1. Greenfoot

Greenfoot (http://www.greenfoot.org/) is a **development environment** based on the **Java programming language**.

It is mainly designed to **ease learning the Java language** and to easily create **2D games**.

Greenfoot is a free software and is available for most computers as it supports Windows, Mac OS X and Linux.

2. Java

The Java language is an "object oriented" programming language created around 1996.

Today, it is one of the most popular and used languages in the world. It is used to develop many different types of applications (games, banking application, robots, Android smartphones, …)

It is a language that allows to write applications that can run on different "operating systems" (Windows, Mac, Linux, …).

You can find many web site and books about Java. Here follows a few links:

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

http://www.bluej.org/

3. Classes and objects

Before starting with Greenfoot, we begin with a few words about **classes** and **objects** in Java.

Java is an "**object oriented**" programming language. This means that an application is made of "**objects**".

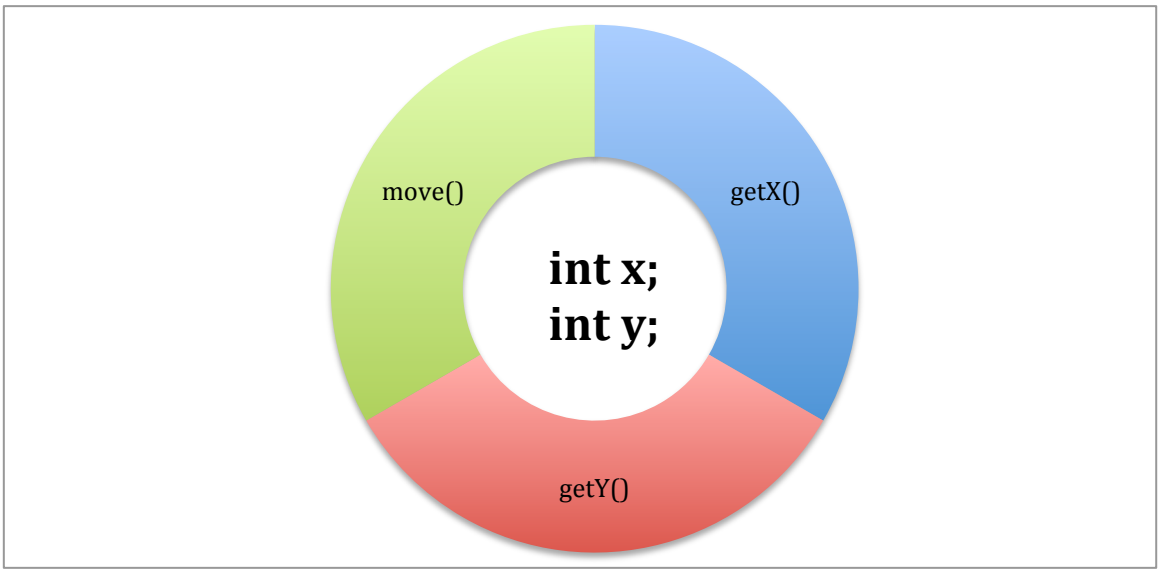
Let's take a concrete example: a game character.

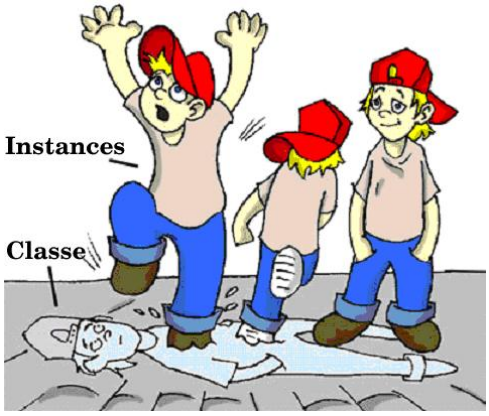
Each object is made of two important parts:

- its « state » composed of a set of **properties**
- its « behaviour », that is the list of actions it can do

In our example of a game character:

- its state is composed of its position in the game, its movement direction, …
- its behavior is for example the set of movements it can do.





To describe a Java object, we will define a **class**. A class is a kind of "model" used to construct the objects (also called instances of the class).

We will see these concepts and the basics of the Java language with code samples, within Greenfoot, during this workshop. But, let's start with a small sample to illustrate the basics of Java. You can also come back to this part later on, to better understand the Java basic concepts, as we go along with the workshop.

```java
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;
	}
}
```

The class

- is declared as « **public** » to allow all other object withing our application to interact with the objects (also called instances) of this class.
- has a **name**, that, by convention, always starts with an **Uppercase**. There is never spaces in Java names. If the name is composed of several words, they are attached together and every word will start with an uppercase to make it more readable.

The object **state**

- is usually declared at the beginning of the class,
- each property is usually declared « **private** », which means that only the object can have access to it. The other objects of the system cannot access to the object « internal state ».
- each property has a **name** that, by convention, starts with a lowercase character.
- each property has a **type**. Here **int** represent the « integer number » type (0, 1, 2, …). We can also use any other class as the property type.

The object **methods**

- Methods are often declared « **public** » so they can be called from the other objects that are in the application. They can also be « **private** » if they only need to be visible from the object itself (and not from the other objects).
- Each method has a **name** that should always starts with a lowercase character, by convention.

- Each method has a **return type**. This is the type of the result returned by the method. If the method does not return any result, the return type is defined as **void**.
- A method can have zero, one or several input **arguments**, declared between brackets, just after the method name. Each argument has a name (which also starts with a lowercase character by convention) and a type.
- Methods that return a result (the ones that define a return type) must use the **return** statement to return the result.
- As the object state is usually private, when an object wants to give **read** access to its internal state (or at least part of it), it usually defines so-called « **getter** » methods. These are methods such as **getX()** and **getY()** in our example
    - they are named **get<Property Name>**
    - they do not have any input argument
    - their **return type** is the property type
    - they simply do a **return <propriété>**
- Similarly, when an object wants to give **write** access to its internal state (or at least part of it), it usually defines so-called « **setter** » methods. These are methods such as **setX()** and **setY()** in our example
    - they are named **set<Property Name>**
    - they have a single input argument, typed with the **property type**
    - their **return type** is **void**
    - they simply do **<property> = <argument> ;**
    -

