PROGRAMMATION ORIENTÉE OBJET AVEC GREENFOOT

OC Info 14 – 15

Table des matières

1 Environnement et concepts de base

Pour programmer une machine, il faut disposer d'une interface de programmation, c'est à dire d'un logiciel qui va permettre de traduire un langage de programmation en instructions que la machine peut exécuter. Nous avons choisi pour vous l'environnement Greenfoot. Pour pouvoir commencer à programmer en Java dans Greenfoot, il faut tout d'abord télécharger et installer le programme à l'adresse suivante :

`http://www.greenfoot.org/download`

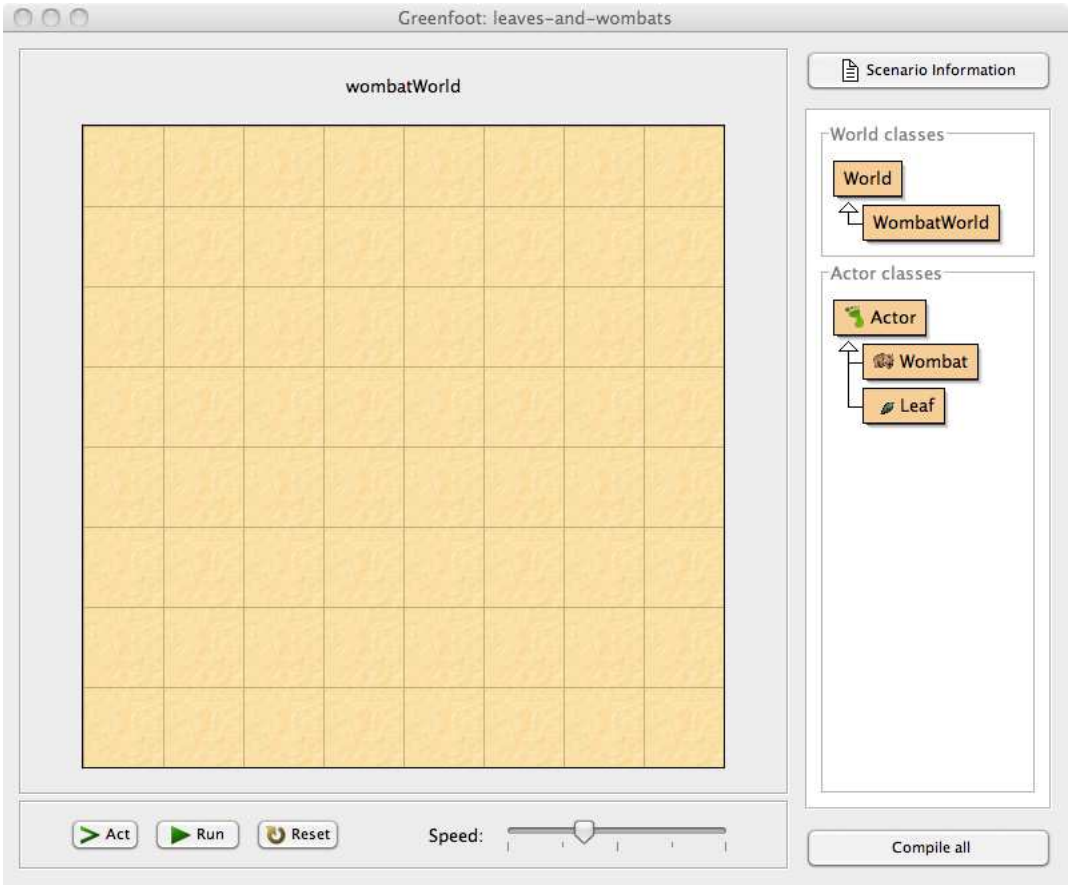
Il faut ensuite télécharger les scénarios du livre qui vont servir de base de travail, à l'adresse

`http://www.greenfoot.org/book`

Cette introduction, tirée du livre *Introduction to programming with Greenfoot* de Michael Kölling, va vous montrer comment programmer des jeux et des simulations avec l'environnement de développement Greenfoot.

1.1 Pour commencer

On commence par lancer l'application Greenfoot et ouvrir le scénario `leaves-and-wombats`.



Dans la fenêtre qui s'est ouverte, on peut observer :

— Le monde du scénario qui se présente ici sous la forme d'une grille sur fond sableux. C'est dans ce monde que pourra agir notre programme, en particulier sur des objets que nous

allons y rajouter.

— Un diagramme de classes qui nous renseigne sur les objets que l'on peut créer et qui donne certaines informations sur la structure hiérarchique du programme du scénario.

— Le panneau du contrôle de l'exécution qui nous permettra de lancer notre programme à l'aide des boutons Act, Run et Reset en particulier.

Commençons par dire quelques mots du diagramme de classes que l'on voit sur la droite de la fenêtre Greenfoot. On peut y observer les classes `World`, `WombatWorld`, `Actor`, `Leaf` et `Wombat`.

1.2 Objets et classes

Nous utiliserons, comme déjà annoncé plus haut, le langage de programmation Java pour nos projets. Java est un langage *orienté objet*. Les concepts de classe et d'objet sont fondamentaux pour ce type de programmation.

Intéressons-nous à la classe `Wombat`. La classe `Wombat` représente le concept général de wombat ; elle décrit tous les wombats qui pourront apparaître dans le scénario. C'est à partir de cette classe `Wombat` que nous pourrons créer des objets du type décrit par cette classe et les disposer dans le monde à notre guise.

Lorsque l'on fait un clic droit sur la classe `Wombat` et que l'on choisit new Wombat() dans le menu, on voit apparaître l'image d'un wombat que l'on peut déplacer dans la fenêtre Greenfoot et déposer dans l'une des cases du monde `WombatWorld`.

Cette opération peut être répétée autant de fois qu'on le désire ; en effet, on peut créer à partir d'une classe autant d'objets ou d'instances de la classe que l'on désire.

Exercice 1.1

Créer quelques wombats et quelques feuilles dans le monde `WombatWorld`.

Nous n'allons pour l'instant nous intéresser qu'aux classes `Wombat` et `Leaf`. Nous reparlerons des autres plus tard.

1.3 Interagir avec des objets

À partir du moment où on a placé un certain nombre d'objets dans notre monde, on peut interagir avec ces objets en faisant un clic droit sur l'objet, ce qui fait apparaître un menu contextuel comme ci-dessous :

Ce menu nous montre quelles sont les actions que l'on peut faire réaliser au wombat en question. On constate ici que le wombat peut agir, bouger, manger une feuille, tourner à gauche, etc.

En Java, ces opérations sont appelées des *méthodes*. Dans le but de prendre des bonnes habitudes tout de suite, nous utiliserons le mot méthode pour les désigner dans la suite de ce texte. On peut *invoquer* une méthode en la sélectionnant depuis le menu.

Exercice 1.2

Invoquer la méthode `move()` d'un wombat. Que fait-il ? Essayer plusieurs fois. Invoquer la méthode `turnLeft()`. Placer deux wombats dans le monde et faire en sorte qu'ils se retrouvent face à face.

Dans Greenfoot, on peut :

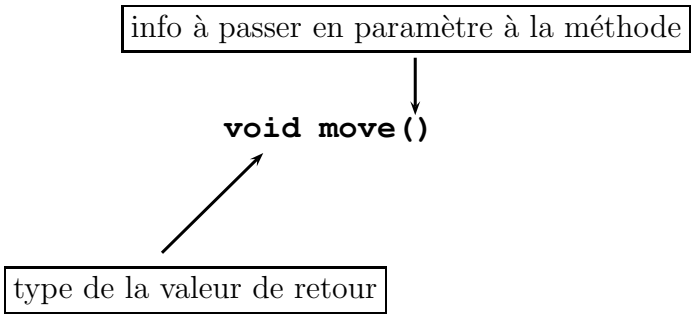
— Créer des objets à partir d'une classe.
— Donner des commandes aux objets en invoquant leurs méthodes.

Observons d'un peu plus près le menu d'un wombat. Les méthodes `move` et `turnLeft` apparaissent dans la liste comme suit :

```
void move()
void turnLeft()
```

Nous observons ici que les noms des méthodes ne sont pas les seules choses qui sont indiquées dans la liste. Il y a aussi le mot `void` avant le nom et une paire de parenthèses le suivant immédiatement. Ces deux éléments cryptiques nous indiquent ce que l'on devra fournir à la méthode pour qu'elle s'exécute et ce qui sera retourné par la méthode après exécution.

On dira de plus que l'ensemble de ces trois éléments forme l'*en-tête* de la méthode.



1.4 Type de la valeur de retour

Le mot placé juste avant le nom d'une méthode s'appelle le type de la valeur de retour. Il nous indique ce que la méthode va renvoyer lorsque nous l'invoquons. Le mot `void` signifie "rien du tout" dans ce contexte : les méthodes dont l'en-tête commence par le mot `void` ne renvoient pas d'information. Elles se bornent à exécuter une action, puis s'arrêtent.

Tout mot différent de `void` nous indique que la méthode renverra quelque chose lorsqu'elle est invoquée et de quel type d'information il s'agit. Dans le menu d'un wombat, on peut également voir les mots `int` et `boolean`. Le mot `int` et un raccourci pour "integer" qui signifie "nombre

entier" et désigne effectivement un nombre entier signé, comme par exemple : 15, 1, −5 et 123 450.

Le type `boolean` n'a que deux valeurs possibles : `true` et `false`, signifiant bien entendu vrai et faux. Une méthode dont le type de valeur de retour est `boolean` retournera la valeur `true` ou la valeur `false`, rien d'autre.

Les méthodes dont le type de valeur de retour est `void` sont comme des ordres pour notre wombat. Si nous invoquons la méthode `turnLeft` d'un wombat, il obéit à l'injonction et tourne à gauche.

Par contre, les méthodes dont la valeur de retour n'est pas `void` sont autant de questions posées à notre wombat. Considérons la méthode

```
boolean canMove()
```

Lorsque nous invoquons cette méthode, nous observons la boîte de dialogue ci-dessous :

L'information importante est ici le mot `true` qui a été renvoyé suite à l'appel de méthode. Nous venons en effet de poser au wombat la question suivante : "Peux-tu bouger ?" et le wombat nous a répondu "Oui !" (`true`).

Exercice 1.3

Invoquer la méthode `canMove()` d'un wombat. La valeur de retour est-elle toujours `true` ? Peut-on mettre le wombat dans une situation où cette valeur sera `false` ?

On va maintenant invoquer une autre méthode dont la valeur de retour n'est pas vide :

```
int getLeavesEaten()
```

En appelant cette méthode, on peut obtenir un nombre indiquant la quantité de feuilles que le wombat a mangées.

Exercice 1.4

Invoquée pour un wombat fraîchement créé, la méthode getLeavesEaten() renvoie systématiquement la valeur 0. Peut-on créer une situation dans laquelle le résultat de l'appel de cette méthode n'est pas égal à 0 ? Peut-on faire en sorte que le wombat mange des feuilles ?

1.5 Paramètres

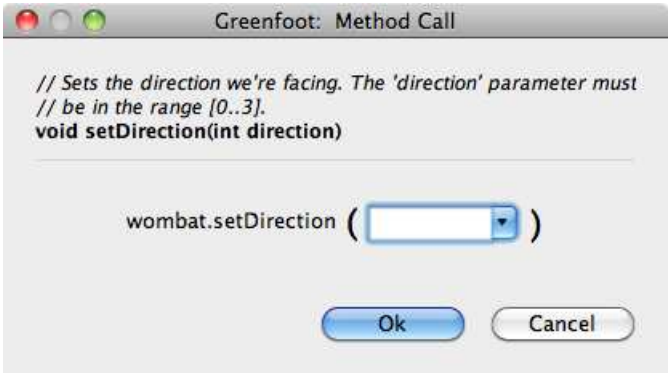
Nous n'avons pas encore parlé des parenthèses qui suivent le nom d'une méthode dans son en-tête :

```
int getLeavesEaten()
void setDirection(int direction)
```

Les parenthèses en question contiennent la *liste des paramètres*. Cela nous indique si la méthode a besoin d'information pour s'exécuter et, si c'est le cas, de quel type d'information.

Si nous ne voyons qu'une paire de parenthèses sans rien à l'intérieur, comme jusqu'à présent, la méthode a une liste de paramètres dans laquelle aucun paramètre n'est spécifié. En d'autres termes, cette méthode n'attend aucun paramètre ; elle ne fera que s'exécuter lorsqu'on l'invoquera. Si, par contre, il y a quelque chose entre les parenthèses, la méthode attend un ou plusieurs paramètres que nous devrons lui fournir au moment de l'invocation.

Essayons maintenant la méthode `setDirection`. Nous pouvons voir les mots `int direction` écrits dans la liste des paramètres de cette méthode. Lorsque nous l'appelons, une boîte de dialogue apparaît :



Les mots `int direction` nous indiquent que cette méthode attend un paramètre de type `int`, qui spécifie une *direction*. Un paramètre est une information supplémentaire que l'on doit fournir à la méthode pour permettre son exécution. Un paramètre est toujours défini au moyen de deux mots : le premier mot est le type du paramètre (ici, `int`) et le second un nom qui donne une idée du rôle que joue le paramètre. Chaque fois qu'une méthode a un paramètre, nous devrons fournir cette information additionnelle lorsque nous invoquerons cette méthode.

Lorsque l'on appelle la méthode `setDirection()`, le type `int` nous indique qu'il faut fournir un nombre entier et le nom `direction` nous suggère que ce nombre spécifie d'une façon ou d'une autre la direction vers laquelle le wombat va tourner.

Dans la fenêtre de dialogue ci-dessus, un commentaire nous indique de plus que le paramètre `direction` devrait être compris entre 0 et 3.

Exercice 1.5

Créer un wombat dans `WombatWorld`.

a) Invoquer la méthode `setDirection(int direction)` de ce wombat.

b) Essayer différentes valeurs du paramètre et observer ce qui se produit.

c) Quel nombre correspond à quelle direction ? Établir un petit tableau de correspondance entre les quatre directions possibles et les nombres donnés en paramètre à cette méthode.

d) Que se passe-t-il lorsque l'on donne en paramètre un nombre supérieur à 3 ?

e) Que se passe-t-il si l'on fournit à la méthode un nombre à virgule (1.8, par exemple) ou un mot (deux, par exemple) ?

La méthode `setDirection` d'un wombat n'attend qu'un seul paramètre. Plus tard, nous verrons des cas dans lesquels plusieurs paramètres doivent être passés à une méthode. Dans ce genre de cas, on trouvera entre les parenthèses de l'en-tête de méthode une liste de tous les paramètres, séparés par des virgules.

La description de chaque méthode dans le menu contextuel associé à un objet, composée d'un type de valeur de retour, du nom de la méthode et d'une liste de paramètres entre parenthèses, s'appelle l'*en-tête* de méthode ou *signature* de la méthode.

Nous sommes maintenant capables d'effectuer des interactions élémentaires avec les objets de Greenfoot. Nous pouvons créer des objets à partir de classes, interpréter les signatures des méthodes et invoquer des méthodes avec ou sans paramètre.

1.6 Exécution dans Greenfoot

Il y a une autre façon d'interagir avec des objets de Greenfoot : le panneau de contrôle de l'exécution.

Exercice 1.6

Placer un wombat et un certain nombre de feuilles dans le monde des wombats et invoquer ensuite la méthode `act` du wombat plusieurs fois. Que fait cette méthode ? Quelle différences peut-on trouver entre la méthode `act` et la méthode `move` ? Faire attention à tester différentes situations, le wombat à la bordure du monde, faisant face à l'extérieur, ou encore le wombat étant sur la même case qu'une feuille.

Exercice 1.7

À nouveau, placer un wombat et un bon nombre de feuilles dans le monde des wombats. Ensuite, cliquer sur le bouton Act du panneau de contrôle de l'exécution placé en bas de la fenêtre Greenfoot, sous le monde des wombats. Que se passe-t-il ?

Exercice 1.8

Quelle est la différence entre l'action résultante d'un clic sur le bouton Act du panneau de contrôle et l'invocation de la méthode `act()` d'un wombat ? On essayera avec plusieurs wombats disposés dans le monde.

Exercice 1.9

Cliquer sur le bouton Run. Que se passe-t-il ?

La méthode `act()` d'un objet Greenfoot est d'une importance capitale. Nous la rencontrerons régulièrement dans la suite de ce texte. Tous les objets de Greenfoot sont munis de cette

méthode. Invoquer `act()` sur un objet revient essentiellement à lui donner l'instruction suivante: "Fais ce que tu veux faire maintenant".

On peut résumer ici les différentes choses qu'un wombat réalise lorsqu'on invoque sa méthode `act()` :

a) Si le wombat est sur une case dans laquelle se trouve aussi une feuille, l'invocation de la méthode `act()` lui fera manger la feuille.

b) S'il n'y a rien à manger dans la case et que la voie est libre, le wombat avance d'une case.

c) Si aucune des deux actions précédentes n'est possible, le wombat tourne à gauche.

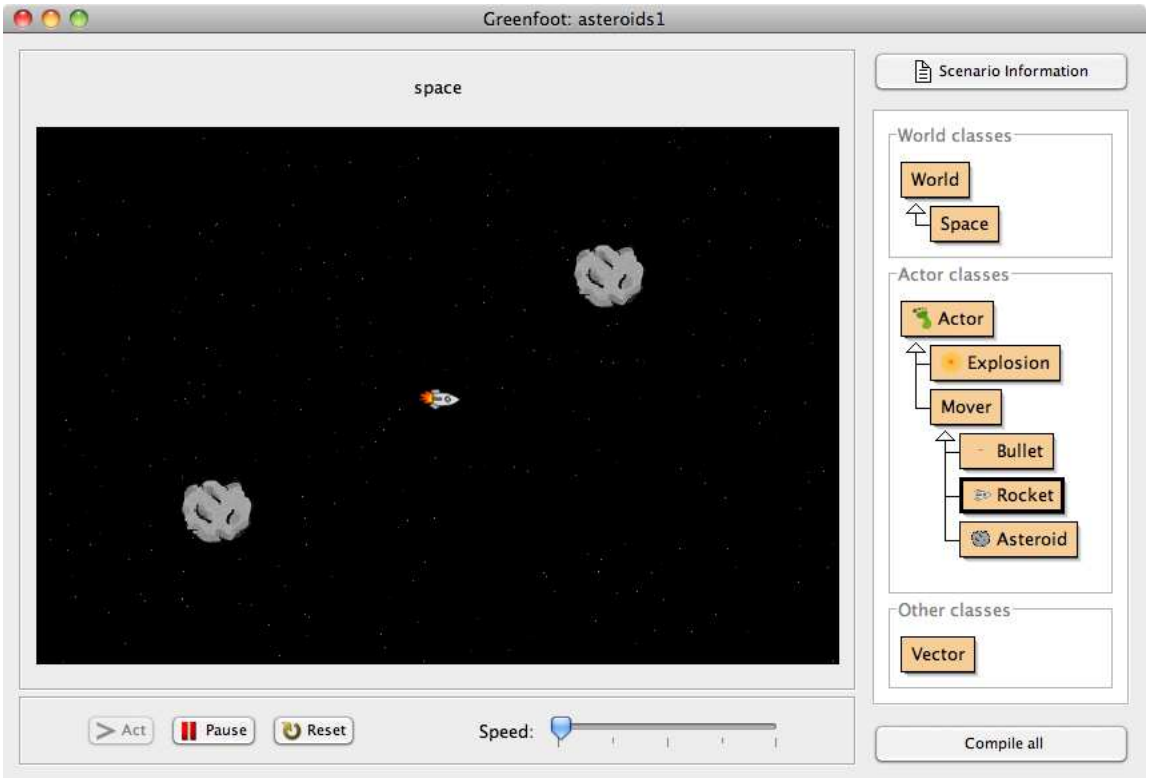
Les exercices ci-dessus devraient avoir également montré que le bouton Act du panneau de contrôle appelle simplement la méthode `act()` de tous les acteurs du monde `WombatWorld`. Le menu objet d'un wombat donné permet simplement d'invoquer la méthode sur ce seul objet.

Le bouton Run du panneau de contrôle permet, quant à lui, d'invoquer la méthode `act()` de tous les objets encore et encore jusqu'à ce que l'utilisateur clique sur le bouton Pause.

Essayons maintenant d'appliquer ce qui a été vu jusqu'ici à un autre scénario.

