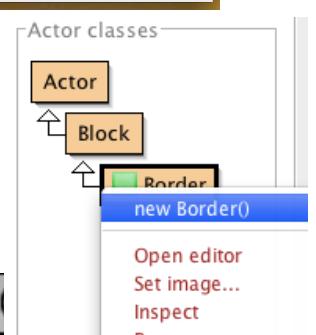
Appelez cette classe « **Border** » et choisissez l'image « green.png ».

Compilez.



Une nouvelle classe « **Border** » apparaît en dessous de Actor et Block.

En faisant un clique-droit, sur la classe Border, nous pouvons choisir l'option « new Border() » qui permet de créer un objet (ou une « instance ») à partir de notre classe.



A l'aide de la souris, vous pouvez positionner cet objet sur l'écran du jeu (dans une cellule de notre « quadrillage »).



Vous pouvez recommencez plusieurs fois, et créer ainsi plusieurs blocs. Vous pouvez ensuite utiliser l'option « inspect » pour inspecter l'état d'un objet.

Vous retrouverez la position (x,y) du bloc.

Vous verrez ainsi que la position du bloc dans le coin supérieur gauche est (0,0) et que la position **x** augmente vers la droite, alors que le **y** augmente vers le bas.

Les positions des blocs sont donc organisées comme dans le tableau suivant :

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

9. Affichage des bordures dans le World

Pour afficher une bordure dans notre World, cliquez sur la classe SnakeWorld et ajoutez le code suivant.

Compilez. Et vous devriez voir apparaître un bloc dans le coin supérieur gauche.

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

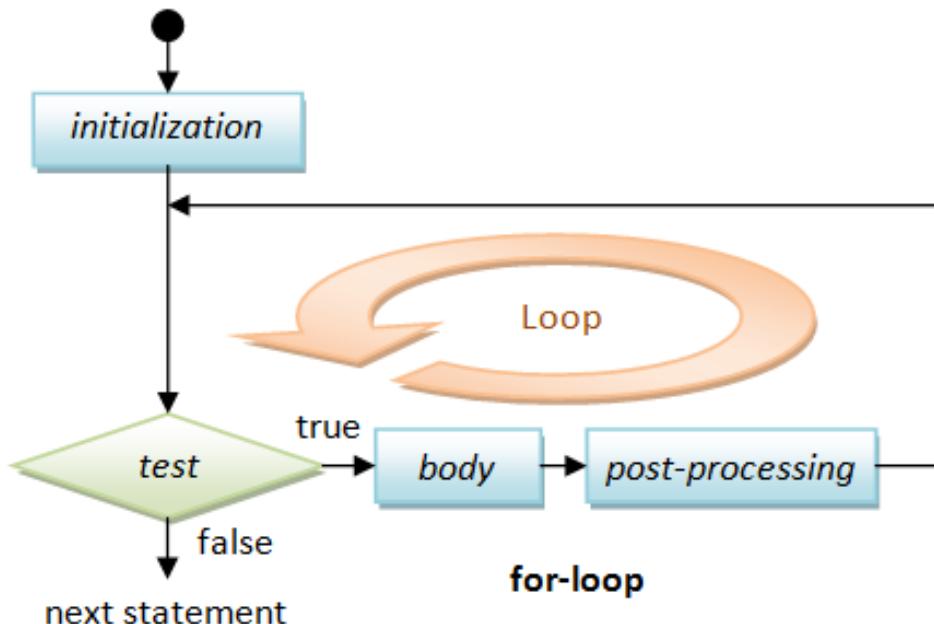
La méthode **addObject** est une méthode héritée de World qui permet d'afficher un Actor à une position x, y donnée. Le « **new** » devant le nom d'une classe permet de créer un objet (ou une instance) de cette classe. Cela correspond à l'appel d'un **constructeur** sur cette classe.

Pour afficher des blocs tout autour de notre écran, ce serait un peu long d'ajouter une commande pour chaque bloc. Pour cela, nous allons utiliser une commande très utile, les « boucles **for** » de Java.