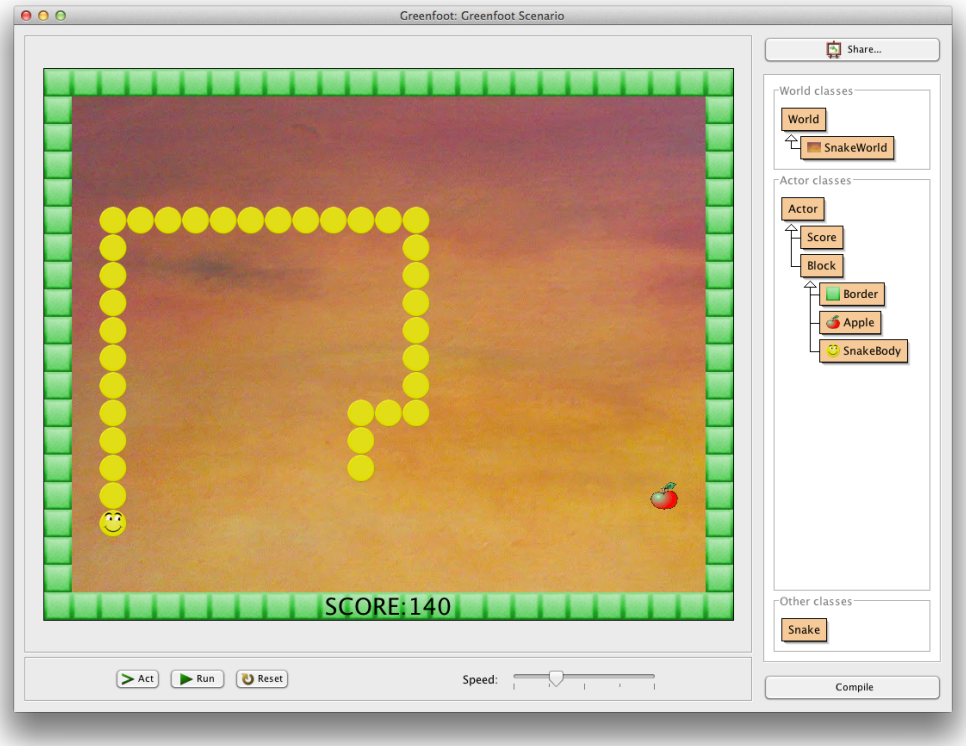
Constructors

- Constructors are similar to methods, except that **they do not have any return type** and their name is always the same as the **class name**.
- A constructor allows to create an object (we also say an "instance") with an expression such as **new GameElement(2, 3) ;**
    - In this case, based on our example, this will create an object (or instance) with 2 and 3 as the values for the properties x and y.
- It is not always needed to have a constructor in a class. In this case, their exists a constructor "by default" that has no input argument and that allow to instantiate (create instances) the class with an expression such as **new GameElement() ;**
- Note that this constructor « by default » will not exist any more as soon as another constructor is explicitly defined in the class.

4. Presentation of the « Bobby Snake » game

An application, within Geenfoot, is called a scenario. You can find many scenarios on the Greenfoot web site : http://www.greenfoot.org/scenarios

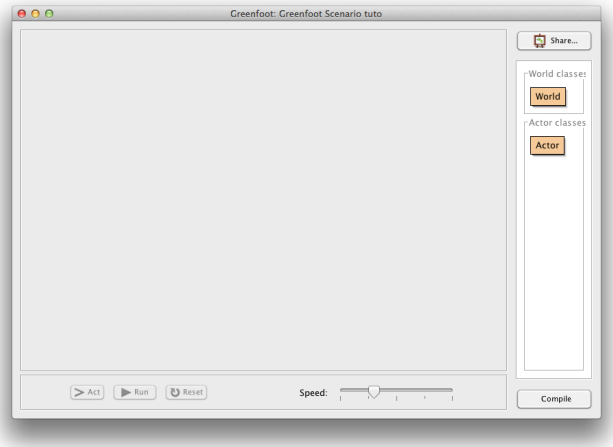
In this workshop, we will create a game scenario that we call « Bobby Snake ». You can open the « Bobby Snake » scenario by double-clicking the file **project.greenfoot** in the **Bobby Snake** folder.



The workshop objective will be to develop this Snake game. You can move Bobby, the snake, with the four arrow keys of the keyboard. Each time Bobby eats an apple, it will grow. Attention, he cannot go on the game borders nor can he eat himself !

5. Start a new scenario

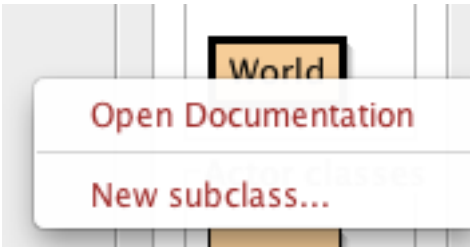
We will start with an empty scenario et will explain all the steps to create the Bobby Snake game. When starting a new scenario, Greenfoot screen is as follows:



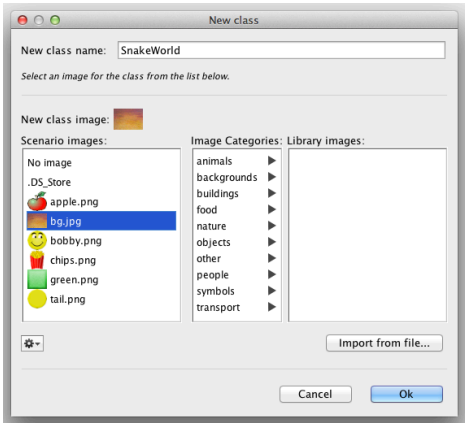
On the right hand side of the screen, GreenFoot shows two important classes:

- **World** : this is the game « world ». The World object will draw the screen.
- **Actor** : the different game elements that will live in the game will be the **Actors** (Bobby, apples, border blocks).