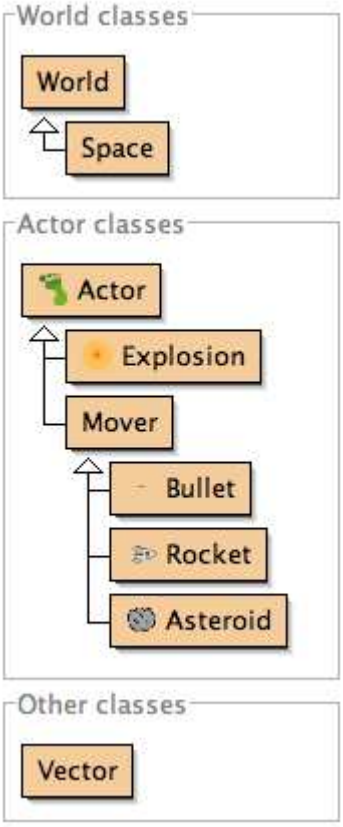
1.7 Un deuxième exemple

On ouvre maintenant un autre scénario, nommé `asteroids1`, qui se trouve dans le dossier du chapitre 1 des scénarios Greenfoot. Une fenêtre s'ouvre alors, qui ressemble à la copie d'écran ci-dessous à cela près qu'elle ne contient encore ni fusée, ni astéroïde.



1.8 Comprendre le diagramme de classes

Observons maintenant le diagramme de classes de ce scénario d'un peu plus près.



Au sommet du diagramme se trouvent les classes nommées `World` et `Space`, reliées par une flèche ascendante.

La classe `World` est présente dans tous les scénarios Greenfoot ; elle est intégrée à Greenfoot. La classe juste en dessous, `Space` dans ce cas, représente le monde particulier à ce scénario avec ses spécificités propres. Chaque scénario aura un monde spécifique à cet endroit, dont le nom peut bien entendu varier.

La flèche qui relie les deux classes représente une relation *"is-a"*: `Space` *is a* `World` (relativement au concept de monde dans Greenfoot : `Space` est ici un monde Greenfoot particulier). On dira aussi parfois que `Space` est une *sous-classe* de la classe `World`.

Nous n'avons pas, en général, à créer des objets à partir des classes de type `World` ; Greenfoot le fait à notre place. Lorsque nous ouvrons un scénario, Greenfoot crée automatiquement un objet de la sous-classe de la classe `World`. L'objet occupe alors la plus grande partie de la fenêtre Greenfoot. Ci-dessus, dans le scénario `asteroids1`, la grande image noire constellée d'étoiles est un objet de la classe `Space`.

Plus bas dans le diagramme de classes, on voit le rectangle des sous-classes de la classe `Actor`. Dans notre scénario, il y a six classes reliées entre elles par des flèches. Chaque classe représente un objet particulier. En commençant à lire depuis le bas, on trouve une classe astéroïde, une classe fusée et une classe projectile qui sont tous des "mobiles", alors que les mobiles et les explosions sont des acteurs.

À nouveau, nous avons des relations hiérarchiques, certaines de ces classes sont des sous-classes d'autres classes : la classe `Rocket`, par exemple, est une sous-classe de la classe `Actor`, vu que la classe `Mover` est elle-même une sous-classe de la classe `Actor`. Nous étudierons plus loin la signification détaillée des termes sous-classe et super-classe.

Reste la classe Vector, tout en bas du diagramme classée sous le titre de *Other classes*. Cette classe ne fait que servir aux autres classes, on ne peut pas créer d'objet à partir de cette classe pour les placer directement dans le monde.

1.9 Jouer avec les fusées et autres astéroïdes

Pour pouvoir jouer avec ce scénario, il faut commencer par créer des objets acteurs (des objets construits à partir des sous-classes de la classe `Actor`) et placer ces acteurs dans le monde. Nous ne pouvons ici créer que des objets qui n'ont pas de sous-classes : `Rocket`, `Bullet`, `Asteroid` and `Explosion`.

On commence par placer une fusée et deux astéroïdes dans l'espace. Rappelons qu'il est possible de créer des objets en faisant un clic droit sur la classe, ou en sélectionnant la classe et faisant un majuscule-clic dans l'espace.

Après avoir placés vos objets, cliquez le bouton *Run*. Vous pourrez alors contrôler votre engin spatial à l'aides des flèches du clavier et tirer des projectiles à l'aide de la barre d'espace. Essayez de vous débarrasser des astéroïdes sans vous écraser sur l'un d'entre eux.

Exercice 1.10

Après avoir joué un certain temps, force vous sera de constater que la cadence de tir n'est pas très élevée. Ajustons un peu le software qui contrôle le tir de notre vaisseau de sorte à pouvoir augmenter la cadence de tir, ce qui devrait nous aider à éliminer les astéroïdes. Plaçons une fusée dans l'espace et invoquons sa méthode `setGunReloadTime` à partir du menu contextuel de l'objet pour pouvoir attribuer la valeur 5 au temps qu'il faut pour recharger le canon de la fusée.

Jouer à nouveau avec au moins deux astéroïdes pour observer les effets du changement.

Exercice 1.11

Après avoir éliminé les astéroïdes ou simplement après avoir joué un moment en faisant en sorte de ne pas faire exploser la fusée, arrêter l'exécution du programme en cliquant le bouton Pause et déterminer le nombre de coups tirés. On peut le faire en utilisant une méthode trouvée dans le menu contextuel de la fusée.

Exercice 1.12

On peut observer que la fusée se déplace lentement dès qu'elle est placée dans l'espace et que le bouton Run a été cliqué. Quelle est sa vitesse initiale ?

Exercice 1.13

Les astéroïdes ont une `stabilité` qui leur est propre. Chaque fois qu'ils sont touchés par un projectile, leur stabilité décroît. Lorsqu'elle atteint zéro, ils cassent. Quelle est la valeur de la stabilité d'un astéroïde au moment de sa création ? De combien d'unités la stabilité d'un astéroïde diminue

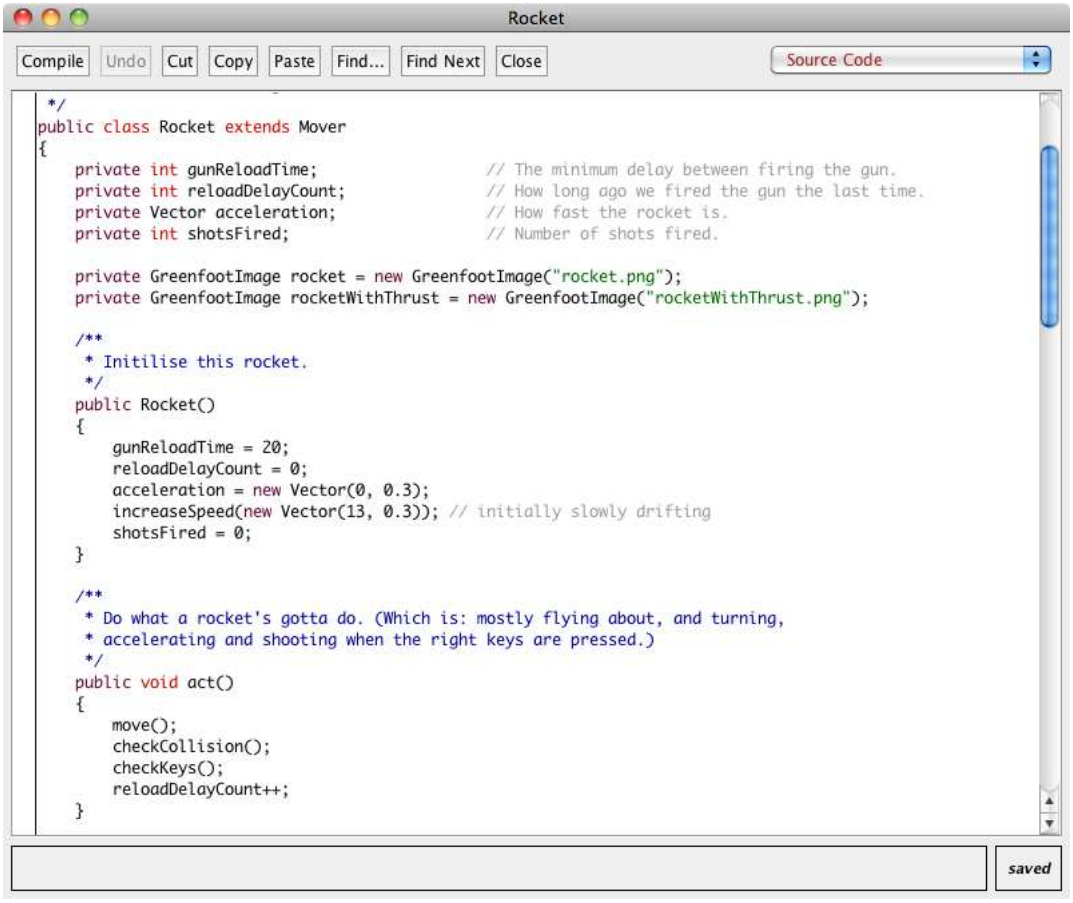
lorsqu'il est touché par un projectile ?

Exercice 1.14

Créer un astéroïde énorme.

1.10 Code source

Le comportement de tout objet Greenfoot est défini par sa classe. Pour spécifier ce comportement, on écrira du *code source* dans le langage de programmation Java. Le code source d'une classe est le code qui spécifie tous les détails concernant une classe et ses objets. En sélectionnant Open editor dans le menu contextuel de la classe, on ouvre une fenêtre de l'éditeur de code qui contient le code source de la classe :




```
 */
public class Rocket extends Mover
{
    private int gunReloadTime;                  // The minimum delay between firing the gun.
    private int reloadDelayCount;               // How long ago we fired the gun the last time.
    private Vector acceleration;                // How fast the rocket is.
    private int shotsFired;                     // Number of shots fired.

    private GreenfootImage rocket = new GreenfootImage("rocket.png");
    private GreenfootImage rocketWithThrust = new GreenfootImage("rocketWithThrust.png");

    /**
     * Initilise this rocket.
     */
    public Rocket()
    {
        gunReloadTime = 20;
        reloadDelayCount = 0;
        acceleration = new Vector(0, 0.3);
        increaseSpeed(new Vector(13, 0.3)); // initially slowly drifting
        shotsFired = 0;
    }

    /**
     * Do what a rocket's gotta do. (Which is: mostly flying about, and turning,
     * accelerating and shooting when the right keys are pressed.)
     */
    public void act()
    {
        move();
        checkCollision();
        checkKeys();
        reloadDelayCount++;
    }
```

Le code source de la classe `Rocket` est plutôt complexe et nous n'avons pas besoin de le comprendre complètement à ce stade. Si toutefois vous étudiez sérieusement l'entier du livre *introduction to programming with Greenfoot* et que vous programmez vos propres jeux et simulations, vous apprendrez petit à petit comment écrire ce code.

À ce niveau, il nous suffit de comprendre que nous pouvons changer le comportement des objets d'une classe Greenfoot en modifiant son code source. Essayons de le faire.

Nous avons déjà vu plus haut que la cadence de tir par défaut d'une fusée est plutôt faible. Nous pourrions changer ceci pour chaque fusée individuellement en invoquant une méthode

pour chaque nouvelle fusée placée dans l'espace, mais nous devrions la faire chaque fois que nous désirons jouer une nouvelle partie. Nous pouvons plutôt changer le code source de la fusée, de sorte que la cadence de tir initiale soit changée (disons à 5) et de façon à ce que toutes les nouvelles fusées créées par la suite aient ce comportement amélioré.

Ouvrez l'éditeur à partir du menu contextuel de la classe `Rocket`. À environ 25 lignes du haut du texte, vous devriez trouver la ligne suivante :

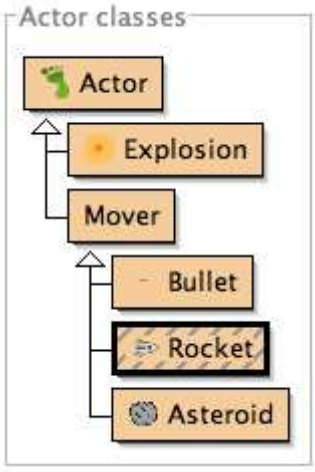
```
gunReloadTime = 20;
```

C'est à cet endroit que la cadence de tir d'une fusée est définie. Modifiez le code de la façon suivante :

```
gunReloadTime = 5;
```

Faites bien attention à ne rien modifier d'autre. Vous vous rendrez compte très rapidement que les environnements de programmation sont très sensibles. Un seul caractère faux ou manquant peut conduire à des erreurs. Si d'aventure vous éliminiez le point virgule de fin de ligne, vous rencontreriez une erreur très rapidement.

Fermez maintenant la fenêtre de l'éditeur – nous en avons fini avec le code pour l'instant – et observez le diagramme de classes. Celui-ci a changé : la classe fusée est maintenant hachurée :



Le fait qu'une classe soit ainsi mise en évidence nous indique que son code source a été modifié et qu'il faut la *compiler*. La compilation est un processus de traduction : le code source de la classe est traduit en code machine que l'ordinateur peut exécuter.

On devra toujours compiler une classe après avoir changé son code source, avant de pouvoir créer des objets de cette classe à nouveau. Il faudra parfois même recompiler plusieurs classes après avoir modifié le code source de l'une d'entre elles. En effet, les classes dépendent souvent les unes des autres et un changement de l'une d'elle produit une cascade d'effets sur les autres.

Nous pouvons compiler toutes les classes en cliquant le bouton Compile en bas à droite de la fenêtre principale de Greenfoot. Une fois que les classes ont été compilées elles ne sont plus hachurées et nous pouvons créer des objets à nouveau.

Exercice 1.15

Faire le changement décrit ci-dessus en modifiant le code source de la classe `Rocket`. Fermer l'éditeur et compiler les classes. Tester la modification : les fusées devraient pouvoir tirer rapidement dès le départ.

1.11 Résumé

Dans ce chapitre, nous avons vu à quoi ressemblent les scénarios de Greenfoot et comment interagir avec eux. Nous avons également vu comment créer des objets et comment communiquer avec ces objets en invoquant leurs méthodes. Certaines de ces méthodes sont des ordres donnés à l'objet, alors que d'autres méthodes renvoient de l'information à propos de l'objet. Les paramètres sont utilisés pour fournir de l'information supplémentaire aux méthodes alors que les valeurs de retour renvoient de l'information à celui qui a appelé la méthode.

Les objets sont créés à partir de leur classe, et le code source contrôle la définition de la classe, autrement dit le comportement et les caractéristiques de tous les objets de cette classe.

Nous avons vu que nous pouvons changer le code source en utilisant un éditeur. Après qu'on a édité le code source d'une classe pour la modifier, il faut la recompiler.

Dans la suite du cours, on cherchera à comprendre comment écrire du code source Java pour créer des scénarios qui font des choses intéressantes et amusantes.

2 Little Crab, un premier programme

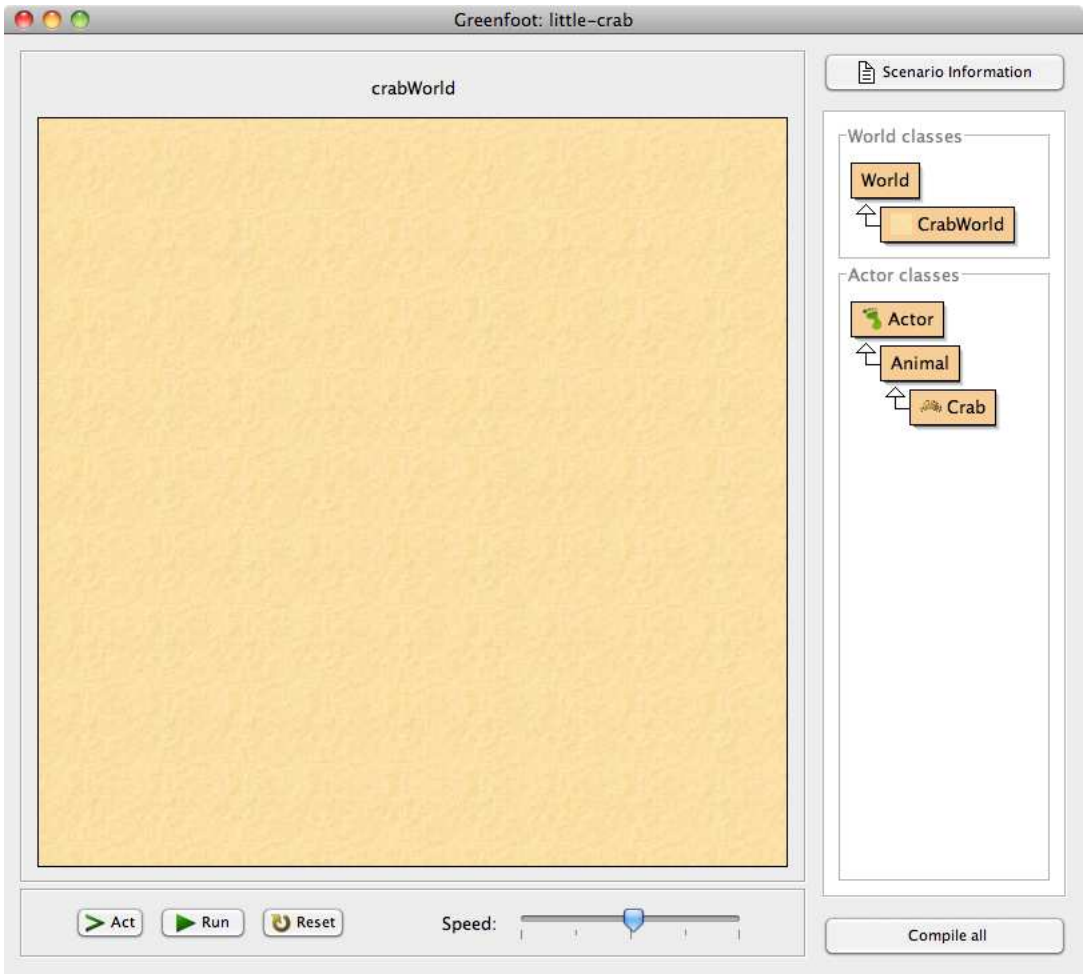
Dans le chapitre précédent, nous avons discuté comment utiliser des scénarios Greenfoot existants : nous avons créé des objets, invoqué des méthodes, et joué.

Nous allons commencer maintenant à créer notre propre jeu.

2.1 Le scénario Little Crab

Le scénario que nous utiliserons tout au long de ce chapitre s'appelle `little-crab`. Vous le trouverez dans le dossier des scénarios du livre.

Lorsqu'on ouvre le scénario, on voit apparaître la fenêtre ci-dessous :



Exercice 2.1

Après avoir ouvert le scénario `little-crab`, placez un crabe dans le monde et lancez l'exécution du programme en cliquant le bouton Run. Qu'observez-vous ?

Sur la droite de la fenêtre du scénario, on voit le diagramme de classes, comme pour les scénarios du chapitre précédent. Observons ici qu'il y a la classe Greenfoot usuelle `Actor`, une classe appelée `Animal` et une classe `Crab`.

La hiérarchie, figurée par les flèches, dénote une relation *is-a*, appelée également *héritage* : Un crabe *est un* animal, et un animal *est un* acteur. Il s'ensuit bien entendu alors qu'un crabe est aussi un acteur.

Nous travaillerons tout d'abord seulement avec la classe `Crab`. Nous parlerons un peu plus des classes `Actor` et `Animal` plus tard.

Si vous avez fait l'exercice plus haut, alors vous connaissez la réponse à la question "Qu'observez-vous ?". La réponse est : "Rien du tout".

Le crabe ne fait rien du tout lorsque Greenfoot s'exécute. La raison en est qu'il n'y a pas de code source dans la définition de la classe `Crab` qui spécifie ce que le crabe devrait faire.

Dans ce chapitre, nous allons travailler à modifier cela. Il s'agit pour commencer de faire se déplacer notre crabe.

2.2 On fait bouger le crabe

Portons notre attention sur le code source de la classe `Crab`. Ouvrons l'éditeur pour faire apparaître ce code. (On peut le faire en sélectionnant la fonction Open editor du menu déroulant de la classe, ou simplement en double-cliquant sur l'icône de la classe.)