La syntaxe d'une « boucle **for** » est la suivante :

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

Le schéma suivant illustre l'exécution de cette boucle :



Pour dessiner les bords de notre jeu, nous allons écrire une première boucle qui va définir une variable x, qui ira de 0 à la largeur du jeu. Dans cette boucle nous affichons le bloc du haut et celui du bas (notez que, comme l'index de la première ligne est 0, celui de la dernière ligne est le nombre de lignes -1).

Ensuite une deuxième boucle va définir une variable y qui ira de 0 à la hauteur du jeu et dans cette boucle, nous affichons les blocs de gauche et de droite.

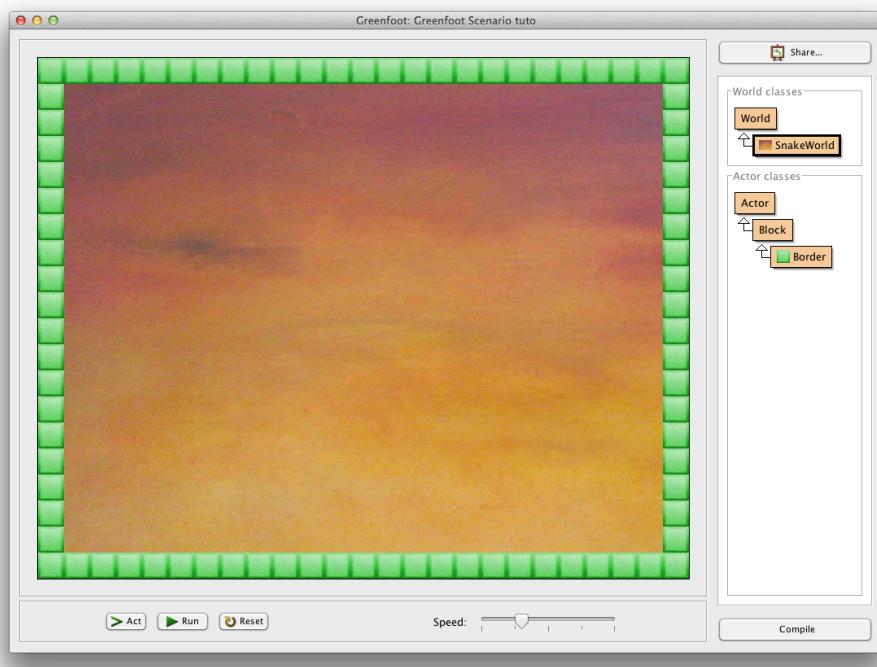
```
public SnakeWorld()  
{  
    super(25, 20, 32);  
  
    for (int x = 0; x < getWidth(); x++) {  
        addObject(new Border(), x, 0);  
        addObject(new Border(), x, getHeight() - 1);  
    }  
  
    for (int y = 0; y < getHeight(); y++) {  
        addObject(new Border(), 0, y);  
        addObject(new Border(), getWidth() - 1, y);  
    }  
}
```

Quelques explications supplémentaires :

- Les variables en Java sont toujours **typées**, ici le type de x et y est **int**. Nous allons beaucoup utiliser ce type qui représente un entier (0, 1, 2, ...)
- L'instruction **variable = <expression>** ; permet d'assigner la valeur de l'expression à la variable. Ainsi, l'instruction **x = 0;** permet d'initialiser la variable **x** avec la valeur **0**.
- Toutes les instructions en Java se terminent toujours par ;