6. Creation of the World

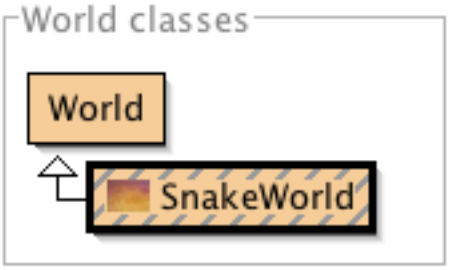


Do a « right-click » on World, and choose « New subclass… »



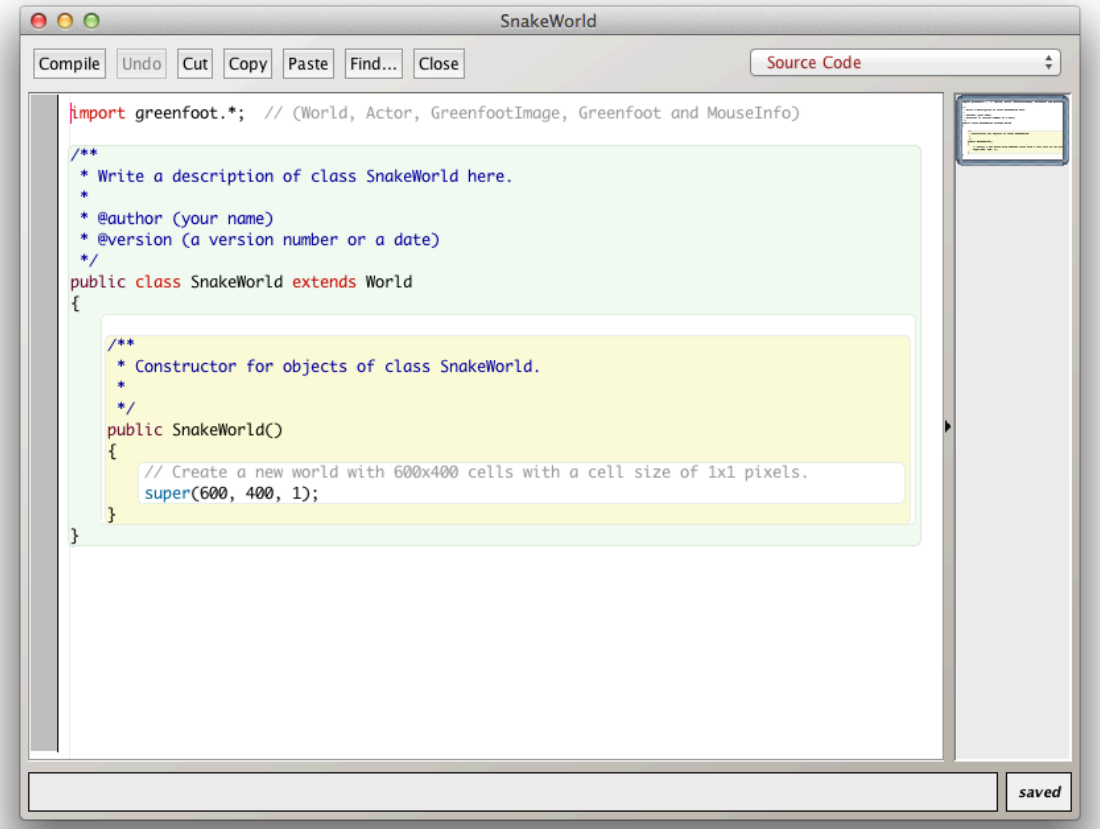
In the dialog window, let's define our class name : **SnakeWorld**

We can also choose a background image to our World. Let's take the image named **bg.jpg**

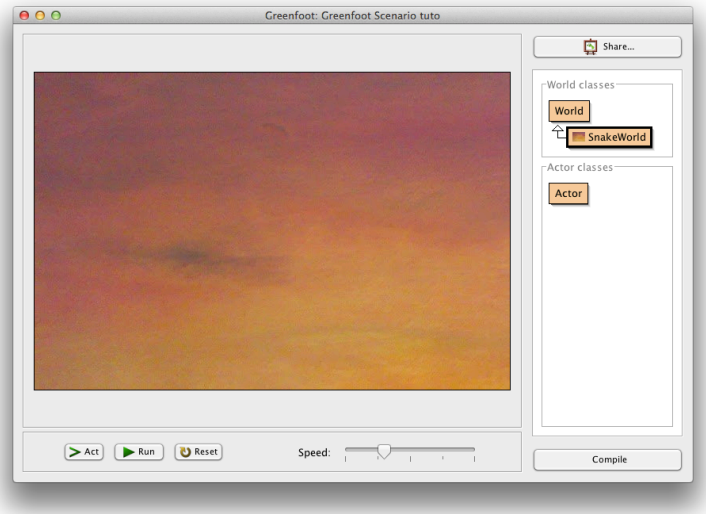


After having clicked OK, Greenfoot creates a new class, which is a « sub-class » of World.

When double-clicking on this new class, Greenfoot opens a « Java code editor ».



We will come back on this code in a moment, but for now, we will first execute it. For that, we must first « compile » the code, by clicking on the « compile » button (down at the right of the Greenfoot main window).



The code **compilation** will analyse the code, identify potential errors, and produce an « executable » allowing to execute it.

In our example, the execution simply displays an image.

Within Greenfoot, classes displayed as **hatched** indicates classes that need to be (re)compiled. If you want to execute your code, you first need to do a compilation by clicking the « **compile** » button.

Let's come back to our code :

```
public class SnakeWorld extends World
```

The part « extends World » indicates that our SnakeWorld class « **inherits**» the characteristics (properties, methods) of the class **World**. This class is provided by Greenfoot and allows to create the main screen of the game.

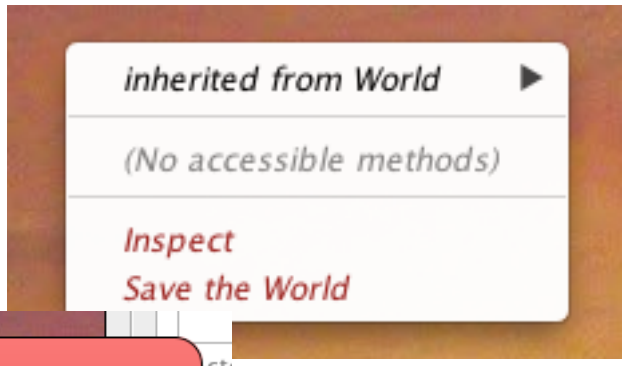
Next, we have a « **constructor** ». Indeed, this is not a method because it does not have any return type and its name is the same as the class name.