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

/**
* This class defines a crab. Crabs live on the beach.
*/
public class Crab extends Animal
{
    public void act()
    {
    }
}
```

Ceci est un définition de classe standard en Java. En d'autres termes, le texte ci-dessus définit ce que le crabe peut faire. Nous verrons cela en détail un peu plus tard. Pour l'instant, nous nous concentrerons sur notre but : faire bouger le crabe.

Dans cette définition de classe, nous voyons ce que nous avons appelé la *méthode act*. Elle ressemble à ceci :

```
public void act()
{
    // Pour faire agir le crabe, ajouter votre code ici.
}
```

La première ligne est l'*en tête de la méthode* ou *signature*. Les trois dernières lignes – les deux accolades et tout ce qui se trouvent entre elles – forment le *corps* de la méthode. C'est à cet endroit que nous pouvons ajouter du code qui va déterminer les actions de notre crabe. Nous

pouvons remplacer le *commentaire* du milieu par une commande. La commande que nous choisissons ici est

```
move();
```

Notons ici qu'il faut l'écrire exactement comme ci-dessus, parenthèses et point virgule compris. La méthode `act` devrait ressembler maintenant à ceci :

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

Exercice 2.2

Modifier la méthode `act` de la class `Crab` de façon à y inclure l'instruction `move()`, comme décrit ci-dessus. Compiler le scénario en cliquant sur le bouton Compile All et placer un crabe dans le monde. Cliquer ensuite sur les boutons Act et Run.

Exercice 2.3

Placer maintenant plusieurs crabes dans le monde. Faites tourner le scénario. Que peut-on observer ?

Vous verrez que le crabe se déplace maintenant sur l'écran. L'instruction `move()` fait bouger le crabe un petit peu vers la droite. Lorsque nous cliquons le bouton Act de la fenêtre principale de Greenfoot, la méthode `act()` est exécutée une seule fois. C'est à dire que les instructions que nous avons écrites à l'intérieur de la méthode `act` s'exécutent.

Cliquer le bouton Run revient à cliquer le bouton Act plusieurs fois, très rapidement, sans s'arrêter. La méthode `act` est donc exécutée en boucle dans ce contexte, jusqu'à ce que nous cliquions le bouton Pause.

Terminologie

L'instruction `move()` est un **appel de méthode**. Une **méthode** est une action qu'un objet sait faire (ici, l'objet est le crabe) et un **appel de méthode** est une instruction indiquant au crabe de réaliser cette action. Les parenthèses font partie de l'appel de méthode. Les instructions de ce genre se terminent par un point virgule.

2.3 Pour tourner

Voyons quelle autre genre d'instruction on peut utiliser. Le crabe comprend également l'instruction `turn`. Voici à quoi elle ressemble :

```
turn(5);
```

Le nombre 5 dans l'instruction spécifie de combien de degrés le crabe doit tourner. Cela s'appelle un *paramètre*.

Nous pouvons également utiliser d'autres nombres, par exemple :

```
turn(23);
```

On peut utiliser pour ce paramètre toute valeur comprise entre 0 et 359 degrés. (Tourner de 360 degrés nous ferait faire un tour complet, ce qui revient à tourner de 0 degrés ou ne pas tourner du tout.)

Si nous voulons faire tourner notre crabe plutôt que de le faire se déplacer, nous pouvons remplacer l'instruction `move()` par `turn(5)`. La méthode `act` aura alors l'allure suivante :

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

Exercice 2.4

Remplacer `move()` par `turn(5)` dans votre scénario. Tester le résultat. Changer le nombre passé en paramètre à la méthode `turn` et voir ce que cela donne. On rappelle ici qu'il faut compiler les classes concernées après chaque changement dans le code source.

Exercice 2.5

Comment faire pour que le crabe tourne dans l'autre sens ?

Nous essayons maintenant de faire bouger et tourner le crabe en même temps. La méthode `act` peut en effet contenir plus d'une instruction ; nous pouvons écrire plusieurs instructions à la suite les unes des autres.

le code ci-dessous montre la classe `Crab` complétée de façon à ce que le crabe bouge et tourne à la fois. D'après le code source ci-dessous, chaque fois que l'on presse sur le bouton Act, le crabe se déplace et ensuite tourne (ces actions se produise à une telle vitesse l'une après l'autre qu'elles semblent se produire au même moment).

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

/**
 * This class defines a crab. Crabs live on the beach.
 */
public class Crab extends Animal
{
    public void act()
    {
        move();
        turn(5);
    }
}
```

Exercice 2.6

Placer les instructions `move()` et `turn(N)` à la suite l'une de l'autre dans la méthode `act` de votre classe `Crab`. Essayez différentes valeurs pour `N`.

Terminologie

Le nombre placé entre parenthèses dans l'instruction `turn` – c'est à dire, le 5 de `turn(5)` – s'appelle un *paramètre*. Un paramètre est une information supplémentaire que nous devons fournir lorsque nous appelons certaines méthodes.

Certaines méthodes, `move` par exemple, n'attendent aucun paramètre. Elles se contentent de s'exécuter dès lors que nous écrivons l'instruction `move()`. D'autres méthodes, comme `turn` demande de l'information en plus : *De combien dois-je tourner ?* Dans ce cas, nous devons fournir cette information sous la forme d'un paramètre à l'intérieur des parenthèses, `turn(17)`, par exemple.

Lorsque nous écrivons du code source, nous devons être très prudents ; chaque caractère compte. Une seule petite erreur, et notre programme ne fonctionne plus ! En général, il ne pourra pas être compilé avec des erreurs.

Cela nous arrivera régulièrement : lorsque nous écrivons des programmes, nous faisons inévitablement des erreurs et nous devons ensuite les corriger. Voyons ce qui se passe lorsque nous nous trompons.

Si, par exemple, nous oublions le point virgule après l'instruction `move()`, l'environnement Greenfoot émettra un message d'erreur lors d'un essai de compilation.

Exercice 2.7

Ouvrir l'éditeur pour voir le code source du crabe et ôter le point-virgule après `move()`. Cliquer ensuite le bouton de compilation. Expérimenter aussi avec d'autres types d'erreur : une faute d'orthographe dans `move` ou un changement aléatoire dans le code source. Faire bien attention à rétablir l'état initial du code après cet exercice.

Exercice 2.8

Faire des changements variés pour obtenir des messages d'erreur variés également. Trouver au minimum cinq messages d'erreur différents. Prendre note des messages et des changements introduits dans le code ayant provoqué ces messages.

Comme nous avons pu le constater au travers des exercices ci-dessus, si nous nous trompons ne serait-ce qu'un peu, Greenfoot va ouvrir l'éditeur, mettre une ligne en évidence et écrire un message d'erreur en bas de la fenêtre de l'éditeur. Le but du message est de chercher à expliquer l'erreur. Toutefois, les messages peuvent varier considérablement dans leur précision et utilité. Parfois, ils nous indiquent de façon assez précise quel est le problème, mais ils sont parfois cryptiques et difficiles à comprendre. La ligne mise en évidence par Greenfoot est souvent la ligne qui pose problème, mais c'est parfois la ligne précédent celle-ci qui recèle l'erreur. Lorsque vous voyez, par exemple, un message du type "`; expected`", il est possible que le point-virgule manque en fait à la ligne juste en dessus.

Vous apprendrez à lire de mieux en mieux ce type de messages au cours du temps. Pour l'instant, si vous recevez un message et que vous n'êtes pas sûr de sa signification, examinez attentivement

votre code et vérifiez que vous avez tout tapé correctement.

2.4 Au bord du monde

Dans les paragraphes précédents, lorsque nous faisions tourner nos crabes, il se retrouvaient coincés très rapidement, dès lors qu'ils parvenaient au bord de notre monde. (La structure de Greenfoot ne permet pas aux acteurs de quitter le monde en passant outre ses limites.)

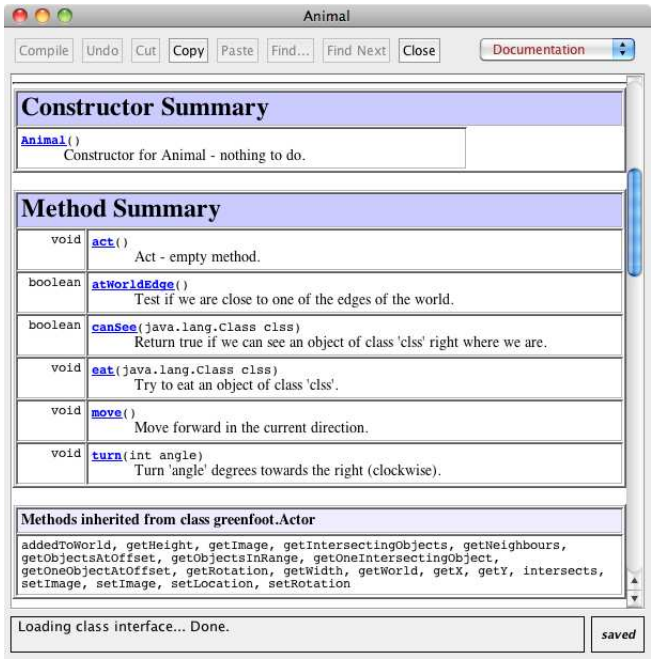
Nous allons maintenant améliorer ce comportement de façon à ce que le crabe se rende compte qu'il a atteint un des bords du monde et qu'il se mette alors à tourner sur lui-même. Une question se pose : comment faire cela ?

Plus haut, nous avons utilisé les méthodes `move` et `turn` ; il y a peut-être une autre méthode qui peut nous aider à réaliser notre nouvel objectif. (En fait, c'est le cas.) Mais comment pouvons-nous trouver quelles méthodes sont à notre disposition ?

Les méthodes `move` et `turn` que nous avons utilisées jusqu'à présent proviennent de la classe `Animal`. Un crabe est un animal, comme indiqué par la flèche qui va de la classe `Crab` à la classe `Animal` dans le diagramme de classes ; il peut donc faire tout ce qu'un animal fait en général. Un animal décrit par la classe `Animal` sait comment tourner et bouger ; c'est pour cela que le crabe peut aussi le faire. Cela s'appelle *l'héritage* : la classe `Crab` hérite de toutes les aptitudes (méthodes) de la classe `Animal`.

Une question se pose maintenant : que peut faire d'autre notre animal ?

Pour découvrir cela, nous pouvons ouvrir l'éditeur de la classe `Animal`. L'éditeur peut présenter deux visages différents : il peut présenter le code source (comme nous l'avons constaté en étudiant la classe `Crab` ou il peut montrer la *documentation*. On peut passer de l'un à l'autre en utilisant le menu déroulant qui se trouve dans le coin en haut à droite de la fenêtre de l'éditeur. Voyons maintenant à quoi ressemble la documentation de la classe `Animal` :



Exercice 2.9

Ouvrir l'éditeur pour la classe `Animal`. Passer à la présentation Documentation. Trouver la liste des méthodes de la classe ("Method summary"). De combien de méthode dispose cette classe ?

Lorsque nous observons le "method summary", nous pouvons voir toute les méthodes dont la classe `Animal` dispose. Parmi celles-ci, il y a trois méthodes qui éveillent notre attention pour le moment. Ce sont :

`boolean atWorldEdge()`

Teste si nous sommes proche de l'une des frontières du monde.

`void move()`

Avance dans la direction courante.

`void turn(int angle)`

Tourne de "angle" degrés vers la droite (dans le sens des aiguilles d'une montre).

Nous voyons ici les en-têtes des trois méthodes, comme nous les avons découverts au chapitre 1. Chaque en-tête de méthode commence par un type de valeur de retour et est suivi d'un nom de méthode et de la liste des paramètres. En dessous se trouve un commentaire qui décrit ce que fait la méthode.

Nous avons utilisé les méthodes `move` et `turn` dans les paragraphes précédents. Nous faisons à nouveau la constatation suivante : `move` n'a pas de paramètres (les parenthèses sont vides), et on doit fournir à `turn` un seul paramètre de type `int` (un nombre entier) pour l'angle.

On peut voir également que les méthodes `move` et `turn` ont `void` comme type de valeur de retour. Cela signifie qu'aucune de ces deux méthodes ne retourne de valeur. En invoquant l'une de ces deux méthodes, nous donnons l'ordre à notre objet de tourner ou d'avancer. L'animal se bornera à obéir mais ne nous donnera aucune réponse.

L'en tête de la méthode `atWorldEdge` est un peu différent :

```
boolean atWorldEdge()
```

Cette méthode n'a pas de paramètre (il n'y a rien entre les parenthèses), mais elle spécifie un type de valeur de retour : `boolean`. Nous avons rencontré le type `boolean` au paragraphe 1.4 ; c'est un type qui comporte deux valeurs, `true` ou `false`.

Appeler des méthodes dont le type de valeur de retour n'est pas `void` ne se borne pas à donner une instruction à l'objet sur lequel on a appelé la méthode, mais revient également à poser une question. Si nous utilisons la méthode `atWorldEdge()`, elle va nous répondre `true` (oui !) ou `false` (non !). Nous pouvons donc utiliser cette méthode pour savoir si notre objet se trouve au bord du monde.

Exercice 2.10

Créer un crabe. En utilisant le menu contextuel de cet objet, trouver la méthode `atWorldEdge()` qui se trouve dans le sous-menu inherited from Animal, vu que le crabe hérite de cette méthode de la classe `Animal`. Appeler cette méthode. Quelle est la valeur de retour ?

Exercice 2.11

Faire se déplacer le crabe jusqu'au bord du monde à l'aide du bouton Run ou manuellement, et ensuite appeler la méthode `atWorldEdge()` à nouveau. Que retourne-t-elle maintenant ?

Nous pouvons maintenant combiner l'appel de cette méthode avec une *instruction conditionnelle if* pour écrire le code présenté ci-dessous :

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

/**
 * This class defines a crab. Crabs live on the beach.
 */
public class Crab extends Animal
{
       public void act()
    {
        if ( atWorldEdge() )
        {
            turn(17);
        }

        move();
    }
}
```

L'instruction conditionnelle if fait partie du langage Java et permet d'exécuter des instructions seulement si une certaine condition est vraie. Par exemple, nous voulons tourner ici uniquement si nous sommes à la frontière du monde. Le code que nous avons écrit pour tester la condition est :

```
if ( atWorldEdge() )
{
    turn(17);
}
```

La forme générale d'une instruction conditionnelle if est la suivante :

```
if ( condition )
{
    instruction;
    instruction;
    ...
}
```

À la place de `condition`, on peut mettre toute expression qui est vraie ou fausse (telle notre appel de la méthode `atWorldEdge()`), et les `instructions` seront exécutées seulement si la `condition` est vraie. Il peut y avoir plus d'une instruction.

Si la `condition` est fausse, les `instructions` seront ignorées, l'exécution continuant à partir de la suite du programme qui se trouve juste en dessous de l'accolade qui ferme l'instruction conditionnelle if.

Notons que l'appel de méthode `move()` est hors de l'instruction conditionnelle if, ce qui fait qu'il sera exécuté à chaque fois. En d'autres termes : Si nous nous trouvons à la frontière du monde, nous tournons et nous déplaçons ensuite ; si nous ne sommes pas au bord du monde, nous ne faisons que de nous déplacer.

Exercice 2.12

Modifier le code source de la classe `Crab` de sorte à ce qu'il soit comme ci-dessus et observer si le crabe tourne maintenant au bord du monde. On fera attention à l'écriture des accolades ouvrantes et fermantes ; il est facile d'en oublier ou d'en rajouter une en trop !

Exercice 2.13

Changer la valeur du paramètre de la méthode `turn`. Trouver une valeur qui donne un bon résultat au niveau visuel.

Exercice 2.14

Placer l'instruction `move()` à l'intérieur de l'instruction conditionnelle if, plutôt qu'après celle-ci. Observer l'effet sur le comportement du crabe et expliquer ce qui se passe. (Rétablir ensuite le code d'origine.)

Note sur l'indentation

En examinant le code source que l'on vous a présenté en exemple jusqu'à présent, vous avez peut-être remarqué l'indentation des différentes lignes de ce code. Chaque fois qu'une accolade s'ouvre, les lignes qui suivent sont indentées un cran vers la droite de plus que les précédentes. Lorsqu'une accolade se ferme, l'indentation est diminuée d'un cran vers la gauche, de sorte à ce que l'accolade fermante se trouve exactement au dessous de l'accolade ouvrante correspondante. Cela nous aide à retrouver l'accolade correspondante.

Nous utilisons quatre espaces pour un niveau d'indentation. La touche de tabulation insère automatiquement les quatre espaces d'un niveau d'indentation.

Il est très important de faire attention à l'indentation de votre propre code. Si vous n'indentez pas votre code soigneusement, certaines de vos erreurs (en particulier concernant des accolades mal placées ou mal appareillées) deviennent très difficile à trouver. L'indentation correcte rend le code source plus facile à lire, et donc permet d'éviter des erreurs potentielles.

2.5 Résumé des techniques de programmation

Dans ce livre, nous abordons la programmation d'un point de vue complètement axé sur la pratique. Nous introduisons des techniques générales de programmation au moment où nous en avons besoin pour améliorer nos scénarios. Nous ferons donc un résumé des techniques de programmation importantes à la fin de chaque chapitre pour indiquer clairement ce que vous devez vraiment retenir de la discussion pour bien progresser.

Dans ce chapitre, nous avons vu comment appeler des méthodes (telles la méthode `move()`), avec ou sans paramètres. Cela formera la base de toute programmation Java subséquente. Nous avons également appris à identifier le corps de la méthode `act`–c'est à cet endroit que nous avons pu insérer des instructions.

Vous vous êtes heurtés à des messages d'erreur. Cela continuera durant toute votre vie de programmeur. Nous faisons tous des erreurs et nous nous heurtons tous à des messages d'erreur. Ce n'est pas signe que nous sommes des mauvais programmeurs—cela fait partie intégrante de la programmation.

Nous avons eu un premier aperçu de la notion d'héritage : Les classes héritent des méthodes des classes qui leur sont supérieures. La *documentation* d'une classe donne un résumé des méthodes à disposition.

Finalement, chose très importante, nous avons vu comment prendre des décisions : Nous avons utilisé une *instruction conditionnelle if* pour pouvoir faire exécuter sous condition une partie du code source. Cette instruction est intimement liée avec l'existence d'un type `boolean`, une valeur qui peut être `true` ou `false`.

3 Améliorer le Crabe à l'aide de programmation plus sophistiquée

Dans le chapitre précédent, nous avons étudié les éléments de base permettant de débuter la programmation de notre premier jeu. Nous avons dû voir plusieurs choses nouvelles. Nous allons maintenant rendre notre crabe capable de comportements plus intéressant. Nous aurons maintenant un peu plus de facilité à ajouter du code dans la mesure ou nous avons vu beaucoup de concepts fondamentaux.

Nous allons nous occuper en premier lieu de comportement aléatoire.

3.1 Ajouter du comportement aléatoire

Dans notre implémentation actuelle, le crabe peut se déplacer sur l'écran, et il peut tourner lorsqu'il parvient à un bord de notre monde. Mais, lorsqu'il se déplace, il le fait toujours exactement en ligne droite. C'est ce que nous voulons changer maintenant. Les crabes ne se déplacent pas toujours strictement en ligne droite ; ajoutons donc un peu de déplacement aléatoire : le crabe devrait grosso modo se déplacer en ligne droite, mais devrait de temps en temps changer un peu sa direction.

Nous pouvons atteindre cet objectif dans Greenfoot en utilisant des nombres aléatoires. L'environnement Greenfoot lui-même dispose d'une méthode qui va nous fournir un nombre aléatoire. Cette méthode, appelée `getRandomNumber`, doit recevoir un paramètre qui spécifie une limite supérieure pour le nombre aléatoire. Elle nous retournera un nombre pris au hasard entre 0 (zéro) et la limite. Par exemple,

```
Greenfoot.getRandomNumber(20)
```

nous donnera un nombre aléatoire entre 0 et 20. Le nombre 20, qui est ici notre limite, est en fait exclu et le nombre que nous obtenons est compris entre 0 et 19.

La notation utilisée dans l'instruction ci-dessus s'appelle la *qualification des noms* ou *dot notation* en anglais. Relier ainsi des noms à l'aide de points est une manière de désigner sans ambiguïté des éléments faisant partie d'ensembles, lesquels peuvent eux-même faire partie d'ensembles plus vastes, etc. Par exemple, l'étiquette `systeme.machin.truc` désigne l'élément `truc`, qui fait partie de l'ensemble `machin`, lui-même faisant part de l'entité `systeme`.

Lorsque nous appelions des méthodes définies dans la classe courante ou une classe dont la classe courante hérite, il suffisait de noter le nom de la méthode et sa liste de paramètres. Si une méthode est définie dans une autre classe que la classe courante, nous devons spécifier quelle est la classe ou l'objet qui dispose de cette méthode, faire suivre d'un point, et terminer par le nom de la méthode et ses paramètres. Vu que la méthode `getRandomNumber` n'est pas définie dans la classe `Crab` ou `Animal`, mais dans une classe nommée `Greenfoot`, nous devons écrire "`Greenfoot.`" devant l'appel de méthode.

Note : Les méthodes statiques

Les méthodes peuvent appartenir à des objets ou à des classes. Lorsqu'une méthode est une méthode de classe, nous écrivons

```
nomClasse.nomMéthode (paramètres);
```

pour appeler la méthode. Lorsqu'une méthode appartient à un objet, nous écrivons

```
nomObjet.nomMéthode (paramètres);
```

pour l'appeler.

Les deux types de méthodes sont définies dans une classe. L'en-tête de méthode nous indique s'il s'agit d'une méthode associée aux objets de cette classe, ou plutôt à la classe elle-même.

Les méthodes de classe sont caractérisées par la présence du mot-clef `static` au début de l'en-tête de la méthode. Par exemple, l'en-tête de la méthode qui permet de générer des nombres aléatoires dans Greenfoot est

```
static int getRandomNumber(int limit)
```

Cela nous indique qu'il faut écrire le nom de la classe elle-même (Greenfoot) avant le point dans l'appel de méthode.

Disons maintenant que nous voulons programmer notre crabe de sorte à ce qu'il y ait 10 pour cent de chances à chaque étape du mouvement que le crabe dévie un peu de sa course. Nous pouvons régler l'essentiel du problème avec une instruction conditionnelle :

```
if ( quelque-chose-est-vrai )
{
    turn(5);
}
```

Nous devons alors trouver une expression à mettre à la place de `quelque-chose-est-vrai` qui renvoie `true` dans exactement dix pour cent des cas.

Nous pouvons le réaliser en utilisant un nombre aléatoire (à l'aide de l'appel de la méthode `Greenfoot.getRandomNumber`) et de l'opérateur "strictement plus petit que". L'opérateur "strictement plus petit que" compare deux nombres et renvoie `true` si le premier est inférieur au second. Cet opérateur est représenté à l'aide du symbole "<". Par exemple, l'expression

```
2 < 33
```

est vraie, alors que

```
162 < 42
```

est fausse.

Exercice 3.1

Avant de continuer la lecture du chapitre, tenter de coucher sur le papier une expression qui emploie la méthode `getRandomNumber` et l'opérateur "strictement plus petit que" qui renvoie la valeur `true` exactement 10% du temps.

Exercice 3.2

Écrire une autre expression qui est vraie 7% du temps.

Exercice 3.3

Trouver tous les opérateurs de comparaison mis à disposition en Java. Par exemple, l'opérateur "plus petit ou égal" est noté à l'aide du symbole <=.

Si nous voulons exprimer une probabilité en %, il est plus facile d'employer des nombres aléatoires générés avec une borne supérieure valant 100. Une expression s'évaluant à `true` dans 10% des cas serait, par exemple

```
Greenfoot.getRandomNumber(100) < 10
```

Vu que l'appel `Greenfoot.getRandomNumber(100) < 10` nous donne un nouveau nombre aléatoire compris entre 0 et 99 chaque fois que nous l'appellons, et vu que ces nombres sont uniformément distribués, ils seront inférieurs à 10 dans 10% des cas.

Nous pouvons maintenant faire usage de tout cela pour faire tourner le crabe dans 10% des étapes d'exécution :

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

 /**
 * This class defines a crab. Crabs live on the beach.
 */
public class Crab extends Animal
{
    public void act()
    {
        if ( atWorldEdge() )
        {
            turn(17);
        }

        if ( Greenfoot.getRandomNumber(100) < 10 )
        {
            turn(5);
        }

        move();
    }
}
```

Exercice 3.4

Modifier le code de la classe `Crab` comme exposé ci-dessus. Tester différentes probabilités.