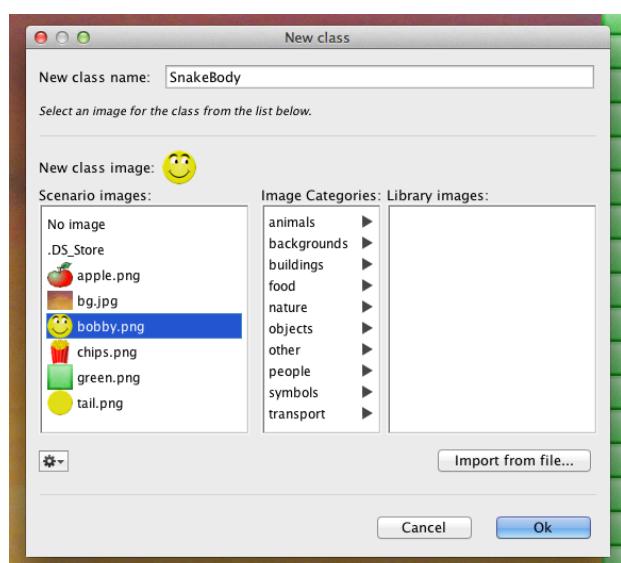
- L'instruction **x++** ; permet de rajouter 1 à la variable x. Il est équivalent à l'instruction **x = x + 1** ;
- Les méthodes **getWidth()** et **getHeight()** sont des méthodes héritées de la classe **World** et permettent de retourner la largeur et la hauteur du world.

Compilez et vous devez obtenir l'écran suivant.



10. Affichage de la tête de Bobby

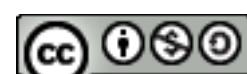
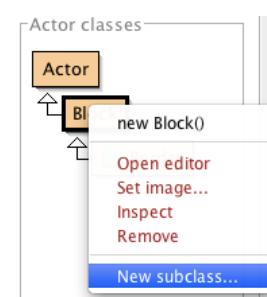
Créez une nouvelle sous-classe de notre classe Block.



Appelez-là **SnakeBody** et choisissez l'image de **Bobby, bobby.png**.

Le serpent dans notre jeu sera une « liste » de **SnakeBody**. Pour cela, notre **SnakeWorld** doit définir une propriété pour que cette liste de **SnakeBody** fasse partie de son « état ».

Ouvrez donc la classe **SnakeWorld** et ajoutez la définition suivante :



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

Cette définition crée une liste vide de **SnakeBody**.

Pour pouvoir utiliser la classe **LinkedList** de Java, vous devez également ajouter, au tout début de la classe **SnakeWorld**, l'instruction suivante :

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

Ensuite, ajoutons le premier élément dans cette liste, dans le constructeur de **SnakeWorld**.

La première instruction crée un nouvel objet **SnakeBody**.

La deuxième utilise la méthode **add()** sur la liste **snake**, qui permet d'ajouter un nouvel élément (passé en paramètre) à la fin de la liste.

Et la troisième affiche le nouvel élément dans le world en position (2,2), grâce à l'utilisation de la méthode **addObject()** que nous avons déjà utilisée pour afficher les bordures.

Compilez, vous devez maintenant voir la tête de Bobby en position (2, 2).



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

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

11. Déplacement de Bobby

Pour savoir dans quelle direction se déplace Bobby, nous allons ajouter deux propriétés supplémentaires, une pour la direction selon l'axe des x et une pour la direction selon l'axe des y.

```

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

```

Pour que le mouvement initial de Bobby soit vers la droite, dx est initialisé à 1 et dy à 0.

Maintenant, ajoutons une méthode **act()** à notre **SnakeWorld**. Cette méthode est appelée par Greenfoot de façon continue pour faire avancer le jeu, sur le **World** et sur tous les **Actors** qui ont été ajoutés au **World**.

Voici une première version de cette méthode

```

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);
}

```

Dans cette première version, nous prenons la tête actuelle (dernier élément dans notre liste). La méthode **getLast()** sur une **LinkedList** permet en effet de retourner le dernier élément dans la liste (last = dernier en anglais).