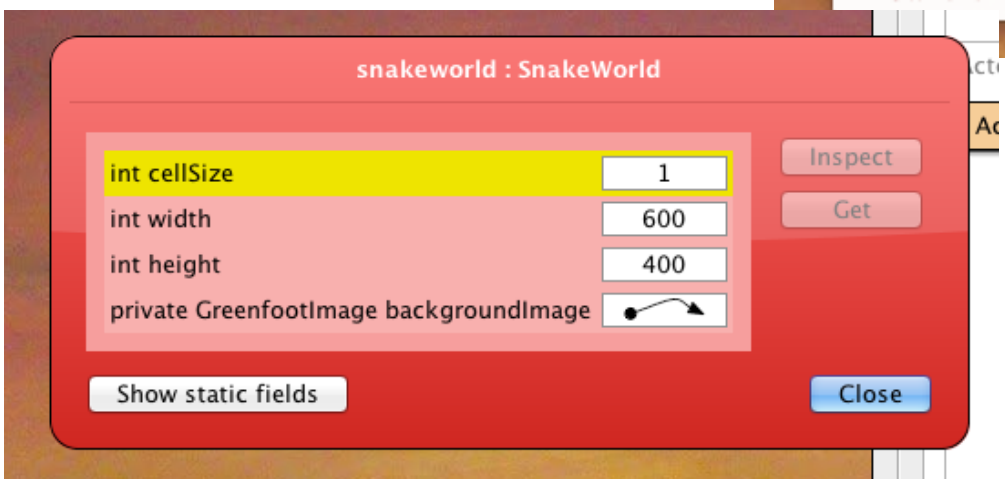
```
public SnakeWorld() {
    super(600, 400, 1);
}
```

The statement « **super(600, 400, 1)** » actually calls a constructor that is defined on the parent (World) and that allow to « create » the World object by giving it the parameters defining the size of the game screen :

- The first parameter corresponds to the **width** of the screen to create
- The second parameter corresponds to the **height** of the screen to create
- The third parameter corresponds to the size of a « block », if we want to have a screen composed of blocks.

By doing a « right-click » on the game screen, we can also « inspect » the created object (click on « inspect »). We see that the created object is well defined with parameters provided to the constructor.





Let's change this code now, to work on a screen composed of blocks of 32x32 pixels, with 25 blocks width and 20 blocks height:

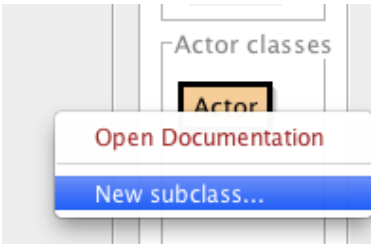
```
public SnakeWorld() {
    super(25, 20, 32);
}
```

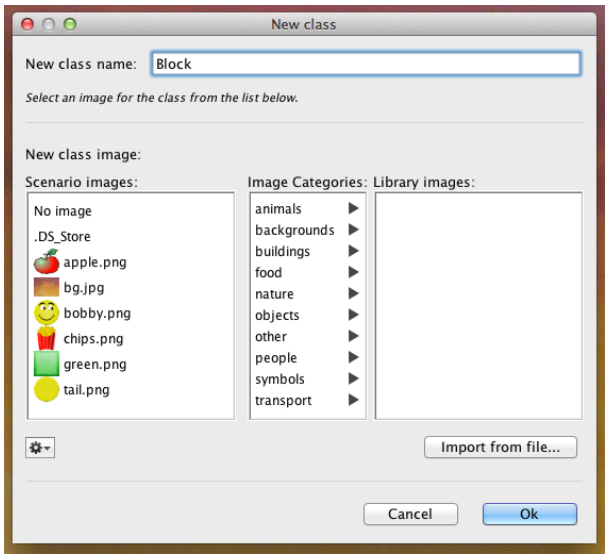
Compile your changes and check it by doing an « inspect » of the world.

7. Creation of a Block

We will start now creating the « Actors » that will be part of our game.

For that, do a « right-click » on the Actor class and select « New subclass… ».



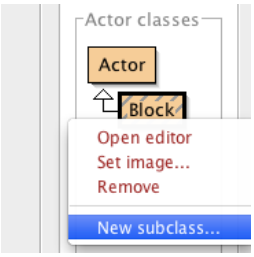


We will create several types of blocks that will be used in our game. For that we will first create a class named **Block**, that will be a sub-class of **Actor** (it will hence inherit the characteristics of the Greenfoot Actor class).

For now, we can just keep this class empty: double-click on it to see the code created by Greenfoot and suppress the act() method (we don't need it).

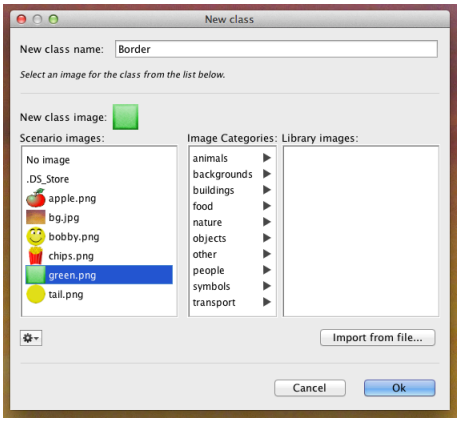
```
public class Block extends Actor
{

}
```

8. Creation of a « border » block



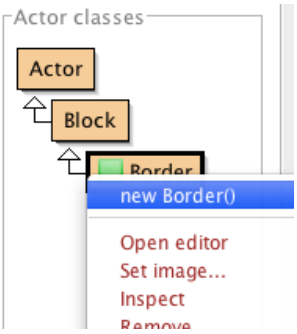
Let's create now a sub-class of our « Block» class we have just created (right-click on Block, then « New subclass… »).

Name the new class « **Border**» and choose the image « green.png ».