C'est un bon début, mais ce n'est pas encore tout à fait ce que nous voulons. Premièrement, lorsque le crabe tourne, il tourne toujours du même angle, soit 5 degrés. Deuxièmement, il tourne toujours vers la droite, jamais vers la gauche. Le comportement que nous aimerions vraiment avoir est le suivant : le crabe tourne aléatoirement d'un petit angle vers la gauche ou vers la droite. (Nous allons détailler la solution à ce problème dans les lignes qui suivent. Si vous vous sentez sûr de vous, vous pouvez l'implémenter vous-même avant de continuer votre lecture.)

Il y a un moyen simple d'éviter que le crabe tourne toujours de la même quantité. Il suffit d'introduire un nombre aléatoire dans notre code source à la place du nombre 5.

```
if ( Greenfoot.getRandomNumber(100) < 10 )
{
    turn(Greenfoot.getRandomNumber(45));
}
```

Dans l'exemple de code ci-dessus, le crabe tourne toujours dans 10% des cas, et, lorsqu'il tourne, ce sera d'un certain nombre de degrés compris entre 0 et 44.

Exercice 3.5

Essayer le code source ci-dessus. Qu'observez-vous ? L'angle de rotation du crabe varie-t-il comme prévu ?

Exercice 3.6

Nous devons encore trouver un moyen de faire tourner notre crabe dans les deux sens, pour lui donner un comportement de crabe normal. Modifier le code de façon à ce que le crabe tourne à gauche ou à droite d'un angle compris entre 0 et 44 degrés chaque fois qu'il change d'orientation.

Exercice 3.7

Exécuter le scénario avec plusieurs crabes placés à différents endroits dans le monde. Tournent-ils tous en même temps ou indépendamment les uns des autres ? Pourquoi ?

Le projet little-crab-2 des scénarios du livre donne une implémentation complète de tout ce qui a été discuté dans ce chapitre jusqu'ici, incluant les derniers exercices.

3.2 Ajouter des vers de sable

Nous allons rendre le monde du crabe un peu plus intéressant en y ajoutant une autre sorte d'animal.

Les crabes mangent des vers de sable. (Cela n'est pas vrai de tous les crabes du monde réel, mais certains le font. On va considérer ici que notre crabe fait partie de cette espèce "vers_de_sablophage".) Ajoutons donc une classe spécifique aux vers.

Nous pouvons ajouter une nouvelle classe acteur à un scénario Greenfoot en sélectionnant New subclass dans le menu contextuel d'une des classes acteur existante.

Dans ce cas, notre nouvelle classe `Worm` est une sorte d'animal spécifique et doit donc être une sous-classe de la classe `Animal`. Souvenez-vous du fait que la relation qui lie une sous-classe avec sa classe est une relation *is a* : a worm *is an* animal.

Lorsque nous créons une nouvelle sous-classe, une fenêtre de dialogue apparaît, dans laquelle

nous devons remplir un champ de nom de classe et choisir une image :

Nous avons choisi ici le nom `Worm` pour notre nouvelle classe. Conventionnellement, les noms de classe en Java commencent toujours par une majuscule. Ils devraient aussi décrire le genre d'objet que la classe représente, ce qui justifie le choix du nom `Worm` pour une classe représentant des vers.

Dans Greenfoot, nous associons de plus une image à la classe. Il y a quelques images associées à notre scénario et toute une librairie d'images génériques parmi lesquelles on peut choisir. Dans le cas qui nous occupe, nous avons préparé une image de ver et l'avons mise à disposition dans la liste des images du scénario ; il ne nous reste donc qu'à sélectionner l'image dont le nom est worm.png.

Une fois que cela a été fait, il ne nous reste plus qu'à cliquer Ok. La classe fait maintenant partie de notre monde, nous pouvons la compiler et ajouter ensuite des vers à notre monde de crabes.

Exercice 3.8

Après avoir compilé la classe `Worm`, peupler le monde avec quelques vers et quelques crabes. Lancer l'exécution du scénario en cliquant Run. Qu'observez-vous ? Que font les vers ? Que se passe-t-il lorsqu'un crabe rencontre un ver ?

Nous avons maintenant compris comment ajouter des classes à nos scénarios. La tâche suivante est de faire interagir ces classes : lorsqu'un crabe rencontre un ver, il devrait le manger.

3.3 Manger des vers de sable

Nous voulons maintenant ajouter un type de comportement à notre crabe : Lorsque le crabe se trouve au même emplacement qu'un ver, il le mange. À nouveau, nous consultons notre liste de méthodes de la classe `Animal` pour savoir de quelles méthodes le ver a hérité directement de cette super-classe. Après avoir ouvert la classe `Animal` dans l'éditeur et avoir passé à la vue Documentation, nous voyons qu'il existe les deux méthodes suivantes :

```
boolean canSee (java.lang.Class clss)
```
Return true if we can see an object of class "clss" right where we are.

`void eat (java.lang.Class clss)`
Try to eat an object of class “clss”.

Nous pouvons implémenter le type de comportement cherché en utilisant ces deux méthodes. La première teste si le crabe voit un ver. (Il ne peut le voir que lorsqu'il lui rentre dedans–nos animaux ont la vue très courte !) Cette méthode nous renvoie un `boolean`–`true` ou `false`, nous pouvons donc l'utiliser dans une instruction conditionnelle `if`.

La seconde méthode permet de manger un ver. On doit passer à ces deux méthodes un paramètre de type `java.lang.Class`. Cela signifie que nous devons leur fournir l'une des classes de notre scénario. On peut le coder comme suit :

```
if ( canSee(Worm.class) )
{
    eat(Worm.class);
}
```

Nous avons passé `Worm.class` en paramètre aux deux appels de méthode (`canSee` et `eat`). Cela nous permet de déclarer quel objet nous sommes en train de rechercher et que nous cherchons ensuite à manger.

Voici maintenant la méthode `act` complète :

```
public void act()
{
    if ( atWorldEdge() )
    {
        turn(17);
    }

    if ( Greenfoot.getRandomNumber(100) < 10 )
    {
        turn(Greenfoot.getRandomNumber(90) - 45);
    }

    if ( canSee(Worm.class) )
    {
        eat(Worm.class);
    }
    move();
}
```

Tester le code source ci-dessus. Placer un certain nombre de vers dans le monde, placer quelques crabes, lancer l'exécution du scénario et observer ce qui se passe.

Note : Les packages

Dans la définition de la méthode `canSee` et `eat`, nous avons vu un paramètre dont le nom de type est `java.lang.class`. Qu'est-ce que cela signifie ?

Il y a beaucoup de types définis par des classes. Un très grand nombre de ces classes font partie de la bibliothèque des classes standard de Java. Vous pouvez accéder à la documentation de cette bibliothèque de classes en sélectionnant *Java library Documentation* du menu *Help* de Greenfoot.

La bibliothèque des classes de Java contient des milliers de classes. Pour faciliter la tâche du programmeur, elles ont été regroupées en *packages*, qui sont des groupes de classes ayant un rapport les unes avec les autres. Lorsqu'un nom de classe contient des points, comme `java.lang.Class`, seule la dernière part est le nom de la classe lui-même, et ce qui précède forme le nom du package. La classe dont il est question ici est donc la classe `Class` du package `java.lang`.

Essayer de trouver cette classe dans la documentation de la bibliothèque Java.

3.4 Créer de nouvelles méthodes

Dans quelques-uns des paragraphes précédents, nous avons ajouté des comportements nouveaux à notre crabe : se tourner au bord du monde, changer aléatoirement de direction de temps en temps, et manger des vers. Si nous continuons de la sorte, la méthode `act` de la classe `Crab` va devenir de plus en plus longue et difficile à comprendre en fin de compte. Nous pouvons éviter cette situation en découpant cette méthode en petits morceaux.

Nous pouvons en effet créer nos propres méthodes dans la classe `Crab` pour atteindre nos objectifs. Par exemple, au lieu de placer directement dans la méthode `act` le code source qui cherche la présence d'un ver et qui le mange le cas échéant, nous pouvons créer une nouvelle méthode qui réalise ces actions. Nous devons commencer par choisir un nom pour la méthode, disons `LookForWorm`. Nous pouvons alors créer notre nouvelle méthode en ajoutant au code source de la classe `Crab` le code suivant :

```
/**
* Teste si nous sommes au meme endroit qu'un ver de sable.
* Si c'est le cas, on le mange. Sinon, on ne fait rien.
*/
public void lookForWorm()
{
    if ( canSee(Worm.class) )
    {
        eat(Worm.class)
    }
}
```

Les quatre premières lignes forment un *commentaire*. Un commentaire est ignoré par le compilateur ; il est écrit pour le lecteur humain. Nous employons le commentaire pour expliquer au lecteur humain à quoi sert la méthode.

Lorsque nous définissons cette méthode, le code n'est pas exécuté immédiatement. Il n'y a en fait aucune raison que la méthode soit exécutée une fois que son code source figure dans une classe. En écrivant le code source d'une méthode, nous nous bornons à définir une action potentielle ("rechercher la présence d'un ver") qui pourra être exécutée plus tard. Effectivement, cette action ne sera réalisée qu'au moment de l'appel de la méthode.

C'est pour cela qu'il nous faut intégrer l'appel de cette méthode à la méthode `act` :

```
lookForWorm()
```

Observons encore une fois que l'appel de méthode comprend les parenthèses ouvrante et fermante pour les éventuels paramètres. Voici le code source complet de la méthode `act` après restructuration :

```
public void act()
{
    if ( atWorldEdge() )
    {
        turn(17);
    }

    if ( Greenfoot.getRandomNumber(100) < 10 )
    {
        turn(Greenfoot.getRandomNumber(90) - 45);
    }

    move();
    lookForWorm();
}

/**
* Teste si nous sommes au meme endroit qu'un ver de sable.
* Si c'est le cas, on le mange. Sinon, on ne fait rien.
*/
public void lookForWorm()
{
    if ( canSee(Worm.class) )
    {
        eat(Worm.class);
    }
}
```

Notons que ce changement dans le code source ne change absolument pas le comportement de notre crabe. Cela ne fait que rendre la compréhension du code plus facile à long terme. Plus nous ajoutons de code à la classe, plus les méthodes seront longues. Des méthodes plus longues sont plus difficiles à comprendre. En décomposant notre code en un certain nombre de méthodes plus courtes, nous le rendons plus facile à lire.

Exercice 3.9

Créer une nouvelle méthode dont le nom est `randomTurn` (cette méthode n'a pas de paramètres et ne renvoie rien). Isoler le code qui s'occupe de faire tourner le crabe aléatoirement, et le déplacer de la méthode `act` à la méthode `randomTurn`. Appeler ensuite cette nouvelle méthode `randomTurn` depuis la méthode `act`. Prendre soin d'ajouter un commentaire pour cette méthode.

Exercice 3.10

Créer encore une nouvelle méthode dont le nom est `turnAtEdge` (cette méthode n'a pas de paramètres et ne renvoie rien non plus). Déplacer le code concerné dans cette nouvelle méthode. Appeler la méthode `turnAtEdge` depuis votre méthode `act`. Votre méthode `act` devrait être identique à l'extrait de code ci-dessous.

```
public void act()
{
    turnAtEdge();
    randomTurn();
    move();
    lookForWorm();
}
```

Par convention, les noms de méthodes commencent toujours, en Java, par une minuscule. Les noms de méthode ne peuvent pas contenir d'espaces ni la plupart des caractères de ponctuation. Si le nom de la méthode est formé de plusieurs mots, on utilisera, comme ci-dessus, des majuscules au milieu du nom de méthode pour marquer le début de chaque mot.

3.5 Ajouter un homard

Nous disposons maintenant d'un crabe qui se déplace plus ou moins aléatoirement dans notre monde et qui mange les vers de sable lorsqu'il en rencontre un.

Pour rendre les choses plus intéressantes, ajoutons une nouvelle créature : un homard.

Les homards de notre scénario chassent les crabes.

Exercice 3.11

Ajouter une nouvelle classe à notre scénario. Cette classe devrait être une sous-classe de la classe `Animal`, nommée `Lobster` avec "`L`" majuscule et devrait employer l'image *lobster.png*.

Exercice 3.12

Que se passe-t-il lorsque vous placez un homard tel-quel dans le monde des crabes ? Compiler le scénario et faire un test.

Nous voulons maintenant programmer nos homards de sorte à ce qu'ils mangent des crabes. C'est relativement facile à faire, vu que leur comportement est similaire à celui des crabes. La seule différence est que les homards cherchent des crabes, alors que les crabes cherchent des vers.

Exercice 3.13

Copier l'entier de la méthode `act` de la classe `Crab` dans la classe `Lobster`. Copier également les méthodes `lookForWorm`, `turnAtEdge` et `randomTurn`.

Exercice 3.14

Changer le code source de la classe `Lobster` de sorte à ce que les homards cherchent les crabes plutôt que les vers. On peut le faire en remplaçant chaque occurence du mot "Worm" par le mot "Crab". On changera par exemple `Worm.class` en `Crab.class`. Modifier également le nom de la méthode : `LookForWorm` deviendra `LookForCrab`. Prendre soin également de mettre à jour ses commentaires.

Exercice 3.15

Placer un crabe, trois homards et de nombreux vers de sable dans le monde des crabes. Lancer l'exécution du scénario. Le crabe arrive-t-il à manger tous les vers avant de se faire manger lui-même par un homard ?

Vous devriez, à ce stade, avoir à disposition une version de votre scénario dans laquelle à la fois des crabes et des homards se déplacent aléatoirement, cherchant à manger des vers et des crabes, respectivement.

Nous allons maintenant faire en sorte de changer ce programme en un jeu.

3.6 Contrôler le crabe à l'aide du clavier

Il est bien entendu qu'il n'y a pas de jeu sans joueur ! Le joueur en question doit pouvoir contrôler le crabe à partir du clavier, tandis que les homards continuent à se déplacer aléatoirement dans le monde, comme ils le font déjà.

L'environnement Greenfoot dispose d'une méthode qui nous permets de tester si quelqu'un a pressé sur une touche du clavier. Cette méthode s'appelle `isKeyDown` et, tout comme la méthode `getRandomNumber` que nous avons rencontré au paragraphe 3.1, il s'agit d'une méthode de classe. L'en-tête de la méthode `isKeyDown` est le suivant :

```
public static boolean isKeyDown(String key)
```

Nous pouvons voir que cette méthode est statique (c'est une méthode de classe) et que son type de valeur de retour est `boolean`. Cela signifie que sa valeur de retour est `true` ou `false`, et qu'elle peut être utilisée directement dans le test conditionnel d'une instruction conditionnelle `if`. Nous voyons également que la méthode attend un paramètre de type `String`. Un `String` est un morceau de texte, un mot ou une phrase, écrit entre guillemets. En voici des exemples :

"`Ceci est une chaîne de caractères`"
"`nom`"
"`A`"

Dans le cas qui nous occupe, la `String` attendue est le nom de la touche du clavier que nous voulons "écouter". Chaque touche du clavier a un nom. Pour les touches qui produisent des caractères visibles, ce caractère est leur nom ; par exemple, la touche A s'appelle "A". Les autres touches ont également un nom.

La flèche gauche s'appelle "left". Ainsi, lorsque nous voudrons tester si la flèche gauche a été pressée, nous écrirons

```
if (Greenfoot.isKeyDown("left"))
{
    ...// faire quelque chose
}
```

Notons qu'il nous faut spécifier `Greenfoot.` au début de l'appel de la méthode `isKeyDown`, vu que cette méthode est définie dans la classe `Greenfoot`.

Si nous désirons que notre crabe tourne à gauche de 4 degrés chaque fois que l'on est en train d'appuyer sur la flèche gauche, nous écrirons

```
if (Greenfoot.isKeyDown("left"))
{
    turn(-4);
}
```

L'idée maintenant est d'enlever le code de la classe `Crab` qui gère les changements de direction aléatoires et également celui qui s'occupe de faire tourner le crabe au bord du monde, et de remplacer tout cela par le code qui nous permet de contrôler le crabe à partir du clavier.

Exercice 3.16

Enlever le code de la classe `Crab` qui fait changer le crabe de direction aléatoirement.

Exercice 3.17

Enlever le code de la classe `Crab` qui fait tourner le crabe au bord du monde.

Exercice 3.18

Ajouter à la méthode `act` de la classe `Crab` du code qui fait tourner le crabe vers la gauche chaque fois que la flèche gauche est pressée. Tester le code.

Exercice 3.19

Par analogie, ajouter à la méthode `act` de la classe `Crab` du code qui fait tourner le crabe vers la droite chaque fois que la flèche droite est pressée.

Exercice 3.20

Si vous ne l'avez pas fait spontanément, faites en sorte que le code qui teste l'état de la touche "flèche gauche" ne soit pas écrit directement dans la méthode `act`, mais bien dans une méthode séparée s'appelant `checkKeypress`, par exemple. Un appel à cette méthode devra être présent dans la méthode `act`.

Le corrigé des exercices ci-dessus se trouvent dans le scénario du livre little-crab-3. Cette version comprend tous les changements discutés jusqu'à présent.