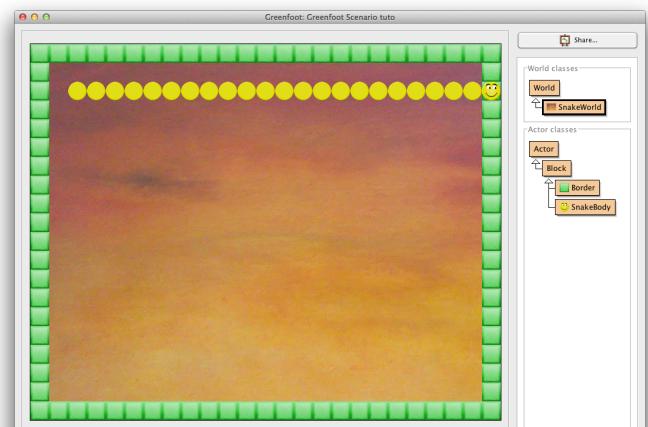
Ensuite, nous changeons l'image de cette tête pour qu'elle devienne l'image d'un élément de la queue (tail = queue en anglais).

Ensuite, nous créons une nouvelle tête avec l'instruction **new SnakeBody()**, et nous calculons sa nouvelle position sur base de la position de la tête courante et de la direction (dx et dy).

Puis, nous ajoutons la nouvelle tête à la nouvelle position dans le world et dans notre liste.

Compilez et testez le programme en cliquant sur « **run** » (pour demander à Greenfoot d'appeler la méthode **act()** de façon continue).

Le serpent avance bien vers la droite, mais il s'agrandit à chaque fois, jusqu'à aller contre le bord.



12. Limiter la taille de la queue

Afin de limiter la taille de la queue, nous allons ajouter une propriété au world, qui indique combien d'éléments de la queue doivent encore être affichés.

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

Dans la méthode **act()**, après avoir affiché la nouvelle tête, on va tester la valeur de ce compteur :

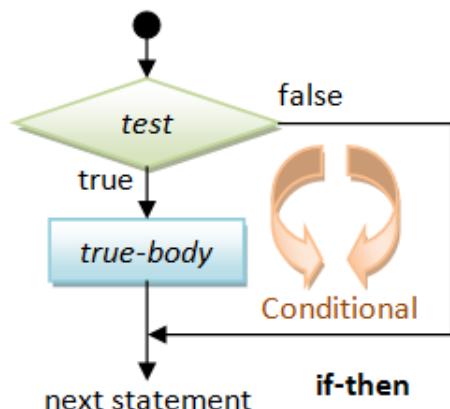
- s'il est à zéro, on efface le dernier élément de la queue (qui est en fait le premier élément dans la liste, puisque l'on ajoute les nouveaux éléments en fin de liste).
- sinon, on diminue le compteur de 1

Pour cela, nous allons utiliser une autre instruction très importante en Java : le **if** (qui veut dire « si » en anglais).

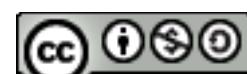
La syntaxe d'un **if** est la suivante

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

Et son exécution dépend du résultat du test sur la **condition** :



Une autre forme du if, est le **if-then-else** qui permet de spécifier deux comportements :

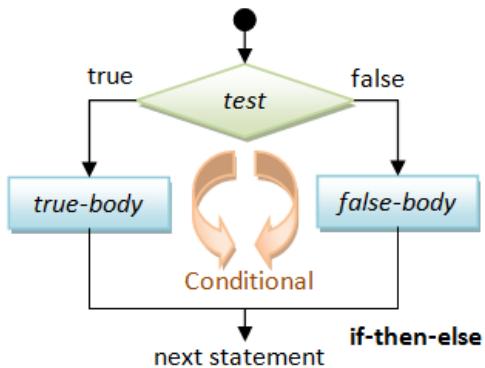


```

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

```

Son exécution dépend du résultat du test sur la **condition** :



Voici le code à ajouter dans la méthode **act()**.

```

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);
    } else {
        tailCounter--;
    }
}

```

La méthode **removeFirst()**, sur une **LinkedList**, supprime le premier élément de la liste et le retourne.
La méthode **removeObject()**, héritée de World, supprime l'actor passé en paramètre du world (et donc l'efface à l'écran).

tailCounter-- ; enlève 1 à la variable **tailCounter**, cette instruction est équivalente à **tailCounter = tailCounter -1** ;

Lors des 5 premiers appels de la méthode **act()**, le **tailCounter** n'est pas égal à 0, le serpent s'agrandit d'un élément et la **tailCounter** diminue de 1.