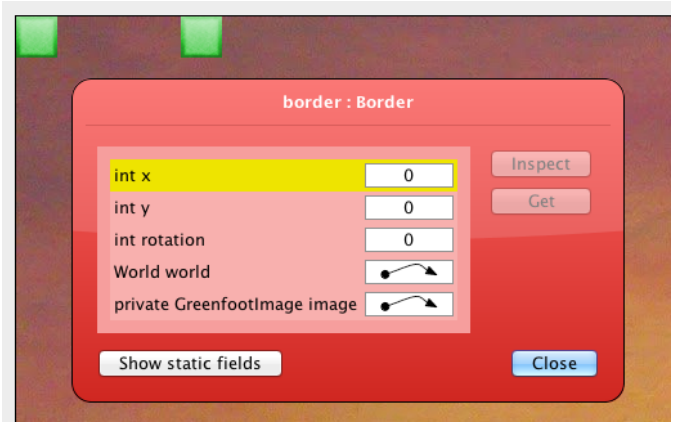


A new class « Boder» appears en below the Actor and Block classes.

By doing a right-click on the Border lass, we can choose the « new Border() » option allowing to create an objet (or « instance ») from our class.



With the help of the mouse, you can position this object anywhere on the game screen (in one of the cells of the screen).



You can do this any time you want to create multiple blocks. And then you can use the « inspect » to inspect the state of each of them.

In their state, you will see the block position (x,y).

You can see that the position of block placed at the top left of the screen is (0,0) and that the **x position** grows when going to the right, and the **y position** grows when going down.

Block positions are hence organized as shown in the following table:

(0,0)	(1,0)	(2,0)	…
(0,1)	(1,1)	(2,1)	…
(0,2)	(1,2)	(2,2)	…
…	…	…	…

9. Display the World's border

To display a border in our world, click on the SnakeWorld class to open it and add the following code.

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

Compile it. You should see a block appearing at the top left corner.

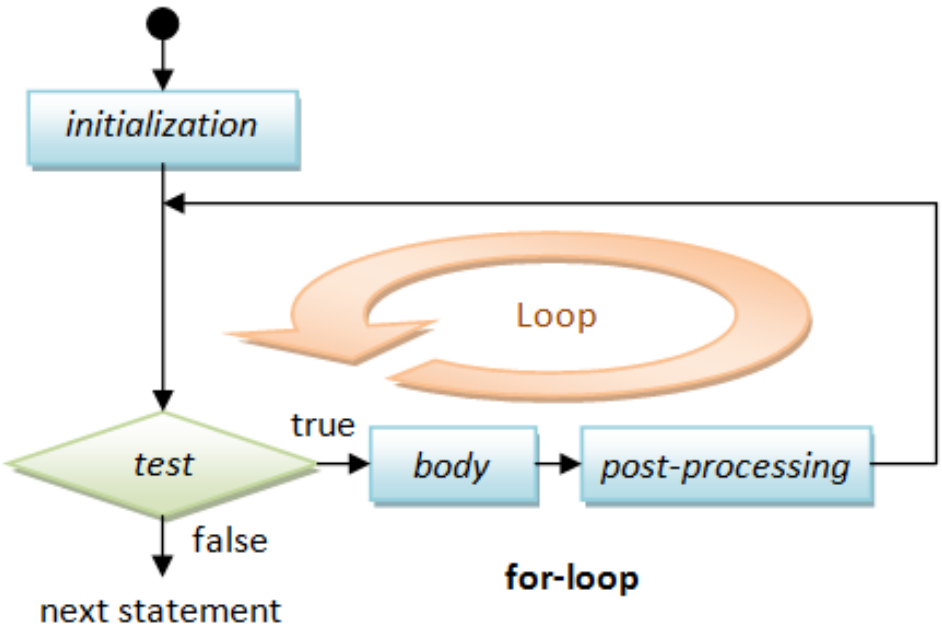
The **addObject** method is a method inherited from the World class and it allows to display an Actor at a given (x, y) position, passed as arguments. The « **new** » just before the name of a class allow to create a new Object (or instance) of that class. This corresponds to a call of a constructor on that class.

To display blocks all around our screen, it would be a little tedious to write down a line of code for each block. For that, we will use a very useful command, the « **for** loop » of Java.

The syntax of a « **for** loop » is the following :

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

The following schema illustrates the execution of that loop:



for-loop

To display the borders of our game, we will write a first loop that defines a variable x, from 0 to the width of the game. In this loop, we display the blocks on the top and the bottom (note that, as we the first line index is 0, the last line index is the number of lines – 1).

Next, a second loop will define a variable y, from 0 to the game height, and in this loop, we will display the blocks on the left and on the right.

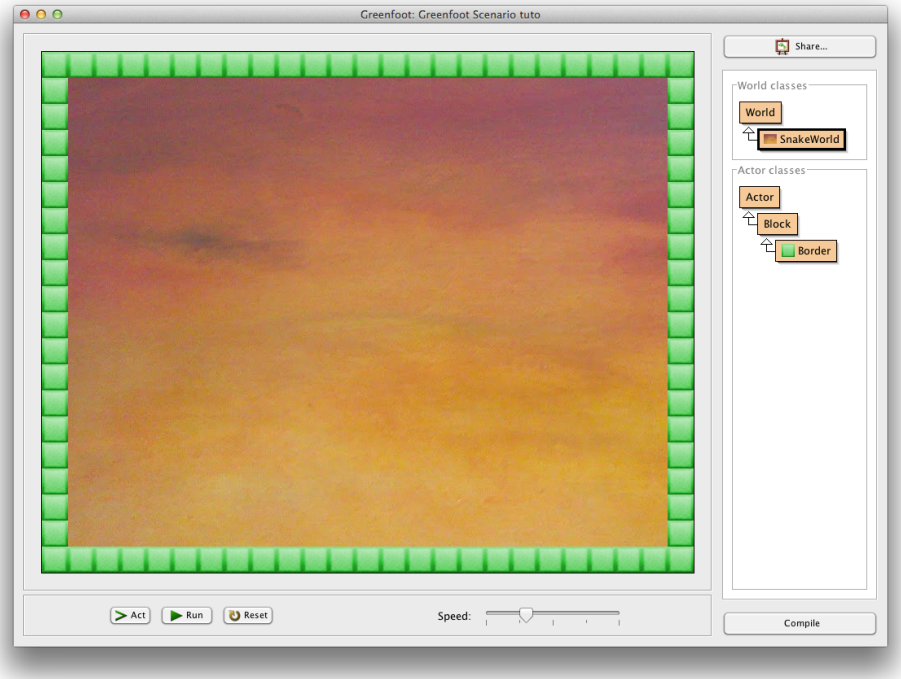
```
public SnakeWorld()
{
    super(25, 20, 32);

    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);
    }

}
```