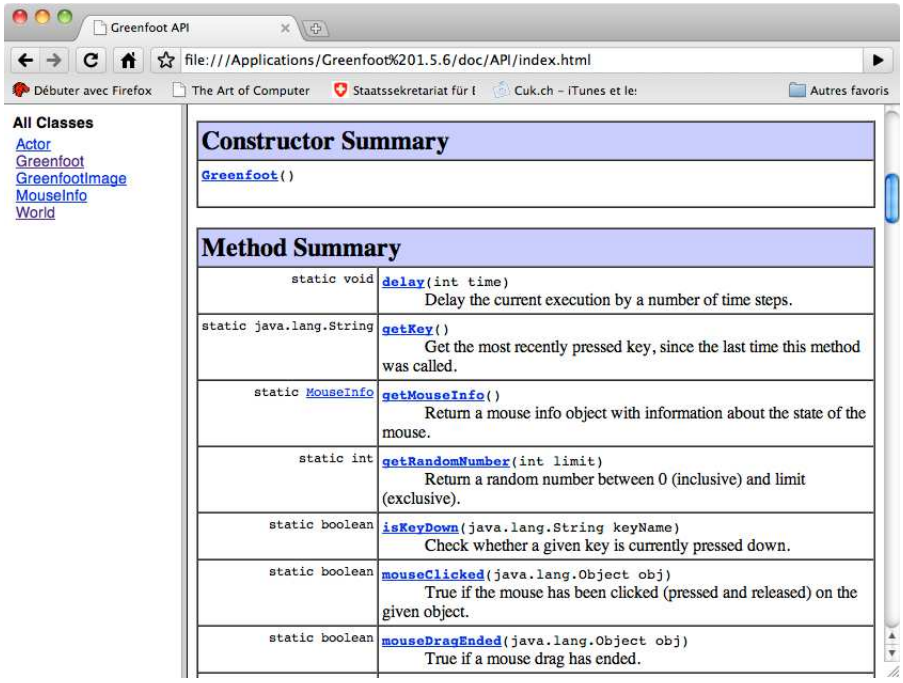
Vous êtes maintenant prêts à essayer votre jeu ! Placez un crabe, quelques vers et quelques homards dans votre monde et voyez si vous arrivez à manger tous les vers avant de vous faire prendre par un homard.

3.7 La fin du jeu

Vous avez sans doute constaté qu'après que le crabe a été mangé, les homards continuent à se déplacer. Nous pouvons modifier notre jeu de façon à ce que l'exécution du scénario s'arrête au moment où le crabe se fait manger. Greenfoot dispose d'une méthode pour cela ; il nous suffit de la trouver.

Pour trouver quelles méthodes sont disponibles dans Greenfoot, nous pouvons regarder dans la documentation des classes Greenfoot.

Dans l'environnement Greenfoot, choisissez Greenfoot Class Documentation depuis le menu Help. Cela fera apparaître la documentation de toutes les classes de Greenfoot dans un navigateur Web.



Cette documentation est aussi appelée l'*API de Greenfoot*, sigle pour *Application Programmers' Interface*. L'API nous présente toutes les classes disponibles et, pour chaque classe, toutes les méthodes qu'elles comprennent. Vous pouvez constater que Greenfoot met à disposition cinq classes : `Actor`, `Greenfoot`, `GreenfootImage`, `MouseInfo` et `World`.

Rappelons-nous que nous cherchions une méthode pour arrêter l'exécution d'un scénario. Elle se trouve dans la classe `Greenfoot`.

Exercice 3.21

Ouvrez l'API de Greenfoot dans votre navigateur. Sélectionnez la classe `Greenfoot`. Dans la documentation trouver le tableau "Method Summary" qui résume les caractéristiques des méthodes à disposition. Dans ce tableau, trouver la méthode qui stoppe l'exécution d'un scénario. Quel est le nom de cette méthode ?

Exercice 3.22

Faut-il passer un paramètre à cette méthode ? Quel est son type de valeur de retour ?

Vous pouvez accéder à la documentation d'une classe Greenfoot en cliquant le lien à son nom dans la liste de gauche de la page web ci-dessus. Pour chaque classe, le panneau principal du navigateur présente un commentaire général, le détails de ses constructeurs et une liste de ses méthodes. Nous parlerons des constructeurs ultérieurement.

Lorsque nous passons en revue les méthodes de la classe `Greenfoot`, nous trouvons une méthode qui s'appelle `stop`. C'est la méthode que nous pouvons employer pour arrêter l'exécution lorsque le crabe se fait attraper.

Pour utiliser cette méthode, nous écrirons simplement

```
Greenfoot.stop();
```

dans notre code source.

Exercice 3.23

Ajouter à votre scénario du code qui stoppe le jeu lorsqu'un homard mange le crabe. Vous devrez décider où ce code doit être ajouté. Trouver l'endroit de votre code qui est exécuté au moment où un homard mange le crabe et ajouter la ligne de code à cet endroit.

Nous utiliserons fréquemment la documentation des classes dans le futur pour trouver des informations à propos des méthodes que nous voudrons utiliser. Nous connaîtrons certaines méthodes par coeur au bout d'un moment, mais il y aura toujours de nouvelles méthodes que nous devrons chercher dans la documentation.

3.8 Inclure du son

Pour améliorer encore notre jeu, nous pouvons inclure quelques sons, ceci une fois de plus grâce à une méthode de la classe `Greenfoot`.

Exercice 3.24

Ouvrir la documentation des classes de Greenfoot. Dans la documentation de la classe `Greenfoot`, trouver les caractéristiques de la méthode qui permet de jouer un son. Quel est son nom ? Quels sont les paramètres attendus ?

Après avoir consulté la documentation, nous voyons que la classe `Greenfoot` dispose d'une méthode `playSound`. Le paramètre attendu par cette méthode est de type `String` ; il n'y a pas de valeur de retour.

Note

Il se peut que vous alliez jeter un coup d'oeil à la structure d'un scénario Greenfoot dans votre système de gestion de fichiers. Si vous regardez le dossier contenant les scénarios du livre, vous trouverez un dossier pour chaque scénario. Pour l'exemple du crabe, il y a différentes versions (*little-crab*, *little-crab-2*, *little-crab-3*, etc.). Dans chaque fichier de scénario, il y a plusieurs fichiers pour chaque

classe du scénario et d'autres fichiers auxiliaires. Il y a également deux dossiers supplémentaires : *images* contient les images du scénario et *sounds* les fichiers son.

Vous pouvez trouver dans ce dossier les sons à votre disposition et vous pouvez ajouter des fichiers pour avoir plus de "choix sonore".

En ce qui concerne notre scénario, il y a déjà deux fichiers son à disposition : leur nom est slurp.wav et au.wav. Nous pouvons facilement faire jouer ces sons par notre scénario en utilisant l'appel de méthode suivant :

```
Greenfoot.playSound("slurp.wav");
```

Exercice 3.25

Ajouter des sons au scénario : Lorsque le crabe mange un ver, faire jouer le son *slurp.wav*, et lorsqu'un homard mange le crabe, faire jouer *au.wav*. Choisir judicieusement les endroits où ajouter les deux lignes de code.

Le scénario little-crab-4 présente la solution à cet exercice. Il s'agit d'une version du projet qui comprend toutes les fonctionnalités dont nous avons discuté jusqu'ici : des vers, des homards, le contrôle à partir du clavier et le son.

À propos de son...

Vous pouvez également enregistrer vos propres sons. Les deux fichiers *slurp.wav* et *au.wav* ont été obtenus à l'aide du microphone intégré à un ordinateur. Il suffit d'utiliser l'un des nombreux programmes gratuits permettant l'enregistrement de son au format WAV, AIFF ou AU. (Audacity est un bon programme, mais il y en a de nombreux autres.)

Exercice 3.26

Si vous disposez d'un microphone intégré à votre machine, enregistrez vos propres sons à utiliser lorsque le crabe ou un ver se font manger. Enregistrez les sons, placez-les dans le dossier adéquat du dossier de scénario et utilisez-les dans votre code source.

3.9 Résumé des techniques de programmation

Dans ce chapitre, nous avons vu plusieurs utilisations possibles de l'instruction conditionnelle `if` ; cette fois-ci pour faire tourner notre crabe et le faire réagir à la pression des touches. Nous avons également vu comment appeler des méthodes d'une autre classe, à savoir les méthodes `getRandomNumber`, `isKeyDown` et `playSound` de la classe `Greenfoot`. Nous l'avons fait en utilisant la notations "point", avec le nom de la classe avant le point.

Dans l'ensemble, nous avons maintenant vu comment appeler des méthodes depuis trois endroits différents. Nous pouvons appeler des méthodes définies dans la classe courante elle-même, des méthodes définies dans une superclasse (*méthodes héritées*), et des méthodes statiques d'une autre classe. Cette dernière forme d'appel de méthode emploie la notation "point". Il nous reste à voir encore une forme d'appel de méthode : sur un autre objet ; nous rencontrerons ce genre d'appel un peu plus loin.

Il est également important de noter que nous avons vu comment lire la documentation de l'API d'une classe existante pour obtenir la liste des méthodes à disposition et comment les appeler.

4 Terminer le jeu du crabe

Dans ce chapitre, nous allons terminer le jeu du crabe. "Terminer" signifie ici qu'après cette discussion nous arrêterons de parler de ce projet dans ce texte. Un jeu n'est jamais terminé, bien sûr, vous pourrez toujours penser à certaines améliorations que vous aurez tout loisir d'ajouter par la suite. Nous suggérons quelques idées à la fin du chapitre. Nous allons maintenant exposer un certain nombre de ces améliorations possibles en détail.

4.1 Ajouter des objets automatiquement

Nous sommes à ce stade du développement proches d'un petit jeu avec lequel s'amuser un peu. Mais il y a encore un certain nombre de choses à faire. Le premier problème qui doit être résolu réside dans le fait que nous devons placer manuellement les acteurs dans le jeu : le crabe, les homards et les vers. Il nous faut faire en sorte que les acteurs soient présent à chaque début de partie.

Une chose se produit automatiquement à chaque fois que nous compilons notre projet sans erreur : le monde lui-même est créé. L'objet "monde", tel que nous le voyons à l'écran (la région carrée couleur sable) est une instance de la classe `CrabWorld`. Les instances de type "monde" sont traitées d'une manière spéciale par Greenfoot : tandis que nous devons créer des instances de nos acteurs nous-mêmes, le système Greenfoot crée toujours automatiquement une instance de notre classe monde et l'affiche à l'écran.

Voyons maintenant le code source de la classe `CrabWorld`. Si vous n'avez pas votre propre jeu du crabe à ce niveau, vous utiliserez little-crab-4 tout au long du chapitre.

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

public class CrabWorld extends World
{
    /**
     * Create the crab world (the beach). Our world has a size
     * of 560 x 560 cells, where every cell is just 1 pixel.
     */
    public CrabWorld()
    {
        super(560, 560, 1);
    }
}
```

Dans cette classe, nous voyons l'instruction `import` sur la première ligne. Nous parlerons de cette instruction en détail plus tard ; il nous suffit pour l'instant de savoir que cette ligne apparaît au début de chacune de nos classes.

On trouve ensuite l'en-tête de classe et un commentaire. Le début d'un commentaire est marqué par les symboles /** et sa fin est marquée par */. Vient ensuite la partie intéressante :

```
public CrabWorld()
{
    super(560, 560, 1);
}
```

Ceci s'appelle le *constructeur* de cette classe. un constructeur ressemble à une méthode, mais il y a quelques différences :

- Un constructeur n'a pas de type de valeur de retour spécifié entre le mot-clé "public" et le nom.
- Le nom du constructeur est toujours le même que le nom de la classe.

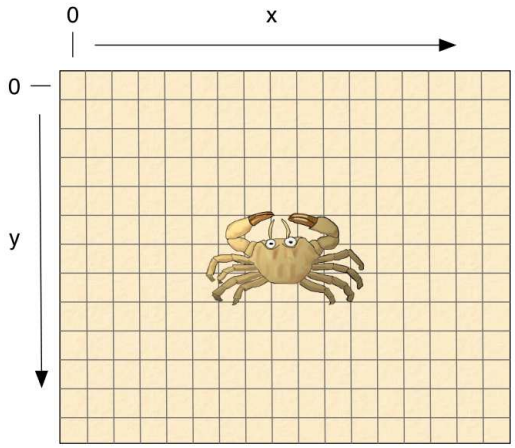
Un constructeur est une méthode particulière qui s'exécute automatiquement chaque fois qu'une instance de classe est créée. Il réalise alors ce qu'il faut faire pour mettre cette nouvelle instance dans l'état de départ désiré.

Dans notre cas, le constructeur définit la taille de notre monde (560 par 560 cellules) et une résolution (1 pixel par cellule). Nous discuterons de la définition de la résolution d'une classe de type "monde" plus en détail plus loin.

Vu que ce constructeur est appelé chaque fois qu'un monde est créé, nous pouvons l'employer pour instancier automatiquement nos acteurs. Si nous insérons le code de création d'un acteur dans le constructeur, ce code sera également exécuté. Par exemple, on pourrait écrire :

```
public CrabWorld()
{
     super(560, 560, 1);
     addObject(new Crab(), 150, 100);
}
```

Ce code va automatiquement créer un nouveau crabe et nous le placera à la position $x = 150$, $y = 100$ dans notre monde. La position $(150, 100)$ est le pixel qui se trouve à 150 pixels du bord gauche du monde, et 100 pixels du haut. L'origine de notre système de coordonnées, soit le point $(0; 0)$, se trouve en haut à gauche de notre monde.



Nous utilisons deux nouvelles choses ici : la méthode `addObject` et l'instruction `new` pour créer le crabe.

La méthode `addObject` est une méthode de la classe `World`. Nous pouvons le voir en ouvrant la documentation de cette classe dans Greenfoot. La documentation nous donne la signature de la méthode :

```
void addObject(Actor object, int x, int y)
```

La lecture de l'en-tête du début à la fin nous donne les informations suivantes :

- La méthode ne renvoie aucun résultat (le type de valeur de retour est `void`).
- Le nom de la méthode est `addObject`.
- La méthode a trois paramètres dont les noms sont `object`, `x` and `y`.
- Le type du premier paramètre est `Actor`, le type des deux autres est `int`

Cette méthode peut être utilisée pour ajouter un nouvel acteur dans le monde. Vu que cette méthode est une méthode de la classe `World` et que `CrabWorld` est une instance `World` et donc hérite de toutes les méthodes de `World`, elle est à disposition dans notre classe `CrabWorld` et nous pouvons tout simplement l'appeler.

4.2 Créer de nouveaux objets

La méthode `addObject` nous permet d'ajouter un objet acteur dans notre monde. Toutefois, pour pouvoir ajouter un objet, il faut qu'il ait été créé d'abord.

En Java, le mot clef `new` nous permet de créer de nouveaux objets à partir de classes existantes. Par exemple, l'expression

```
new Crab()
```

crée une nouvelle instance de la classe `Crab`. L'expression qui permet de créer de nouveaux objets commence toujours par le mot clef `new`, suivi du nom de la classe dont nous voulons une instance et d'une liste de paramètres qui est vide dans notre exemple. La liste de paramètres nous permet de passer des informations supplémentaires au constructeur. Dans la mesure où nous n'avons pas spécifié de constructeur particulier pour notre classe `Crab`, la liste de paramètres est ici vide.

Après avoir créé un nouvel objet, il faut faire quelque chose avec. Nous pouvons l'utiliser comme paramètre acteur de la méthode `addObject` pour ajouter cet objet au monde du crabe :

```
addObject(new Crab(), 150, 100)
```

Les deux paramètres restants donnent les coordonnées x et y de la position qui sera celle de notre objet dans le monde.

Exercice 4.1

Compléter le code du constructeur de la classe `CrabWorld` de votre projet de façon à ce qu'un crabe soit créé automatiquement, comme discuté ci-dessus.

Exercice 4.2

Ajouter du code de façon à ce que trois homards soient créés automatiquement dans le monde `CrabWorld`. Vous pouvez les placer à votre guise.

Exercice 4.3

Ajouter le code créant 10 vers de sable placés à différents endroits du monde `CrabWorld`.

Exercice 4.4

Déplacer tout le code qui génère les différents objets dans une méthode séparée, nommée `populateWorld`, dans la classe `CrabWorld`. Vous devrez déclarer la méthode `populateWorld` vous-même (elle n'attend aucun paramètre et ne renvoie rien du tout) et l'appeler depuis le constructeur. Compiler le code pour le tester.

Exercice 4.5

Utiliser des nombres aléatoires pour les coordonnées des vers. Vous pouvez le faire en remplaçant les valeurs de vos coordonnées par des appels à la méthode `getRandomNumber` de la classe `Greenfoot`.

Vous devriez disposer maintenant d'une version de votre "projet crabe" qui place automatiquement le crabe, trois homards et une dizaine de vers dans le monde chaque fois que vous compilez votre scénario. En cas de problème, consulter le scénario little-crab-5 des projets du livre qui contient le code dont il est question ci-dessus et les changements qui vont encore suivre d'ici à la fin de ce chapitre.

4.3 Animer des images

Nous allons maintenant discuter de la procédure à suivre pour animer l'image du crabe. Pour que le mouvement du crabe soit un peu plus réaliste, nous avons prévu de faire en sorte que le crabe bouge ses pattes et ses pinces en se déplaçant.

L'animation sera réalisée à l'aide d'un truc simple : Nous disposons de deux images différentes de notre crabe (appelés crab.png et crab2.png dans notre scénario), et nous passons de l'une à l'autre relativement rapidement.

Comme on peut le constater ci-dessous, la position des pattes et des pinces est légèrement différente d'une image à l'autre.

L'impression de mouvement sera donnée par le passage de l'une à l'autre de ces images. On aura bien l'illusion que le crabe bouge ses pattes et ses pinces.

Pour réaliser cela dans Greenfoot, nous devons introduire deux nouveaux concepts : les variables et les images dans Greenfoot.

4.4 Les images dans Greenfoot

Greenfoot met à disposition une classe appelée `GreenfootImage` qui va nous aider à utiliser et manipuler les images. Nous pouvons obtenir une image en construisant un nouvel objet de type `GreenfootImage` grâce au mot clef Java `new` ; il faudra donner en paramètre au constructeur le nom du fichier image. Par exemple, pour accéder à l'image crab2.png, nous écrirons

```
new GreenfootImage("crab2.png")
```

en gardant à l'esprit que le fichier dont nous donnons le nom en paramètre doit être présent dans le fichier images du scénario dans lequel nous travaillons.

Tout acteur Greenfoot vient avec une image. Par défaut, les acteurs héritent de l'image de leur classe. Nous assignons une image à la classe lorsque nous la créons et tout objet créé à partir de cette classe recevra, au moment de sa création, une copie de cette même image. Cela ne signifie pas pourtant que tous les objets d'une même classe devront toujours conserver la même image. Chaque acteur peut individuellement décider de changer son image en tout temps.

Exercice 4.6

Ouvrir la documentation de la classe `Actor`. Il y a deux méthodes qui permettent de changer l'image d'un acteur. Quel est leur nom et quels sont leurs paramètres ? Que retournent-elles ?

Si vous avez fait l'exercice ci-dessus, vous avez vu que l'une des méthodes permettant de changer l'image d'un acteur attend un paramètre de type `GreenfootImage`. C'est la méthode que nous utiliserons. Nous pouvons créer un objet de type `GreenfootImage` à partir d'un fichier d'image comme décrit ci-dessus, et ensuite utiliser la méthode `setImage` de l'acteur. Voici l'extrait de code qui permet de le faire :

```
setImage(new GreenfootImage("crab2.png"));
```

Notons que nous faisons deux choses différentes dans la même ligne de code : Nous appelons la méthode `setImage`, qui attend une image en paramètre :

```
setImage( image_quelconque );
```

Et à la place du texte "*image_quelconque*", nous écrivons

```
new GreenfootImage("crab2.png")
```

Cela nous permet de créer un objet image à partir du fichier crab2.png. Au moment de l'exécution de la ligne, la partie intérieure du code, qui permet la création de l'objet `GreenfootImage`, est

exécutée en premier. L'appel à la méthode `setImage` est ensuite exécuté et l'objet image que nous venons de créer lui est passé en paramètre.

En ce qui nous concerne, il est préférable de séparer la création de l'objet image et l'appel de la méthode `setImage`. La raison en est que nous voulons passer d'une image à l'autre un grand nombre de fois pendant les déplacements du crabe. Nous allons donc devoir appeler la méthode `setImage` très souvent, mais nous ne devrons créer nos deux images qu'une seule fois.

Ainsi, nous commencerons par créer les images et les stockerons dans deux objets ; nous utiliserons ensuite les objets en question pour les afficher alternativement sans devoir les recréer à nouveau.

Pour conserver les images dans notre objet `Crab`, nous devons employer un nouveau concept : la variable.

4.5 Variables d'instance (champs)

Nos acteurs devront souvent conserver de l'information. Les langages de programmation permettent de conserver de l'information en la stockant dans une *variable*.

Java permet l'utilisation de plusieurs sortes de variables. la première sorte que nous utiliserons ici s'appelle une *variable d'instance*, ou un *champ*. Ces deux termes sont synonymes. Nous verrons d'autre types de variables plus tard.

Une variable d'instance est un morceau de mémoire qui est attribué à l'objet, l'instance de la classe, d'où le nom. Tout ce qui y sera stocké sera conservé aussi longtemps que l'objet existera et on pourra y accéder en tout temps.

Une variable d'instance est déclarée dans une classe en écrivant le mot clef `private` suivi du type de la variable et du nom de la variable :

```
private type_de_la_variable nom_de_la_variable;
```

Le type de la variable définit ce que nous voulons y mettre. Dans notre cas, vu que nous voulons y stocker des objets de type `GreenfootImage`, le type doit précisément être `GreenfootImage`. Le nom de la variable nous permet de nous référer à celle-ci ultérieurement. Il devrait être représentatif de ce à quoi sert notre variable.

Voyons notre classe `Crab` en guise d'exemple.

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

// comment omitted

public class Crab extends Animal
{
    private GreenfootImage image1;
    private GreenfootImage image2;

    // methods omitted
}
```

Dans cet exemple, nous avons déclaré deux variables dans notre classe `Crab`. Les deux sont de type `GreenfootImage`, et elles s'appellent `image1` et `image2`.