Dès que le **tailCounter** arrive à 0, le serpent ne grandit plus, puisque on ajoute une tête et on enlève un élément de la queue.



13. Changement de direction

Maintenant, nous voudrions gérer les changements de direction en fonction des touches appuyées sur le clavier.

Pour cela, nous allons créer une nouvelle méthode dans **SnakeWorld**.

```
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;
    }
}
```

L'instruction
Greenfoot.isKeyDown(« key »)
permet de tester si une touche du clavier est enfoncée.

Pour changer la direction vers la gauche ou la droite, il faut cependant aussi tester que la direction actuelle n'est pas déjà la gauche ou la droite car Bobby ne peut pas faire demi-tour sur lui-même.

On utilise alors une expression comme

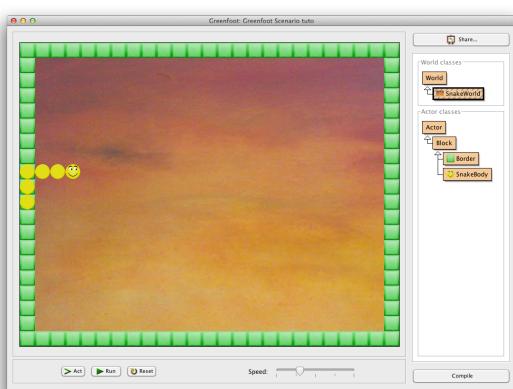
Greenfoot.isKeyDown("left") && dx == 0

Qui se traduit par « la touche gauche est enfoncée **et** dx est égal à 0 ». Le signe **&&** correspond donc au **et**, et le signe **==** permet de comparer deux valeurs.

Il ne reste plus qu'à exécuter cette méthode au début de notre méthode **act()**

Compilez et testez.

```
public void act()
{
    changeDirection();
    // un nombre limité de 1
```



Vous pouvez maintenant diriger Bobby, notre serpent, grâce aux flèches du clavier.

Cependant, remarquez que Bobby peut aller où il veut, y compris sur les bords, et peut même se croiser sur lui même.

14. Gérer les collisions

Avant de gérer les collisions, nous allons ajouter une propriété **dead** (dead = mort en anglais) à notre **SnakeWorld**. Cela nous permettra d'arrêter le jeu lorsque Bobby rencontre un obstacle.

```
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;
```

Cette variable est du type « **boolean** ». C'est un type important en java qui ne peut prendre que 2 valeurs :

- **true** : vrai
- **false** : faux

Nous pouvons ensuite modifier la méthode **act()** pour qu'elle ne fasse plus rien, lorsque Bobby est mort.

Si **dead** vaut **true**, la méthode retourne directement, et le jeu s'arrête.

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

Ajoutons également une méthode **dead()** sur notre world, pour permettre de changer la valeur de la propriété **dead** :

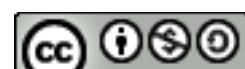
```
public void dead() {
    dead = true;
}
```

Occupons-nous maintenant des « collisions ». Dans la méthode **act()**, *juste avant d'afficher la nouvelle tête*, nous allons voir s'il y a des blocs à la position de la nouvelle tête et exécuter une méthode sur ce(s) bloc(s).

```
//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);
```



La méthode **getObjectsAt()**, héritée de la classe **World**, retourne la liste des blocs à la position **x, y** passée en paramètre et du type passé en troisième paramètre (pour nous, **Block.class**).