A few additional explanations here:

- Variables in Java are always **typed**, here the type of the variable x and y is **int**. We will often use this type which represents an integer number (0, 1, 2, …)
- The statement **variable = <expression> ;** allow to assign the value of the expression to the variable. So, the statement **x = 0;** allow to initialize the variable **x** with the value **0**.
- All statements in Java always end with **;**
- The statement **x++ ;** allow to add 1 to the variable x. it is equivalent to the instruction **x = x + 1 ;**

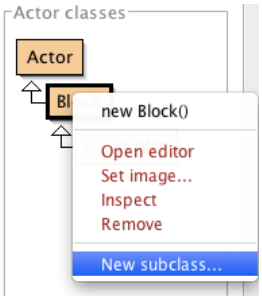
- The methods **getWidth()** and **getHeight()** are inherited from the **World** class and allow to get the width and height of the world.

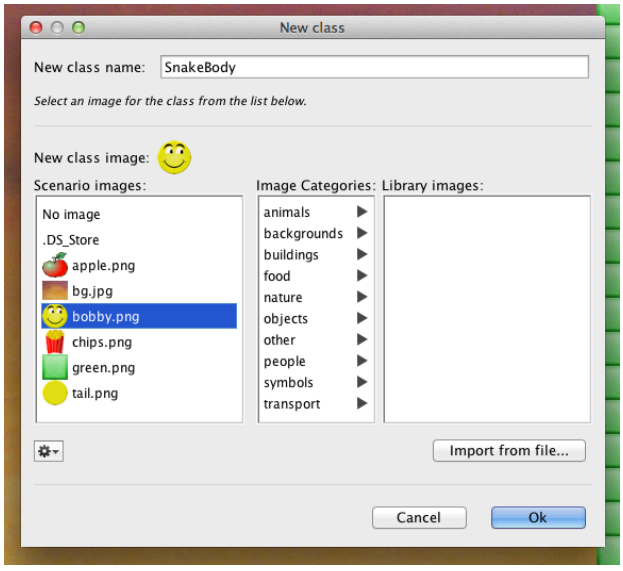
Compile your code, you should get the following screen.



10. Display Bobby's head

Create a new sub-class of our Block class.





Call it **SnakeBody** et choose the image of Bobby, **bobby.png**.

The snake in our game will be a « list » of SnakeBody. For that, the **SnakeWorld** will define a property so the list of SnakeBody will be part of its « state ».

Open the **SnakeWorld** class and add the following definition:

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

This definition creates an empty list of **SnakeBody**.

To be able to use the **LinkedList** class of Java, you also need to add at the beginning of the **SnakeWorld** class, the following instruction:

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

Next, let's add the first element in this list, in the SnakeWorld constructor.

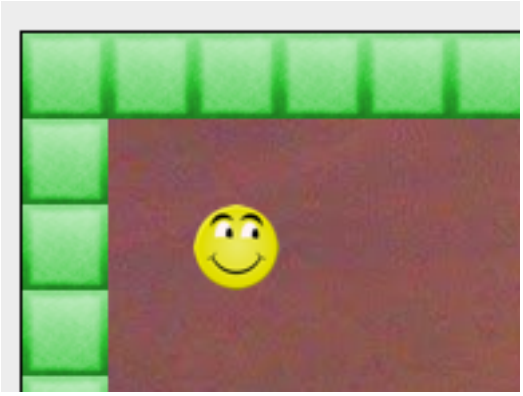
```
public SnakeWorld()
{
    super(25, 20, 32);

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

The first statement creates a new **SnakeBody** object.

The second one uses the **add()** method on the **snake** list, allowing to add a new element (passed as input parameter) at the end of the list.

And the third statement, display the new element in the world at the position (2,2), thanks to the use of the **addObject()** method that we already used to display the borders.

Compile your changes, you should now see Bobby's head at the position (2, 2).



11. Bobby's Movement

To know in which direction Bobby is moving, we will add two new properties. One for the direction in the x-axis and one for the direction along the y-axis.

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

To start with a movement of Bobby to the right, the variable dx is initialize to 1 and dy is initialized to 0.

Now, let's add a method **act()** to our **SnakeWorld**. This method is called by Greenfoot continuously to make the game advance. It calls it on the **World** and on all the **Actors** that have been added to the **World**.

Here follows a first version of the **act()** method.