Nous écrirons toujours les déclarations des variables d'instance au sommet de nos classes, avant les constructeurs et les méthodes. Java n'oblige pas à procéder ainsi, mais c'est une bonne façon de faire car elle nous permet de trouver facilement les déclarations de variables lorsqu'il nous est nécessaire de les trouver.

Exercice 4.7

Avant d'ajouter le code ci-dessus à la classe `Crab`, faites un clic droit sur un crabe de votre monde et sélectionnez inspect dans le menu contextuel du crabe. Prenez note de la liste des variables présentes dans ce menu.

Exercice 4.8

Expliquer pourquoi le crabe dispose de variables, malgré le fait que nous n'en avons pas déclaré dans notre classe `Crab`.

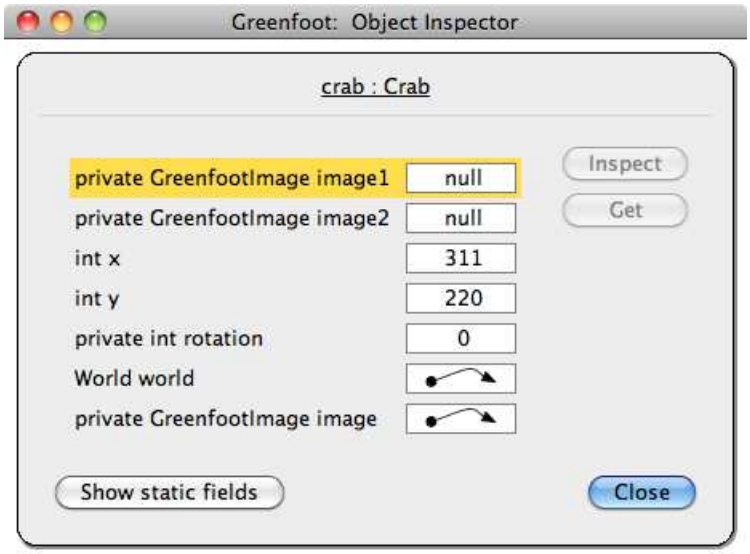
Exercice 4.9

Ajouter les déclarations de variable d'instance ci-dessus à votre class `Crab`. Prendre soin de compiler la classe sans erreur.

Exercice 4.10

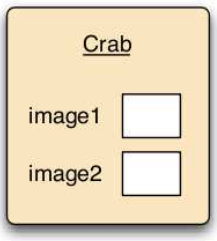
Après avoir ajouté les variables, inspecter à nouveau l'objet crabe. Prendre note de la liste des variables présentes et de leur valeur que l'on peut lire dans les boîtes blanches.

Notons que la déclaration de ces deux variables `GreenfootImage` ne nous donne pas deux objets `GreenfootImage`. Cela nous donne juste de l'espace vide pour y stocker nos deux objets, comme on peut le constater sur la copie d'écran ci-dessous :



D'un point de vue conceptuel, on peut se représenter les deux variables comme les étiquettes de deux espaces vides dans lesquels on pourra stocker deux objets images, comme sur la figure ci-dessous dans laquelle nos variables sont représentées comme deux rectangles blancs étiquetés

par les noms des variables :



Nous devons ensuite créer les deux objets images et les stocker dans les variables. Nous avons déjà montré le code de création des objets ci-dessus :

```
new GreenfootImage("crab2.png")
```

Pour stocker l'objet dans la variable, nous avons besoin du concept d'*affectation*.

4.6 Affectation

Une affectation est une instruction qui nous permet de stocker quelque chose dans une variable. On utilise le signe égal pour l'écrire :

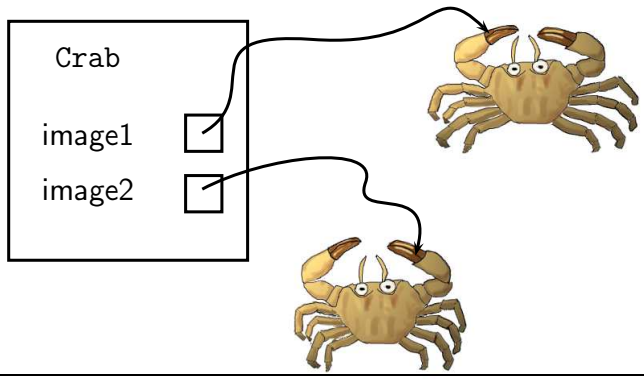
```
variable = expression;
```

À la gauche du signe égal se trouve le nom de la variable dans laquelle nous voulons stocker quelque chose et à droite se situe ce que nous désirons stocker. Comme le signe égal est utilisé pour l'affection, on lui donne également le nom de *symbole d'affectation*. On dira que l'on *affecte* une valeur à la variable. Il ne faut pas confondre l'utilisation du signe égal en programmation comme symbole d'affectation avec l'utilisation que l'on en fait en mathématiques du même signe égal.

Dans le code source de la classe `Crab`, on écrit les deux lignes suivantes :

```
image1 = new GreenfootImage("crab.png");
image2 = new GreenfootImage("crab2.png");
```

Ces deux lignes de code vont créer les deux images que nous désirons utiliser et les stocker dans nos deux variables `image1` et `image2`. Après ces deux instructions, nous disposons de trois objets (un crabe et deux images), et les variables du crabe contiennent des références aux objets. On peut illustrer la situation comme ceci :



La question qui se pose maintenant concernant ces images est de savoir où nous allons mettre le code qui permet la création de ces images et leur stockage dans les variables. Vu que cette opération ne doit être réalisée qu'une seule fois lorsque l'objet crabe est créé, et non chaque fois que la méthode `act` est appelée, nous ne pouvons pas mettre ce code dans la méthode `act`. Nous mettrons plutôt ce code dans un constructeur.

4.7 Utiliser les constructeurs des acteurs

Au début de ce chapitre, nous avons vu comment utiliser le constructeur de la classe `World` pour initialiser le monde du crabe. Nous pouvons utiliser de manière analogue le constructeur d'une classe acteur pour initialiser l'acteur lui-même. Le code qui se trouve dans le constructeur est exécuté une fois au moment de la création de l'acteur. le code ci-dessous montre un constructeur pour la classe `Crab` qui initialise les deux variables d'instance en créant les objets images et en les assignant aux variables.

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

// comment ommitted

public class Crab extends Animal
{
    private GreenfootImage image1;
    private GreenfootImage image2;

    /**
     * Create a crab and initialize its two images.
     */
    public Crab()
    {
        image1 = new GreenfootImage("crab.png");
        image2 = new GreenfootImage("crab2.png");
        setImage(image1);
    }

        // methods omitted
}
```

Les règles valant pour le constructeur de la classe `World` s'appliquent également au constructeur de la classe `Crab` :

- L'en tête d'un constructeur n'inclus pas de type de valeur de retour.
- Le nom du constructeur est le même que le nom de la classe.
- Le constructeur est exécuté automatiquement lorsqu'un crabe est créé.

La dernière règle stipulant que le constructeur est automatiquement exécuté garantit que les objets images sont systématiquement générés et assignés aux deux variables images lorsque nous créons un crabe. Ainsi, après avoir créé le crabe, nous aurons une situation correspondant à la figure de la page 47.

La dernière ligne du constructeur de la classe `Crab` assigne la première des deux images à l'objet crabe qui vient d'être créé :

```
setImage(image1);
```

Ceci nous montre comment le nom de la variable, ici image1 peut être utilisé maintenant pour se référer à l'objet image qui s'y trouve stocké.

Exercice 4.11

Ajouter ce constructeur à votre classe `Crab`. Vous ne constaterez aucun changement au comportement du crabe, mais la classe devrait pouvoir être compilée sans erreur et vous devriez pouvoir créer des crabes.

Exercice 4.12

Inspecter à nouveau un objet crabe à l'aide du menu contextuel. Noter les noms des variables de cet objet ainsi que leur valeur. Comparer avec les notes précédentes.

4.8 Alterner entre les deux images

Nous disposons maintenant d'un objet crabe contenant deux images qui nous permettront de réaliser notre animation, mais nous n'avons pas encore codé l'animation proprement dite. C'est relativement facile à réaliser.

Pour animer le crabe, nous devons alterner entre nos deux images. En d'autres termes, à chaque étape, si nous sommes en train de montrer l'image1, nous voulons passer à l'image2 et vice versa. On peut l'exprimer avec du pseudo-code, comme ci-dessous :

```
if (notre image courante est image1) then
    utiliser image2 maintenant
else
    utiliser image1 maintenant
```

Le pseudo-code que nous utilisons ici permet d'exprimer une tâche sous la forme d'un code qui ressemble à du code Java, mais qui contient également du langage courant. Cela peut souvent nous aider à écrire notre code Java final. Voici maintenant le code Java correspondant au pseudo-code ci-dessus :

```
if ( getImage() == image1 )
{
    setImage(image2);
}
else
{
    setImage(image1);
}
```

Dans cet extrait de code, nous pouvons observer plusieurs éléments nouveaux :

- La méthode `getImage` peut être utilisée pour obtenir l'image courante de l'acteur.
- L'opérateur `==` formé de deux signes égal peut être utilisé pour comparer deux valeurs entre elles. Le résultat est ou bien `true` ou bien `false`.
- L'instruction conditionnelle `if` a une extension que nous n'avions pas encore vue. Pour construire l'instruction conditionnelle étendue, on ajoute le mot clef `else` après le premier bloc et on fait suivre ce mot d'un deuxième bloc d'instructions. Nous étudierons l'instruction conditionnelle `if` en détail dans le paragraphe suivant.

Piège

C'est une erreur courante que de mélanger l'opérateur d'affectation (=) avec l'opérateur de comparaison (==). Pour vérifier si deux variables sont égales, il faudra toujours écrire deux signes égal.

4.9 L'instruction conditionnelle `if/else`

Avant de continuer dans la programmation de notre jeu, nous allons étudier un peu plus en détail l'instruction conditionnelle `if`. Comme nous venons de le voir, la conditionnelle peut s'écrire sous la forme suivante :

```
if ( condition )
{
    instruction;
    instruction;
    ...
}
else
{
    instruction;
    instruction;
    ...
}
```

La conditionnelle ci-dessus contient deux blocs (paires d'accolades contenant une liste d'instructions) : la clause `if` et la clause `else`, *dans cet ordre*.

Lors de l'exécution de cette conditionnelle, la condition est évaluée en premier. Si la condition est vraie, la clause `if` est exécutée et le programme continue à partir de ce qui suit immédiatement la clause `else`. Si la condition est fausse, alors la clause `if` n'est pas exécutée, et le programme continue à partir de la clause `else`. Ainsi, l'un parmi les deux blocs d'instructions est toujours exécuté, mais jamais les deux en même temps.

La partie "`else`" de l'instruction est optionnelle et lorsqu'on l'omet, on retrouve l'instruction conditionnelle `if` que nous avions utilisée plus haut.

Nous avons maintenant tous les outils qu'il nous faut pour terminer notre tâche.

Exercice 4.13

Ajouter le code qui permet de passer d'une image à l'autre à méthode `act` de la classe `Crab`. Compiler

le code source de la classe et éliminer les erreurs le cas échéant. Tester l'exécution en cliquant le bouton Act plutôt que le bouton Run, ce qui permet d'observer le nouveau comportement du crabe plus clairement.

Exercice 4.14

Dans le chapitre 3, nous avons discuté de l'utilité d'employer des méthodes séparées pour accomplir des sous-tâches plutôt que d'écrire le code directement dans la méthode `act`. Mettre cette idée en pratique : Créer une nouvelle méthode appelée `switchImage`, déplacer le code source concerné dans le corps de cette méthode et appeler `switchImage` depuis la méthode `act`.

Exercice 4.15

Appeler la méthode `switchImage` à partir du menu contextuel du crabe. Cela fonctionne-t-il ?

4.10 Compter les vers

Nous allons finalement discuter du décompte des vers mangés par notre crabe. Nous voulons ajouter à notre code ce qu'il faut pour le crabe puisse compter le nombre de vers qu'il a mangé ; nous aimerions de plus que le jeu soit gagné dès que le crabe a mangé huit vers. Nous voulons encore que l'ordinateur joue le "son de la victoire" lorsque le crabe mange son huitième ver.

Pour arriver à nos fins, nous devons ajouter plusieurs éléments au code de notre crabe :

- une variable d'instance pour stocker le nombre courant de vers mangés ;
- une affectation qui initialise la variable ci dessus à 0 au moment de la création du crabe ;
- du code qui incrémente notre compteur à chaque fois qu'un ver est mangé ;
- et finalement, du code qui teste si nous avons mangé huit vers, qui stoppe le jeu et qui joue le "son de la victoire" dès que le test est positif.

Réalisons dans l'ordre les différentes tâches ci-dessus.

Nous pouvons définir une nouvelle variable d'instance en suivant la procédure expliquée dans le paragraphe 4.5. Immédiatement après les deux définitions de variables d'instance qui s'y trouvent déjà, on ajoute la ligne de code suivante :

```
private int wormsEaten;
```

Le mot `private` est utilisé au début de toutes nos définitions de variables d'instance. Les deux mots qui suivent sont le type et le nom de notre variable. Le type `int` indique ici que nous voulons stocker un entier dans cette variable, et le nom `wormsEaten` donne une idée de ce à quoi servira la variable.

Nous ajoutons ensuite la ligne suivante à la fin de notre constructeur :

```
wormsEaten = 0;
```

Cette ligne permet d'initialiser la variable `wormsEaten` à 0 lorsque le crabe est créé. À strictement parler, cette instruction est redondante car les variables d'instances de type `int` sont

initialisées à 0 automatiquement. Toutefois, nous voudrons parfois que la valeur initiale d'une variable prenne une autre valeur que 0 ; il est donc recommandé d'écrire soi-même l'initialisation d'une telle variable d'instance.

Il nous reste à compter les vers et tester si nous avons atteint le nombre de huit. Nous devrons le faire chaque fois que nous mangeons un ver, ce qui fait que nous allons nous placer dans le code de la méthode `lookForWorm`, là où se trouvent les lignes qui permettent de manger un ver. Nous ajoutons ici la ligne qui incrémente le nombre de vers :

```
wormsEaten = wormsEaten + 1;
```

Dans l'affectation ci-dessus, la partie qui est à droite du signe égal est évaluée en premier (`wormsEaten + 1`) : la valeur de `wormsEaten` est lue et on lui ajoute 1. Le résultat est ensuite affecté à la variable `wormsEaten`. En fin de compte, la valeur de la variable sera augmentée de 1.

Il nous faut maintenant une instruction `if` qui va tester si nous avons atteint le compte des huit vers, et qui joue le "son de la victoire" et stoppe l'exécution si c'est le cas. Le code ci-dessous montre la méthode `lookForWorm` complète. Le fichier son fanfare.wav utilisé ici se trouve dans le dossier sounds de notre scénario, il suffit donc le faire jouer par Greenfoot.

```
/**
 * Check whether we have stumbled upon a worm.
 * If we have, eat it. If not, do nothing. If we have
 * eaten eight worms, we win.
 */
public void lookForWorm()
{
    if ( canSee(Worm.class) )
    {
        eat(Worm.class);
        Greenfoot.playSound("slurp.wav");

        wormsEaten = wormsEaten + 1;
        if (wormsEaten == 8)
        {
            Greenfoot.playSound("fanfare.wav");
            Greenfoot.stop();
        }
    }
}
```

Exercice 4.16

Ajouter le code ci-dessus à votre scénario. Tester et faire en sorte que tout fonctionne.

Exercice 4.17

Pour être vraiment sûr que tout marche bien, ouvrir un "inspecteur" de votre crabe à l'aide du menu

contextuel de l'objet avant de commencer à jouer. Laisser la fenêtre de l'inspecteur ouverte durant la partie et garder la variable `wormsEaten` à l'oeil durant le jeu.

4.11 Quelques idées supplémentaires

Le scénario little-crab-5, qui se trouve dans le dossier des scénarios du livre, montre une version du projet qui inclut toutes les extensions discutées ci-dessus.

Nous allons laisser ce scénario de côté maintenant et passer à un autre exemple, malgré le fait qu'il y encore beaucoup d'améliorations évidentes à faire à ce scénario (et encore plus d'améliorations moins évidentes ...). Voici quelques idées :

- utiliser d'autres images pour le fond et les acteurs ;
- ajouter d'autres sortes d'acteurs ;
- ne faire bouger le crabe que lorsque la "flèche haut" est pressée ;
- construire un jeu à deux joueurs en introduisant une autre classe contrôlée par le clavier qui réagit à d'autre touches du clavier ;
- faire apparaître d'autres vers lorsqu'un ver est mangé ou à un moment pris au hasard ;
- et tout ce à quoi vous penserez par vous-même.

Exercice 4.18

L'image de notre crabe change très rapidement lorsqu'il se déplace, ce qui donne l'impression qu'il est un peu hyperactif. Cela donnerait sans doute meilleure impression si l'image du crabe ne changeait que tous les deux ou trois cycles de la méthode `act`. Essayer d'implémenter ce comportement. Pour le faire on peut ajouter un compteur qui est incrémenté dans la méthode `act`. Chaque fois que la valeur de ce compteur atteint 2 ou 3, l'image change et le compteur est remis à zéro.

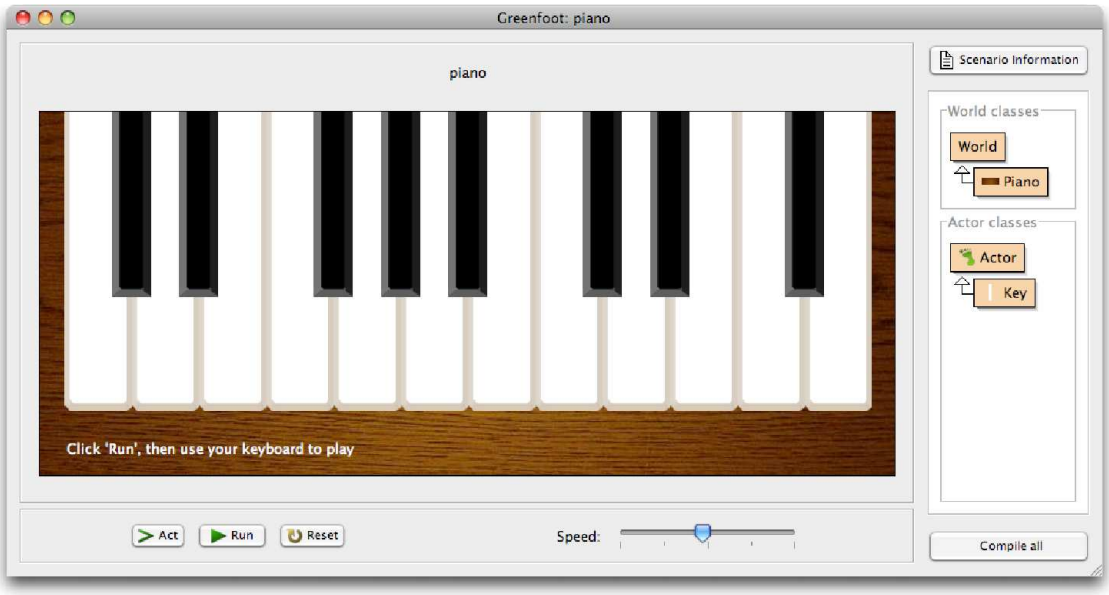
4.12 Résumé des techniques de programmation

Dans ce chapitre, nous avons abordé un certain nombre de nouveaux concepts de programmation. Nous avons vu comment les constructeurs peuvent être utilisés pour initialiser des objets ; les constructeurs sont toujours exécutés lorsqu'un nouvel objet est créé. Nous avons vu comment utiliser les variables d'instance ou *champs*, comment écrire les instructions d'affectation pour stocker l'information et comment récupérer cette information plus tard. Nous avons utilisé l'instruction `new` pour créer de nouveaux objets dans le cadre de nos programmes et nous avons finalement vu la forme complète de l'instruction conditionnelle `if`, qui comporte une partie `else`, exécutée lorsque la condition testée par le `if` est fausse.

En combinant toutes ces techniques, nous pouvons déjà écrire de nombreuses lignes de code.

5 Faire de la musique : un piano à l'écran

Dans ce chapitre, nous allons commencer un nouveau scénario : un simulateur de piano qui apparaîtra à l'écran et dont nous pourrons jouer à l'aide des touches du clavier. On peut voir ci-dessous une image montrant le piano terminé :



Nous commençons à nouveau en ouvrant un scénario du livre : piano-1. Il s'agit d'une version du scénario qui contient les ressources dont nous aurons besoin (images et fichiers son), mais pas grand chose d'autre. Nous utiliserons ces éléments simples pour commencer à écrire le code qui nous permettra de construire notre piano.

Exercice 5.1

Ouvrir le scénario *piano-1* et examiner le code source des deux classes existantes, `Piano` et `Key`. Faire en sorte d'avoir compris ce que fait le code qui y est déjà écrit.

Exercice 5.2

Créer un objet de la classe `Key` et le placer dans le monde. En créer plusieurs et les placer côte à côte.

5.1 Animation de la touche

Après avoir examiné le code source existant, vous avez sans doute constaté qu'il ne contient pas grand chose pour le moment : La classe `Piano` ne fait que spécifier la taille et la résolution du monde, et la classe `Key` ne contient que le squelette du constructeur et de la méthode `act` ; il n'y a que l'en-tête et un bloc vide qui suit.

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

public class Key extends Actor
{
    /**
     * Create a new key.
     */
    public Key()
    {
    }

    /**
     * Do the action for this key.
     */
    public void act()
    {
    }
}
```

Nous pouvons commencer à expérimenter ce scénario en créant un objet de la classe `Key` et en le plaçant dans le monde, comme dans l'exercice plus haut. Vous avez constaté sans doute que son image est celle d'une simple touche blanche et que cet acteur ne fait rien lorsqu'on clique sur le bouton Run.