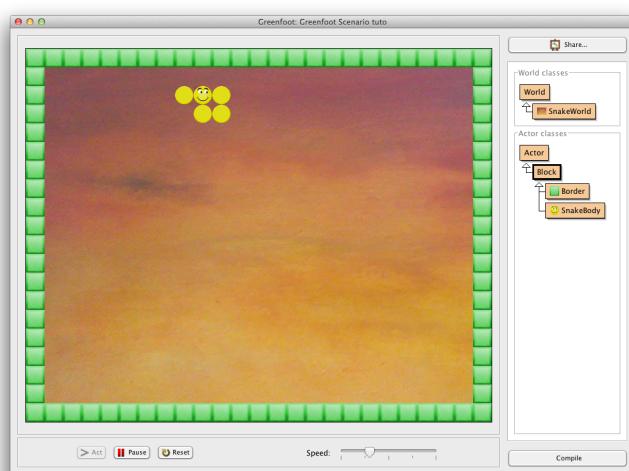
Cette méthode retourne une liste de **Block**. L'instruction **for** qui suit, permet de « boucler » sur chaque bloc de la liste et d'exécuter la méthode **collision()** sur ceux-ci. Le « **this** » passé en paramètre permet de passer l'objet courant (dans notre cas, le **SnakeWorld**) à la méthode appelée.

A ce stade, si vous compilez, il y a une erreur, car la méthode **collision()** n'existe pas encore sur **Block**.

Ouvrez, la classe **Block** et ajoutez une méthode **collision** :

La méthode, appelle simplement **dead()** sur le world passé en paramètre, ce qui va arrêter le jeu.

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

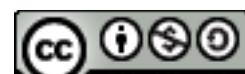
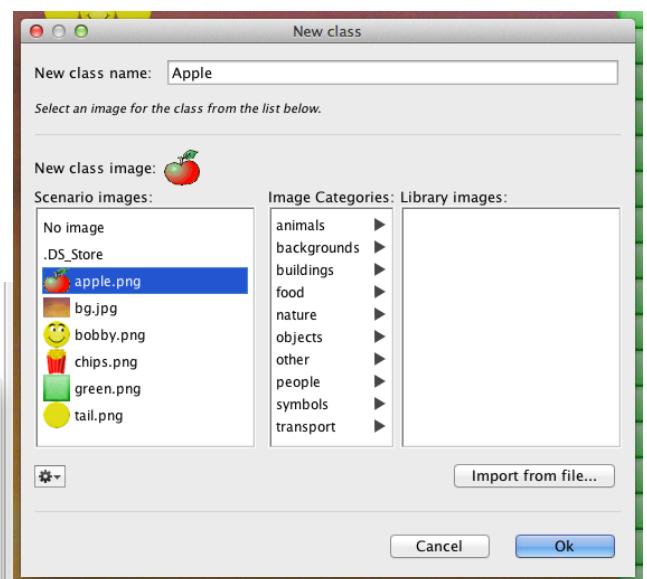
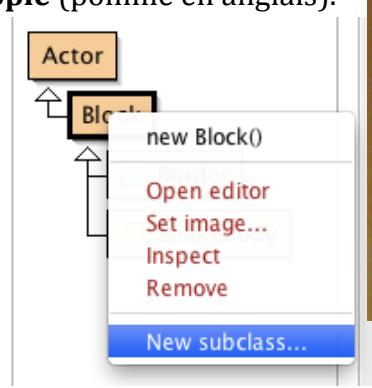


Compilez et testez.

Le jeu s'arrête dès que vous allez sur un des bords ou si vous rentrez en collision avec la queue du serpent.

15. Ajout d'une pomme

Ajoutons maintenant une nouvelle sous-classe de **Block** que l'on appelle **Apple** (pomme en anglais). Choisissez l'image de **apple.png**.



Ouvrez la nouvelle classe **Apple**, et supprimez la méthode **act()**.