```
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);
}
```

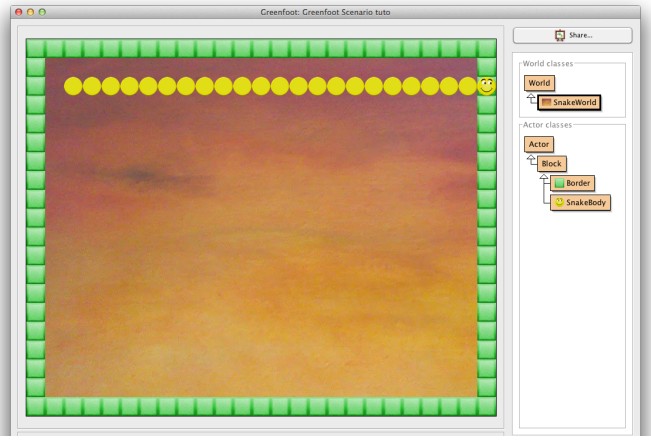
In this first version, we take the current head (i.e. the last element in our list). The **getLast()** method on a **LinkedList** indeed returns the last element of that list.

Next, we change the image of that element so it becomes an image of the tail element of the Snake.

Next, we create a new head with the statement **new SnakeBody()**, and we compute the new position of this new head, based on the current position and the current movement direction (dx et dy).

Finally, we add the new head at the position in the world and we also add it to the list of the snake elements.

Compile your changes and test the game by clicking « **run** » (to ask Greenfoot to call the act method continuously).



The snake is indeed going to the right, but it is also growing at each step until it hits the border.

12. Limit the snake tail size

In order to limit the size of the snake tail, we will add a new property to the world, indicating how many elements of the tails still need to be created.

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

In the **act()** method, after having displayed the new head, we will test the value of this counter:

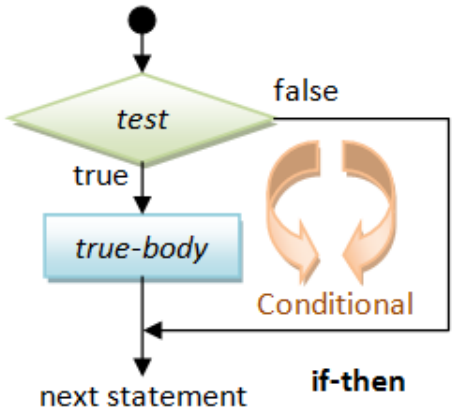
- If it is equal to 0, we will suppress the laste element of the snake tail (which is actually the first element in our list since we always add the new head to the end of the list).
- Otherwise, we decrement the counter of 1

For doing such a test we will use another very important statement of Java the **if statement**.

The syntax of an **if statement** is the following

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

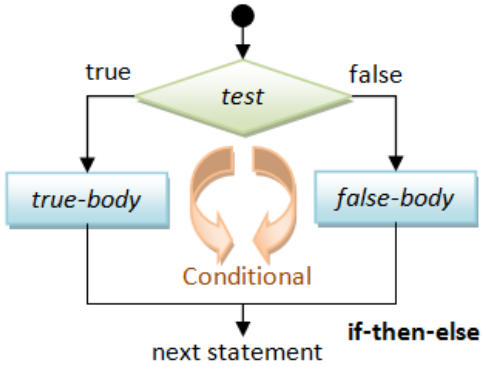
And its execution depends on the **condition** :



Another form of the if statement is the **if-then-else** allowing to define two behaviors:

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

Again, its execution depends on the **condition :**



Here is the code we will add to the **act()** method.

```
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--;
  }
}
```

The **removeFirst()** method on a **LinkedList,** remove the first element of the list and returns it.
The **removeObject()** method, inherited from World, removes the actor passed as argument from the world (and hence suppress it from the screen).

tailCounter-- ; is equivalent to
tailCounter = tailCounter -1 ;

For the 5 first calls to the **act()** method, the **tailCounter** is not equal to 0, the snake grows and the **tailCounter** is decremented of 1.

As soon as the **tailCounter** is equal to 0, the snake stop growing, as we remove a tail element each time we add a new head.

13. Change of direction

Now, we should manage the changes of directions based on the key pressed on the keyboard.

For that, we will create a new method in the **SnakeWorld** class.

```
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;
    }

}
```

The statement **Greenfoot.isKeyDown(« key »)** allows to test if a key is pressed.

To change the direction to the left or the right, it also requires that the current direction is not already left or right because Bobby cannot turn on himself !

We use expressions like the following

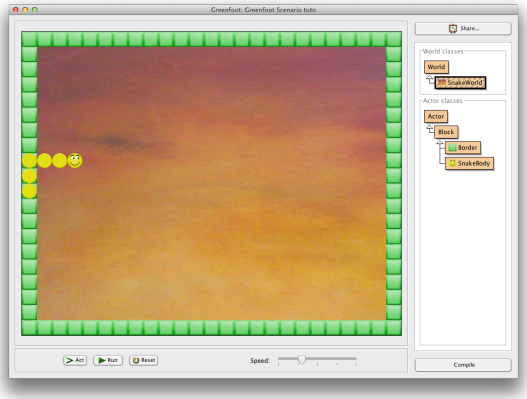
Greenfoot.isKeyDown("left") && dx == 0

Which can be translated with « the left key is pressed **and** dx is equals to 0 ». The operator **&&** corresponds to the **and**, and the operator **==** compares two values to see if they are equals.

It just remains to call this method at the beginning of our act() method.

```
public void act()
{
    changeDirection();
```

Compile your changes and test them.



You can now move Bobby using the keyboard arrow keys.

However, Bobby can go anywhere, even on the borders and he can cross himself.

14. Manage collisions

Before managing the collisions, we will add a property **dead** in our **SnakeWorld** class. It will allow us to stop the game when Bobby has a collision with an 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;
```

This property is a « **boolean** ». This is a simple yet very important type in Java. It can only take two values:

- **true**
- **false**

We can then change the **act()** method so that it does nothing when Bobby is **dead**.

If **dead** is **true**, the method directly returns et the game stops.

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

Let's also add a **dead()** method on the SnakeWorld, to change the value of **dead** :

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

Let's have a look at how to manage « collisions » now. In the **act()** method, *just before displaying the new head*, we will check if there is already another block at the position of the new head. So such blocks, we will execute a **collision** method on the block.

```
//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);
```

The **getObjectsAt()** method, inherited from the **World** class, returns the list of blocks at the **x**, **y** position passed as input parameters that are of the type given as the third parameter (for us, **Block.class**).

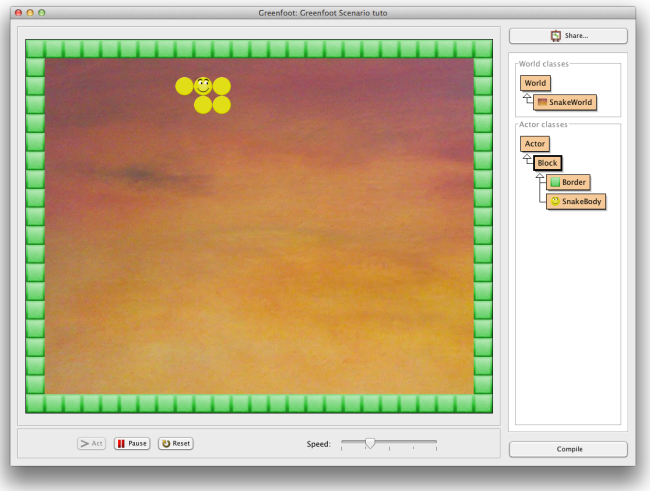
So, this method returns a list of Blocks. The following **for** statement, allows to « loop » on each block of the list and execute the **collision**() method on each of them.
The « **this** » passed as parameter allows to pass the current object itself (in our case, the **SnakeWorld**) to the called method.