Notre première tâche sera de programmer l'animation de la touche de piano : Nous aimerions que, lorsque nous pressons sur une touche du clavier, la touche dessinée à l'écran change d'aspect de sorte à avoir l'impression que quelqu'un a appuyé sur cette touche. Tel quel, le scénario contient déjà deux fichiers d'image nommée white-key.png et white-key-down.png, que nous pouvons employer pour obtenir l'effet désiré. Il contient également deux fichiers image en plus, black-key.png et black-key-down.png, que nous utiliserons plus tard pour les touches noires. L'image que nous voyons lorsque nous créons une touche est l'image white-key.png.

Nous pouvons très facilement donner l'impression que l'on appuie sur la touche en changeant d'image lorsqu'une touche spécifique du clavier subit une pression. Voici un premier essai :

```
public void act()
{
    if ( Greenfoot.isKeyDown("g") )
    {
        setImage ("white-key-down.png");
    }
    else
    {
        setImage ("white-key.png");
    }
}
```

Dans ce code, nous avons choisi une touche arbitraire du clavier de l'ordinateur (la touche "g") qui provoque la pression sur le "piano" à l'écran. Ce choix n'est pas important pour l'instant ; nous attribuerons plus tard une touche du clavier de l'ordinateur différente à chaque touche du simulateur de piano. Ce qui compte ici est que nous montrons l'image "down" lorsque la touche du clavier reçoit une pression et l'image "up" lorsque rien ne se passe.

Exercice 5.3

Implémenter cette version de la méthode `act` dans votre propre scénario. La tester et faire en sorte d'obtenir l'effet désiré.

Malgré le fait que cette version fonctionne, il y a un problème : L'image n'est pas modifiée une seule fois au moment ou elle change, mais continuellement. Chaque fois que la méthode `act` s'exécute, la méthode `setImage` change l'image courante, même si c'est déjà la bonne. Par exemple, si la touche "g" ne subit pas de pression, l'image sera changée en white-key.png, même si c'était déjà l'image affichée.

Ce problème peut paraître mineur à ce stade. Après tout, changer d'image lorsque ce n'est pas nécessaire est juste inutile, mais ce n'est pas faux de le faire. Malgré cela, plusieurs raisons font que nous désirons régler ce problème. Une des raisons vient du fait qu'il n'est pas bon de gaspiller les ressources du processeur en lui donnant du travail inutile. Une autre raison est que lorsque nous ajouterons le son à l'image ce détail deviendra crucial : En effet, lorsque nous appuyons sur une touche du clavier, cela fait une grosse différence si l'on entend le son une fois ou encore et encore à partir de ce moment.

Améliorons donc notre code en faisant en sorte que l'image ne change que lorsque c'est vraiment nécessaire. Pour le faire, nous ajoutons un champ booléen à notre classe pour conserver l'état courant de notre touche de clavier : enfoncée ou non. Nous appelons ce champ `isDown` et le déclarons comme suit :

```
private boolean isDown;
```

Nous mettrons la valeur `true` dans cette variable si l'utilisateur est en train d'appuyer sur la touche et `false` sinon.

Nous pouvons alors savoir si la touche de notre clavier vient de subir une pression : Si notre champ `isDown` a la valeur `false`, mais que la touche “g” est enfoncée, elle doit avoir reçu la pression un tout petit peu auparavant. Inversément, si notre champ `isDown` a la valeur `true` (nous pensons que la touche est enfoncée), mais que la touche “g” du clavier n’est pas enfoncée, elle doit juste avoir été libérée. Nous ne devons donc changer les illustrations que dans ces deux situations. Nous mettons en pratique cette idée dans l’extrait de code ci-dessous :

```
public void act()
{
        if ( !isDown && Greenfoot.isKeyDown(“g”) )
        {
                setImage (“white-key-down.png”);
                isDown = true;
        }
        if ( isDown && !Greenfoot.isKeyDown(“g”) )
        {
                setImage (“white-key.png”);
                isDown = false;
        }

}
```

Dans les deux cas, nous prenons garde à changer l’état de la variable `isDown` chaque fois qu’un changement est détecté.

Dans l’extrait de code ci-dessus, on remarque l’utilisation de deux nouveaux symboles : le point d’exclamation (`!`) et le double esperluette ou double “et commercial” (`&&`).

Les deux symboles en question sont des opérateurs logiques. Le point d’exclamation signifie NON, tandis que le double esperluette veut dire ET.

Ainsi, les lignes de la méthode `act`

```
if ( !isDown && Greenfoot.isKeyDown(“g”) )
{
    setImage (“white-key-down.png”);
    isDown = true;
}
```

Peuvent s’écrire moins formellement au moyen de pseudo-code :

```
if ( (not isDown) and Greenfoot.isKeyDown(“g”) ) ...
```

Le même code pourrait encore être décrit en “français”

```
if ( (la-touche-du-piano-n'est-pas-enfoncée-maintenant)
    et (la-touche-du-piano-subit-une-pression) )
{
    changer l'image pour montrer la touche enfoncée;
    prendre maintenant note du fait que la touche est enfoncée;
}
```

Reprenez maintenant le code source de la méthode `act` tel qu'il est écrit ci-dessus en java et ne continuez la lecture que lorsque vous l'avez entièrement compris.

Exercice 5.4

Faire une liste de tous les opérateurs logiques du langage Java.

Exercice 5.5

Implémenter la nouvelle version de la méthode `act` dans votre propre scénario. Vérifier que la méthode fonctionne comme auparavant, sans changement visible. Ne pas oublier de rajouter le champ booléen `isDown` au début de votre classe.

5.2 Ajouter le son

La prochaine chose à faire est de faire en sorte que la pression sur la touche fasse jouer un son à notre machine. Pour le faire, nous ajoutons une nouvelle méthode à notre classe `Key`, appelée `play`. Nous pouvons ajouter cette méthode à l'aide de notre éditeur de texte, directement sous la méthode `act`. Nous commençons par écrire le commentaire, l'en-tête de la méthode et un bloc vide pour la nouvelle méthode :

```
/**
 * Play the note of this key
 */
public void play()
{

}
```

Malgré le fait que ce code ne provoque aucune action (le corps de la méthode est vide), il ne devrait pas poser de problème à la compilation.

L'implémentation de la méthode est plutôt simple : Nous voulons juste faire jouer un son à partir d'un seul fichier musical. Le scénario piano-1, que vous avez utilisé pour commencer ce projet, dispose d'une collection de sons déjà inclus dans le dossier sounds, et chacun de ces fichiers contient le son d'une touche du piano. Les noms de ces fichiers sont 2a.wav, 2b.wav, 2c.wav, 2c#.wav, 2d.wav, 2d#.wav, 2e.wav, et ainsi de suite. On va prendre l'un de ces fichiers au hasard, disons 3a.wav, pour notre touche d'essai.

Pour faire jouer cette note par notre machine, nous pouvons utiliser la méthode `playSound` de la classe `Greenfoot` à nouveau :

```
Greenfoot.playsound(\textrm{\}3a.wav\textrm{"});
```

C'est là tout ce qu'il nous faut comme code source pour faire jouer le son. Voici l'implémentation de la méthode `play` :

```
/**
 * Play the note of this key.
 */
public void play()
{
        Greenfoot.playSound("3a.wav");
}
```

Exercice 5.6

Implémenter la méthode play dans votre scénario. Compiler sans erreur.

Exercice 5.7

Tester votre méthode. Vous pouvez le faire en créant un objet de la classe `Key` et en invoquant ensuite la méthode `play()` depuis le menu contextuel.

Nous y sommes presque, maintenant. Nous pouvons produire le son d'une touche en invoquant la méthode `play`, et nous pouvons exécuter notre scénario et appuyer sur la touche "g" pour donner l'impression que la touche affichée par le simulateur s'enfonce.

Nous n'avons plus qu'à faire jouer le son lorsque l'utilisateur appuie sur la touche de son clavier. Il suffit pour cela d'écrire la ligne suivante dans le code source de notre classe `Key` :

```
play();
```

Exercice 5.8

Ajouter à la classe `Key` le code qu'il faut au bon endroit pour que la note soit jouée par le simulateur lorsque la touche associée subit une pression. Pour pouvoir le faire, il faudra d'abord trouver l'endroit où il faut ajouter l'invocation de la méthode `play`.

Exercice 5.9

Que se passe-t-il lorsque l'on crée deux touches, que l'on lance l'exécution du scénario et que l'on appuie sur la touche "g" ? Que faudrait-il faire pour faire réagir chaque touche du piano à une autre touche du clavier ?

Tous les changements dont nous avons discuté ici sont à disposition dans le scénario piano-2 qui se trouve dans le dossier des scénarios du livre. Si vous n'avez pas réussi à résoudre tous les problèmes rencontrés ou que vous désirez simplement comparer votre solution avec la notre, vous pouvez sans autre consulter cette version.

5.3 Créer plusieurs touches, un effort d'abstraction

Nous avons atteint le stade suivant : Nous pouvons créer une touche de piano qui réagit à une touche du clavier de notre ordinateur. Le problème auquel nous devons faire face maintenant est clairement le suivant : Lorsque nous créons plusieurs touches, elles réagissent toutes à la même touche du clavier et produisent toutes la même note. Nous voulons changer cela.

L'obstacle auquel nous sommes confrontés vient du fait que nous avons "codé en dur", de l'anglais *hard-coded*, le nom de la touche clavier ("g") et le nom du fichier son ("3a.wav") dans le code source de notre classe. Cela signifie que nous utilisons ces noms directement, sans plus avoir la possibilité de les changer, à moins de modifier le code source à nouveau et de le recompiler.

Lorsque l'on programme un ordinateur, il est bien et bon d'écrire du code qui permet de résoudre une tâche spécifique, comme par exemple calculer la racine carrée de 1764 ou jouer un la à 220 Hz, mais cela n'est pas très utile. Nous chercherons plutôt en général à écrire du code qui permet de résoudre toute une *famille* de problèmes ; par exemple, trouver la racine d'un nombre quelconque ou jouer toute une série de notes à l'aide d'un simulateur de piano. Nos programmes peuvent ainsi devenir vraiment efficaces.

Pour atteindre ce but, nous utilisons une technique qui s'appelle l'*abstraction*. L'abstraction se présente en informatique sous bien des formes ; ce que nous allons réaliser maintenant en est un exemple d'application. Nous allons utiliser l'abstraction pour transformer notre classe `Key`, qui permet de créer des objets jouant un la à 220 Hz lorsque l'on presse sur la touche "g", en une classe qui peut nous fournir des objets permettant de jouer toute une série de notes, tout ceci commandé par différentes touches du clavier.

L'idée de base permettant d'atteindre le but fixé est d'utiliser une variable pour le nom de la touche du clavier à laquelle nous voulons réagir, et une autre variable pour le nom du fichier son que nous voulons faire jouer par la machine.

```
public class Key extends Actor
{
        private boolean isDown;
        private String key;
        private String sound;

        /**
          * Create a new key linked to a given keyboard key,
          * and with a given sound.
          */
        public Key(String keyName, String soundFile)
        {
                key = keyName;
                sound = soundFile;
        }

        // Methods omitted.
}
```

Le code source ci-dessus montre un début de solution. Nous définissons ici deux variables d'instance supplémentaires, `key` et `sound` pour permettre le stockage du nom de la touche clavier et du fichier son que nous désirons utiliser. Nous ajoutons également deux paramètres au constructeur, de façon à ce que les deux éléments d'information puissent être fournis au moment où l'objet touche est créé ; dans le corps du constructeur, nous ajoutons les instructions qui attribuent à nos champs les valeurs des deux paramètres.

Nous avons maintenant à disposition une *abstraction* de notre classe `Key` initiale. À partir de maintenant, lorsque nous créons un nouvel objet `Key`, nous pouvons spécifier à quelle touche du clavier il doit réagir et quel son la machine doit alors jouer. Bien sûr, nous n'avons pas encore écrit le code qui utilise effectivement ces variables ; cela reste à faire.

Nous laisserons ceci en exercice pour vous.

Exercice 5.10

Implémenter les changements discutés ci-dessus. Il faut ajouter les champs concernant la touche clavier et le fichier son, et ajouter le constructeur à deux paramètre pour initialiser les champs.

Exercice 5.11

Modifier votre code de façon à ce que votre objet "touche de piano" réagisse à la bonne touche clavier et joue le son spécifié lors de sa création. Tester en créant plusieurs touches liées à différents sons.

Nous avons maintenant à disposition un scénario qui nous permet de créer un ensemble de touches jouant une série de notes. Nous n'avons pour l'instant que des touches blanches, mais nous pouvons déjà créer un demi-piano avec ces touches. Cette version du projet se trouve parmi les scénarios du livre, sous le nom de piano-3.

Le fait est qu'il est ennuyeux de devoir construire toutes les touches à la main. Nous devons, pour l'instant, créer les touches les unes après les autres, en donnant tous les paramètres. Pire encore : chaque fois que nous faisons un changement dans le code source, nous devons tout recommencer. Il est temps d'écrire du code source qui va créer les touches du piano à notre place.

5.4 Construire le piano

Nous allons maintenant écrire dans la classe `Piano` un peu de code qui crée et place les touches du piano à notre place. Ajouter une touche, ou encore quelques touches, est chose aisée : il suffit d'ajouter la ligne suivante au constructeur de `Piano` et une touche est créée et placée dans le monde chaque fois que nous cliquons la touche Reset de l'environnement Greenfoot.

```
addObject ( new Key ( "g", "3a.wav"), 300, 180 );
```

Souvenez-vous du fait suivant : l'expression

```
new Key ( "g", "3a.wav")
```

crée une nouvelle instance de la classe `Key` avec les paramètres choisis.

Par contre, l'instruction

```
addObject ( un-objet-quelconque, 300, 180 );
```

ajoute l'objet dans le monde, à l'emplacement donné par les coordonnées x et y. Le choix $x = 300$ et $y = 180$ est ici arbitraire.

Exercice 5.12

Ajouter à la classe `Piano` le code permettant la création et l'affichage d'une touche de piano au démarrage du scénario.

Exercice 5.13

Changer la coordonnée y dans l'instruction `addObject (...);` de façon à ce que le haut de la touche se situe exactement au sommet de l'image de fond. Indice : l'image de la touche mesure 280 pixels de haut par 63 pixels de large.

Exercice 5.14

Ecrire le code source qui permet de créer une deuxième touche de piano qui joue un sol à 195.998 Hz (fichier son 3g.wav lorsque la touche "f" subit une pression. Placer cette touche à gauche de la première sans recouvrement ni espace entre les deux. Elles seront bien entendu à la même hauteur.

Plus haut dans ce texte, nous avons vu l'intérêt d'utiliser des méthodes séparées pour effectuer des tâches séparées. Créer toutes les touches du piano est une tâche logiquement séparée du reste ; plaçons donc le code source chargé de cette tâche dans une méthode séparée. Le résultat sera exactement le même, mais le code sera plus facile à lire.

Exercice 5.15

Dans la classe `Piano`, créer une nouvelle méthode appelée `makeKeys()`. Déplacer le code qui gère la création des touches dans le corps de cette méthode. Appeler cette méthode depuis le constructeur de la classe `Piano`. Prendre soin d'écrire un commentaire pour cette nouvelle méthode.

Nous pourrions maintenant ajouter toute une liste d'instructions `addObject (...);` dans le but de créer toutes les touches dont nous avons besoin pour le clavier de notre piano. Ce n'est pourtant pas la meilleure façon d'atteindre notre but.

5.5 Utiliser des boucles: la boucle `while`

Les langages de programmation offrent des instructions spécifiques pour effectuer une tâche répétitive un grand nombre de fois : les *boucles*.

Une boucle est une structure de contrôle du programme qui nous permet d'écrire de manière concise des commandes telles que *"Faire ceci 20 fois"* ou *"Appeler ces deux méthodes 3 millions de fois"* facilement (sans devoir écrire 3 millions de lignes de code). Le langage Java dispose de plusieurs sortes de boucles.

Nous allons étudier plus en détail la *boucle while* dans ce paragraphe.

Une boucle `while` s'écrit de la façon suivante :

```
while ( condition )
   {
       instruction;
       instruction;
       ...
   }
```

Le mot clef Java `while` est suivi par une condition entre parenthèses et un bloc (une paire d'accolades) contenant une ou plusieurs instructions. Ces instructions seront répétées encore et encore, aussi longtemps que la condition a la valeur `true`.

Nous rencontrerons très souvent le motif de programmation suivant : une boucle qui exécute des instructions un nombre donné de fois. Nous utiliserons alors une *variable de boucle* comme compteur. Il est d'usage courant de désigner par la lettre `i` la variable utilisée dans la boucle comme compteur, nous le ferons donc aussi. Voici un exemple dans lequel le corps de la boucle `while` est exécuté 100 fois :

```
int i = 0;
while ( i < 100 )
{
    instruction;
    instruction;
    ...
    i = i + 1;
}
```

Il est important de noter que nous utilisons ici un concept que nous n'avions pas encore rencontré : la *variable locale*.

Une variable locale est une variable similaire à un champ. Nous pouvons l'utiliser pour y stocker des valeurs comme des nombres entiers ou des références à d'autres objets.

Elle diffère d'une variable d'instance par plusieurs aspects :

- une variable locale est définie dans le corps d'une méthode et non au début de la classe ;
- les mots clef `private` ou `public` ne figurent pas dans sa déclaration ;
- elle n'existe que jusqu'à la fin de l'exécution de la méthode dans laquelle elle est définie, elle est effacée ensuite.

À proprement parler, le dernier point ci-dessus n'est pas totalement correct. Les variables locales peuvent également être déclarées dans d'autres blocs, comme, par exemple, dans une instruction conditionnelle `if` ou dans le corps d'une boucle. Elles n'existent que jusqu'à ce que l'exécution du bloc dans lequel elles ont été déclarée se termine.

On déclare une variable locale en écrivant une ligne qui commence par le type de la variable et qui se termine par le nom de celle-ci :

```
int i;
```

Après avoir déclaré la variable, nous pouvons lui attribuer une valeur. Cela donne les deux instructions ci-dessous :

```
int i;
i = 0;
```

En Java, on peut utiliser le raccourci suivant pour écrire les deux instructions sur une seule ligne, pour en même temps déclarer la variable et lui assigner une valeur :

```
int i = 0;
```

Voyons de nouveau la structure de la boucle `while` ; nous devrions maintenant être capables de comprendre ce qu'elle fait en gros. Nous utilisons une variable `i` et l'initialison à 0. Nous répétons ensuite l'exécution du corps de la boucle en ajoutant 1 à `i` à la fin de chaque exécution du corps de la boucle, tant que la valeur de `i` est inférieure à 100. Lorsque nous atteignons 100, l'exécution de la boucle s'arrête. Le programme reprend alors en exécutant les instructions qui suivent immédiatement le corps de la boucle.