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

Revenons dans la classe **SnakeWorld**. Lors de l'initialisation du jeu (dans le constructeur), après avoir créé Bobby, nous allons également créer une pomme et la positionner dans le jeu **aléatoirement**.

```
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.getRandomNumbergetHeight()-2)+1);
```

La pomme est créée avec l'instruction **new Apple()**. Ensuite, elle est ajoutée au World avec la méthode **addObject()**. Sa position x, y est obtenue en utilisant une méthode **getRandomNumber()** de Greenfoot. Cette méthode prend en argument un nombre et retourne un nombre aléatoire (au hasard) entre 0 et le nombre passé en argument. Les nombres passés sont la largeur et la hauteur moins 2 (pour ne pas compter les bords) et on ajoute 1 (pour ne pas être sur le bord).

Compilez et testez. Essayez d'attrapez la pomme... que se passe-t-il ?

Le jeu s'arrête ! En effet, nous avons programmé notre jeu pour qu'il s'arrête dès que le serpent entre en collision avec un bloc, et la pomme est un bloc !

16. Collision avec une pomme

Nous allons alors « **surcharger** » la méthode collision dans la classe **Apple**.

En Java, on peut en effet **surcharger** une méthode héritée pour définir un comportement différent dans une sous-classe.

```
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);
    }
}
```



Lorsque le serpent rentre en collision avec une pomme, il ne faut plus appeler la méthode **dead()**, mais plutôt demander au world d'agrandir (grow en anglais) le serpent (par exemple de 2 éléments).

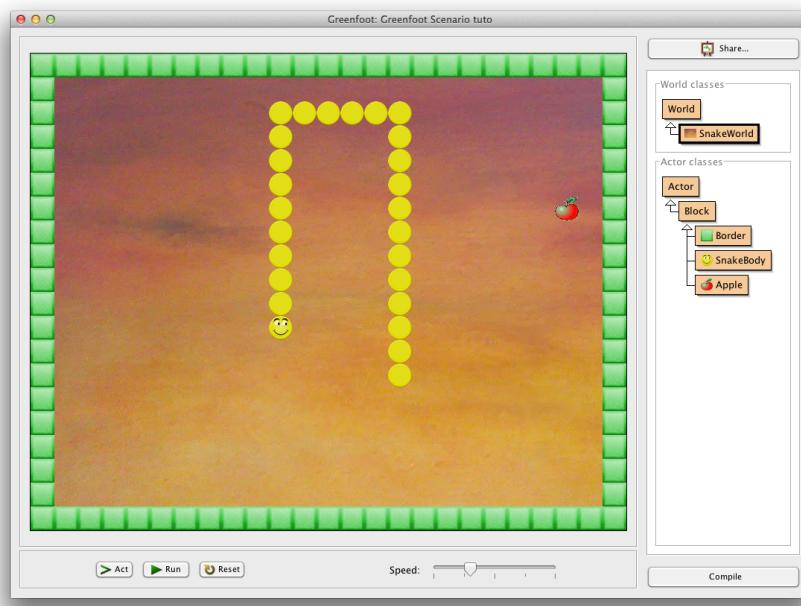
En plus, il faut repositionner la pomme à une autre position.

Reste à ajouter la méthode **grow()** dans la classe **SnakeWorld** :

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

Cette méthode va juste ajouter le nombre d'élément à notre compteur **tailCounter**.

Compilez et testez. Le serpent s'agrandit maintenant à chaque pomme mangée !



17. Ajouter du son

Pour améliorer notre jeu, nous allons maintenant ajouter du son, lorsque le serpent mange une pomme, ou lorsqu'il meurt.

Pour ajouter du son, lorsqu'une pomme est mangée, ajoutez simplement le code suivant dans **Apple** :

```
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);  
}
```

Compilez et testez !

La méthode **playSound()** prend en paramètre le nom d'un fichier de son qui doit se trouver dans le dossier **sounds** de votre projet Greenfoot.

Ajoutez également un autre son, lorsque Bobby meurt, en ajoutant le code suivant dans la classe **Block** :

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

Compilez et testez !

18. Fin

Félicitation ! Vous avez terminé votre premier jeu Greenfoot !

Vous pouvez maintenant continuer à apporter de nouvelles fonctionnalités dans votre jeu, ou en créer d'autres !