At this point, if you compile, there is an error, because the **collision()** method does not yet exist on the **Block** class.

Open the Block class and add the collision method:

```
public class Block extends Actor
{

    public void collision(SnakeWorld world) {
        world.dead();
    }
}
```

This method simply call **dead()** on the world given as parameter. This will directly stop the game.

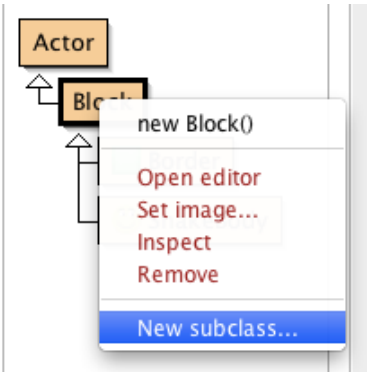


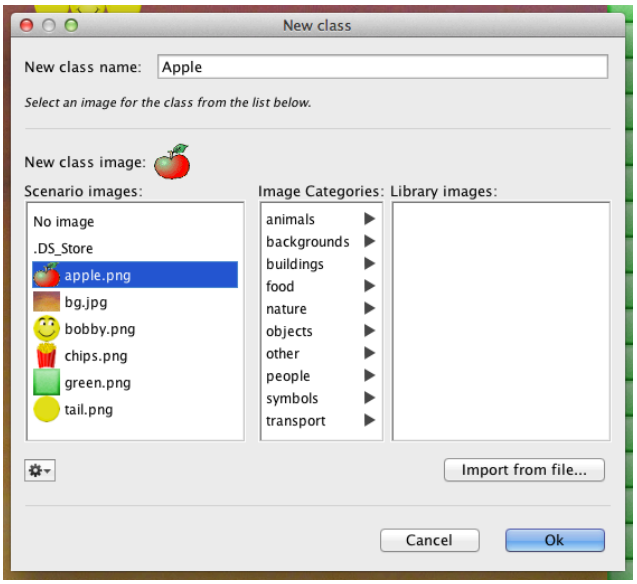
Compile the change and test it.

The game stops as soon as Bobby goes on the border or if he has a collision with its own tail.

15. Add an apple

Let's add now a new subclass of **Block** and name it **Apple**. Choose the **apple.png** image.





Open the new **Apple** class, and delete the **act()** method.

```
public class Apple extends Block
{

}
```

Come back to the **SnakeWorld** class. At the game initialization (in the constructor), after having created Bobby, we will also create an apple and position it **randomly** in the game.

```
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);
```

The apple is created with **new Apple()** statement. Then, it is added to the World using the **addObject()** from Greenfoot. Its position x, y is defined using another Greenfoot method, **getRandomNumber()**. This method takes a number as argument and return a random number between 0 and the given number. The numbers that are passed to this method are the width and height of the game minus 2 (we do not count the borders) and we add 1 (to avoid being on the border).

Compile your changes and test. Try to catch an apple… What happens ?

The game stops! Indeed, we have programmed our game to stop it as soon as the snake has a collision with a block… and the apple is a block !

16. Collision with an apple

We will then « **override** » the **collision** method in the **Apple** class.

In Java, we can indeed **override** an inherited method to define a different behavior in a sub-class.

```
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);
    }
}
```

When the snake enters in collision with an apple, we should not call the **dead()** method anymore, but rather ask the world to make the snake grow (for example of 2 more elements).

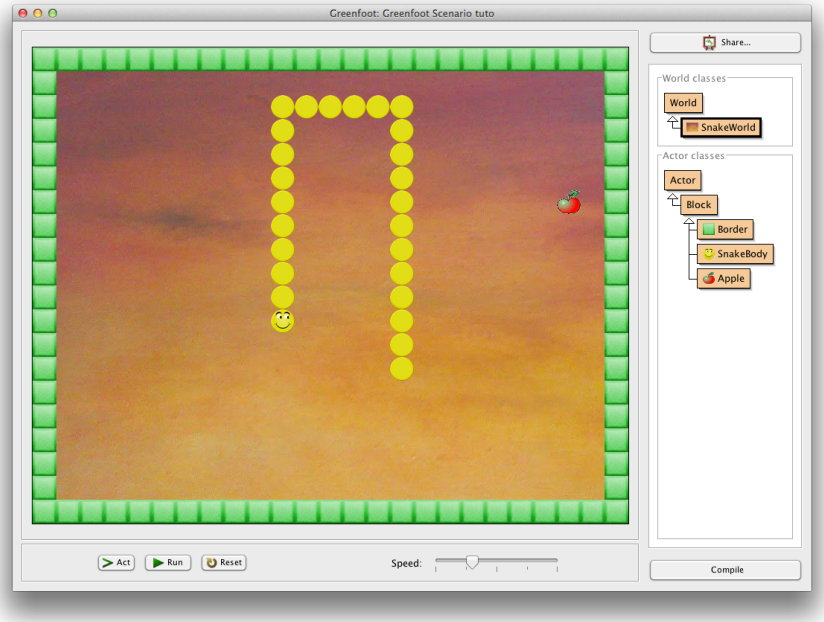
In addition, the apple should also be moved elsewhere in the game.

It just remains to add a **grow()** method in the **SnakeWorld** class:

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

It just need to add the number of elements to our **tailCounter**.

Compile and test again. The snake now grows each time it eats an apple !



17. Add sounds

To improve our game, we will now add some sound when the Bobby eats an apple or it becomes dies.

To play a sound when Bobby eats an apple, simply add the following code in the collision method of the **Apple** class:

```
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);
}
```

Compile and test!

The **playSound()** method takes the name of a sound file as input parameter. This file needs to be in the **sounds** folder of your Greenfoot project folder.

Also add another sound when Booby dies by adding the following code in the collision method of **Block** class:

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

Compile and test!

18. End

Congratulations! You have completed your first Greenfoot game!

You can now start to add new features in your game or start creating new games !