Il y a encore deux détails techniques méritant d'être cités :

- Nous utilisons l'instruction

  ```
  i = i + 1;
  ```

  à la fin du corps de la boucle pour incrémenter notre variable locale de 1 chaque fois que nous avons exécuté les instructions du corps de la boucle. C'est important. Une erreur fréquente est l'oubli de l'incrémentation du compteur de boucle. Dans ce cas, la valeur de la variable ne change jamais, la condition reste toujours vraie, et la boucle s'exécute en continu, sans jamais s'arrêter. Cela s'appelle une *boucle infinie*, et c'est la cause de beaucoup d'erreurs de programmation.
- Notre condition stipule que nous devons exécuter le corps de la boucle tant que la valeur de `i` est inférieure (nous utilisons le symbole $<$) à 100 et non pas inférieure ou égale (symbole $\leq$). Le corps de la boucle ne sera donc pas exécuté lorsque `i` vaudra 100. À première vue, on pourrait se dire que cela signifie que la boucle ne sera exécutée que 99 fois et non 100 fois. Ce n'est pourtant pas le cas ; en effet, vu que nous avons commencé à compter à 0 et non à 1, nous exécutons le corps de la boucle 100 fois, en comptant de 0 à 99. Il est très fréquent de commencer à compter depuis 0 en programmation ; nous verrons bientôt l'intérêt d'une telle pratique.

Maintenant que nous connaissons l'instruction `while`, nous pouvons l'utiliser pour créer toutes nos touches de piano.

Notre piano aura 12 touches blanches. Nous pouvons créer ces 12 touches en plaçant l'instruction qui permet de créer une seule touche dans le corps d'une boucle qui va s'exécuter 12 fois :

```
int i = 0;
while (i < 12)
{
        addObject (new Key ("g", "3a.wav"), 300, 140);
        i = i + 1;
}
```

Exercice 5.16

Placer le code ci-dessus dans votre méthode `makeKeys` pour y intégrer la boucle `while` générant automatiquement les touches. Qu'observez-vous ?

Lorsque l'on teste ce code, on a en premier lieu l'impression qu'une seule touche a été créée. Ce n'est qu'une impression, toutefois. En fait, nous obtenons bien 12 touches, mais comme elles ont été insérées toutes au même endroit (elles ont les mêmes coordonnées !), elles sont exactement superposées et nous ne pouvons voir que celle du dessus. On peut se rendre compte qu'elles sont toutes présentes en les déplaçant avec la souris.

Exercice 5.17

Comment modifier le code de façon à ce que les différentes touches n'apparaissent pas toutes au même endroit ? Pouvez-vous changer ce code de sorte à ce que les touches soient côte à côte, sans espace intercalaire ?

La raison qui fait que toutes nos touches se sont superposées est la suivante : nous les avons toutes insérées à la position $(300, 140)$ de notre monde. Nous devons maintenant insérer chaque touche à un endroit différent. C'est maintenant chose relativement facile : Nous pouvons faire usage de notre variable `i` pour atteindre notre but.

Exercice 5.18

Combien de fois le corps de notre boucle sera-t-il exécuté ? Quelle est la valeur de la variable `i` durant chacune de ces exécutions ?

Nous pouvons remplacer la coordonnée x fixée à 300 par une expression qui inclut la variable `i` :

```
addObject (new Key ("g", "3a.wav"), i * 63, 140);
```

(L'astérisque "$*$" représente ici l'opérateur de multiplication.)

Nous avons choisi `i * 63` parce que nous savons que l'image de chaque touche a une largeur de 60 pixels. Les valeurs de `i` sont, au fur et à mesure de l'exécution de la boucle : 0, 1, 2, 3, et ainsi de suite. Les coordonnées x des touches seront donc 63, 126, 189, et ainsi de suite.

Après avoir testé ce code, nous observons que la première touche de gauche sort du cadre du piano. C'est normal car la position d'un objet dans Greenfoot fait référence au centre de cet objet, ce qui place le centre de notre première touche à 0 relativement à x ; la première touche est donc à moitié hors du cadre. Pour régler ce problème, nous ajoutons un décalage fixe à la première coordonnée de chaque touche. Le décalage est choisi de façon à ce que l'ensemble des touches apparaisse au milieu de notre piano :

```
addObject (new Key ("g", "3a.wav"), i * 63 + 54, 140);
```

La coordonnée y reste constante, vu que nous voulons que toutes nos touches soient placées à la même hauteur.

Exercice 5.19

L'utilisation de constantes dans notre code, comme 140 ou 63 dans les instructions ci-dessus, n'est

en général pas la meilleure solution, vu que cela rend notre code vulnérable aux changements. Par exemple, si nous remplaçons nos images de touches par de plus jolies images qui ont une taille différente, notre programme les placeraient pas correctement. Nous pouvons éviter l'utilisation directe de ces nombres en appelant les méthodes `getWidth()` et `getHeight()` de l'image de notre touche. Pour le réaliser, il faut en premier lieu assigner l'objet touche à une variable locale de type `Key` au moment de sa création, et ensuite utiliser l'expression

```
key.getImage().getWidth()
```

à la place du nombre 63. On procède de manière analogue pour la hauteur.

Pour remplacer le nombre fixe 54, on utilisera également la méthode `getWidth()` de l'image du piano.

Une fois ces modifications réalisées, notre code placera toujours les touches de la bonne façon, même si leur taille change.

Notre programme dispose maintenant les touches blanches correctement ; c'est un bon point. Un problème évident à ce stade est que les touches du piano réagissent toutes à la même touche du clavier et jouent toutes la même note, également. Pour modifier cet état de fait, nous avons besoin d'un nouveau concept : le *tableau*.

5.6 Utiliser des tableaux

Dans l'état actuel du programme, nos 12 touches sont créées et placées correctement sur l'écran, mais elles réagissent toutes à la touche "g" et jouent donc toutes la même note. Ceci malgré le fait que nous avons préparé un constructeur permettant d'attribuer différentes touches du clavier et différents sons à une touche du piano. En effet, comme nous avons créé toutes nos touches avec la même ligne de code exécutée en boucle, elles auront toutes "g" et "3a.wav" comme paramètres.

La solution à ce problème est analogue à celle que nous avons employé pour modifier la coordonnée x de nos touches : Il nous faudra utiliser des variables pour la touche du clavier et le nom du fichier son, et leur attribuer une valeur différente chaque fois que le corps de la boucle sera exécuté.

Ce qui est plus problématique à réaliser que dans le cas de la modification des coordonnées, car les noms des touches et des fichiers son ne sont pas calculables facilement à partir du compteur `i`. Comment allons nous obtenir ces valeurs ?

Voici notre réponse : Nous les stockerons dans un tableau.

Un tableau est un objet qui peut contenir un grand nombre de variables, et qui peut en conséquent stocker un grand nombre de valeurs. Nous pouvons illustrer ce concept de la façon suivante : Supposons que nous ayons une variable dont le nom est `name`, de type `String`. À cette variable, nous attribuons la chaîne de caractères "`Fred`" :

```
String name;
name = "Fred";
```

Ce cas de figure est très simple, la variable est une boîte qui peut contenir une seule valeur. La valeur est stockée dans la variable.

Dans le cas d'un tableau, nous créons un objet séparé qui peut contenir plusieurs variables. Nous pouvons alors stocker une référence à cet objet tableau dans une variable de type tableau que nous appellerons ici `names`.

Le code Java permettant de créer un tableau est le suivant :

```
String[] names;
names = {"a", "b", "c", "d", "e", "f", "g", "h", "i", "j", "k", "l"};
```

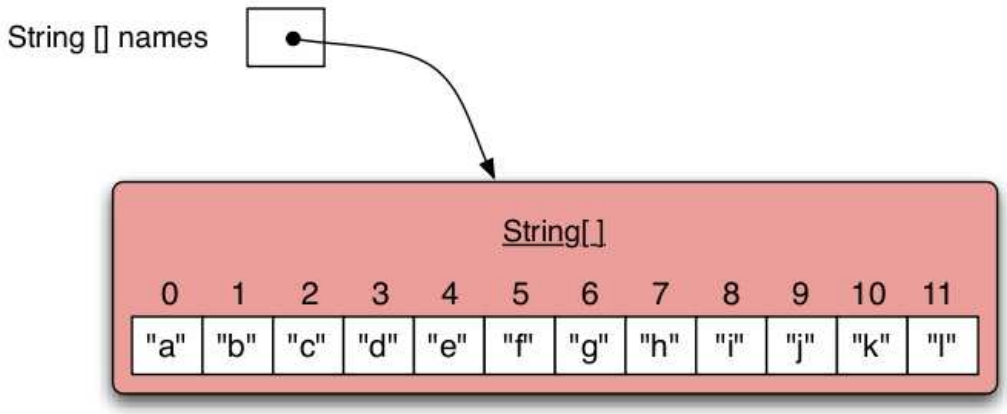
Dans la déclaration de la variable `names`, la paire de crochets `[]` indique que le type de la variable est un tableau. Le mot qui précède les crochets indique le *type des éléments* du tableau, c'est à dire le type de chaque variable stockée comme un élément de ce tableau. Ainsi, `String[]` représente un tableau de chaînes de caractères, tandis que `int[]` précède le nom d'un tableau d'entiers.

L'expression

```
{"a", "b", "c", "d", "e", "f", "g", "h", "i", "j", "k", "l"}
```

crée l'objet tableau et le remplit avec les chaînes de caractères "`a`" à "`l`". Cet objet est ensuite assigné à notre variable `names`.

Le diagramme ci-dessous illustre le fait que lorsqu'un objet tableau est attribué à une variable, celle-ci contient ensuite un *pointeur* vers cet objet.



Une fois que notre variable tableau est en place, nous pouvons accéder aux divers éléments du tableau en utilisant un *index*, soit le numéro représentant la position de l'élément dans le tableau. Dans la figure ci-dessus, on voit l'index de chaque chaîne de caractère du tableau figurer juste en dessus de l'élément.

Notons ici que l'on numérote les éléments à partir de 0, ce qui fait que la chaîne "`a`" se trouve en position 0, la chaîne "`b`" occupe la position 1, etc. En Java, on accède aux différents éléments en ajoutant au nom du tableau l'index de l'élément entre crochets. Par exemple,

```
names[3]
```

permet d'accéder à l'élément du tableau `names` qui occupe la position 3, soit la chaîne de caractères "d".

Pour notre projet de piano, nous pouvons maintenant préparer deux tableaux : le premier contenant dans l'ordre les noms des touches du clavier et le second les noms des fichiers son associés à ces touches. Nous pouvons également déclarer des champs dans la classe `Piano` pour ces tableaux et les y stocker. Voici le code source correspondant :

```
public class Piano extends World
{
    private String[] whiteKeys =
        { "a", "s", "d", "f", "g", "h", "j", "k", "l", ";", "'",
          "\\" };

    private String[] whiteNotes =
        { "3c", "3d", "3e", "3f", "3g", "3a", "3b", "4c", "4d", "4e",
          "4f", "4g" };

        // constructor and methods omitted.

}
```

On notera que les valeurs des éléments du tableau `whiteKeys` sont les touches de la ligne du milieu d'un clavier britannique. Les claviers étant différents d'un pays à l'autre, il faudra peut-être changer l'une ou l'autre touche pour s'accorder avec le clavier de la machine sur laquelle on travaille. L'autre chose un peu bizarre est la chaîne de caractères "\\". Le caractère "backslash" s'appelle un *caractère d'échappement* est a une signification spéciale à l'intérieur d'une chaîne de caractères Java. Pour créer une chaîne qui contient le backslash comme caractère normal, il vous faudra le taper deux fois. Ainsi, taper "\\" dans votre code source Java crée en fait la chaîne de caractères "\".

Nous disposons maintenant de tableaux qui contiennent la liste des touches du clavier et des fichiers son que nous voulons associer aux touches de notre piano. Nous pouvons alors adapter la boucle de la méthode `makeKeys` pour utiliser les éléments du tableau lors de la création des touches du piano. Voici le code source correspondant :

```
/**
* Create the piano keys and place them in the world.
*/
private void makeKeys()
{
    int i = 0;
    while (i < whiteKeys.length)
    {
        Key key = new Key(whiteKeys[i], whiteNotes[i] + ".wav");
        addObject(key, 54 + (i*63), 140);
        i = i + 1;
    }
}
```

Quelques remarques à propos de cet extrait de code :

- Nous avons déplacé la création de la nouvelle touche hors de l'appel de la méthode `addObject` à la ligne du dessus et assigné l'objet touche à une variable locale appelée `key`. Cela a été fait dans le but de clarifier le code : La ligne était devenue longue et plutôt difficile à lire. Décomposer une ligne de ce genre en deux étapes la rend plus facile à comprendre.
- Nous passons les expressions `whiteKeys[i]` et `whiteNotes[i]` en paramètre au constructeur de la classe `Key`. Cela signifie que nous utilisons notre variable de boucle `i` comme index de tableau, ce qui nous permet d'accéder séquentiellement aux différentes chaînes de caractères représentant les touches et aux différents fichiers son.
- Nous utilisons le symbole plus (`+`) entre l'expression `whiteNotes[i]` et la chaîne de caractères "`.wav`". La variable `whiteNotes[i]` contient également une chaîne de caractères. Lorsque l'opérateur `+` est utilisé entre deux chaînes de caractères, il effectue la *concaténation de chaînes de caractères*. La concaténation de chaînes est une opération qui colle deux chaînes ensemble et les transforme en une seule chaîne. En d'autres termes, dans le cas qui nous occupe, nous collons la chaîne "`.wav`" à la fin de la valeur de `whiteNotes[i]`. En effet, nous avons stocké dans le tableau la chaîne "`3c`" alors que le nom du fichier son est "`3c.wav`". Nous aurions pu stocker le nom complet du fichier dans le tableau, mais, dans la mesure où l'extension du fichier est la même pour tous les sons, nous pouvons économiser un peu de travail inutile en ajoutant cette extension automatiquement.
- Nous avons également remplacé le 12 dans la condition de la boucle `while` par l'expression

  ```
  whiteKeys.length
  ```

  L'attribut `length` d'un tableau nous renvoie le nombre d'éléments du tableau. Dans notre cas, nous avons bien 12 éléments ; nous aurions donc pu laisser le nombre 12 et le programme aurait fonctionné sans problème. Il est toutefois plus sûr d'employer l'attribut `length`. Dans le cas où nous voudrions un jour changer le nombre de nos touches, la boucle serait toujours correcte telle quelle, sans changement de la condition.

Après ces changements, nous devrions maintenant pouvoir jouer de notre piano avec la ligne médiane de notre clavier et chaque touche devrait produire un son différent.

Exercice 5.20

Réaliser les changements discutés ci-dessus dans votre propre scénario. Faire en sorte que chaque touche fonctionne. Le cas échéant, adapter le tableau `whiteKeys` à votre clavier.

Exercice 5.21

Le dossier sounds du scénario du piano contient plus de notes que celles que nous sommes en train d'utiliser ici. Changer le piano de sorte à ce que les sons joués soient tous une octave plus bas. Il suffit pour cela d'utiliser le son "2c" à la place du son "3c" pour la première touche et de continuer vers le haut à partir de là.

Exercice 5.22

Si vous le désirez, vous pouvez associer d'autres sons à vos touches de piano. Il suffit pour cela

d'enregistrer vos propres sons ou de trouver des sons sur internet. Déplacez ensuite les fichiers dans le dossier sounds et de faire en sorte que vos touches soient associées à ces nouveaux sons.

La version du piano dont nous disposons maintenant se trouve dans les scénarios du livre sous le nom de piano-4.

Il reste maintenant à compléter le piano. Cela revient naturellement à ajouter les touches noires.

Il n'y a rien de particulièrement nouveau à cela. Nous devons dans l'essentiel écrire du code similaire à celui qui nous a permis d'obtenir les touches blanches. Nous vous laisserons réaliser ceci sous la forme d'un exercice. Il est toutefois relativement compliqué de réaliser tout cela d'un seul coup. En général, lorsque l'on veut réaliser une tâche importante, il est bien de la décomposer en plusieurs petites étapes. Nous allons donc découper la tâche consistant à coder les touches noires en une suite d'exercices qui conduit à la solution étape par étape.

Exercice 5.23

Pour l'instant, notre classe `Key` ne peut produire que des touches blanches. Cela vient du fait que nous avons "codé en dur" (de l'anglais "hard-code") les noms des fichiers images ("white-key.png" et "white-key-down.png"). Utilisez votre capacité d'abstraction pour modifier la classe `Key` de façon à ce qu'elle puisse représenter des touches noires ou blanches. Nous avons déjà réalisé un travail similaire pour les noms des touches et les noms des fichiers sons : Il suffit d'ajouter deux champs et deux paramètres pour les noms des fichiers images et d'utiliser ensuite ces variables à la place des noms de fichiers codés en dur. Tester le résultat en créant quelques touches noires et quelques touches blanches.

Exercice 5.24

Modifier la classe `Piano` de sorte à ce qu'elle crée deux touches noires quelque part sur le piano.

Exercice 5.25

Ajouter deux tableaux à la classe `Piano` pour les touches du clavier correspondant aux touches noires et pour les notes de ces touches.

Exercice 5.26

Ajouter une boucle dans la méthode `makeKeys` de la classe `Piano` qui crée et dispose au bon endroit les touches noires. Cette tâche est rendue difficile par le fait que les touches noires ne sont pas régulièrement espacées, il n'y en a pas partout. Essayez de trouver vous-même une solution à ce problème. On pourra, par exemple, insérer dans le tableau un élément spécial indiquant l'absence de touche noire. La lecture de la note ci-dessous concernant la classe `String` est conseillée ici.

Exercice 5.27

L'implémentatiojn complète de ce projet inclut également une courte méthode permettant d'écrire une ligne de texte à l'écran. Étudier cette méthode et réaliser quelques changements : changer, par exemple, sa couleur et déplacer le texte de sorte à le centrer horizontalement.

Note : La classe `String`

Le type `String` que nous avons utilisé un grand nombre de fois jusqu'ici est défini par une classe. Trouvez cette classe dans la documentation Java et étudier ses méthodes. Il y en a un grand nombre et certaines sont souvent très utiles.

Vous verrez des méthodes servant à construire des sous-chaînes de caractères, obtenir la longueur d'une chaîne, convertir la casse et bien plus encore.

Une méthode particulièrement utile à la résolution de l'exercice 5.x ci-dessus est la méthode `equals` qui permet la comparaison de deux chaînes de caractères. Cette méthode renvoie `true` si les deux chaînes sont égales.

Nous n'irons pas plus loin dans le cadre de ce projet. Le piano est plus ou moins complet à présent. Nous pouvons y jouer des morceaux simples, et même des accords.

Sentez-vous libre d'ajouter des éléments à ce scénario. On pourrait ajouter une deuxième série de sons et ensuite un bouton sur l'écran permettant de changer d'une série de sons à l'autre.

5.7 Résumé des techniques de programmation

Dans ce chapitre, nous avons abordé deux concepts fondamentaux pour la programmation plus sophistiquée : les boucles et les tableaux. Les boucles nous permettent d'écrire du code qui exécute une suite d'instructions un grand nombre de fois automatiquement. Le type de boucle dont nous avons discuté ici s'appelle une *boucle while*. Le langage Java dispose d'autres types de boucles, que nous rencontrerons plus loin. Nous utiliserons les boucles dans énormément de nos programmes, il est donc essentiel de bien les comprendre.

À l'intérieur d'une boucle `while`, nous utilisons souvent le compteur de boucle pour faire des calculs ou générer des valeurs à chaque itération de l'exécution du corps de la boucle.

L'autre concept principal que nous avons introduit est celui de tableau. Un tableau nous met à disposition plusieurs variables du même type comprises dans un seul objet. Souvent, les boucles sont utilisées pour parcourir un tableau si nous devons faire quelque chose avec chacun de ses éléments. On accède à un élément donné en utilisant des crochets.

Nous avons aussi rencontré quelques nouveaux opérateurs : Nous avons utilisé les opérateurs AND (`&&`) et NOT (`!`) pour créer des expressions booléennes et avons vu que l'opérateur d'addition (`+`) sert à la concaténation des chaînes de caractères. La classe `String` figure dans la documentation de Java et dispose de nombreuses méthodes utiles.

6 Des objets qui interagissent : La gravité

Dans ce chapitre, nous allons étudier des interactions plus sophistiquées entre les objets d'un monde donné. Comme point de départ, nous examinerons l'une des interactions universelles entre objets qui existe : la gravité.

Dans ce scénario nous nous occupons de corps célestes comme des planètes et des étoiles. Nous allons simuler les trajectoires de ces corps célestes dans l'espace en utilisant la loi de la gravitation universelle de Newton. (Nous savons aujourd'hui que les formules de Newton ne sont pas complètement correctes, et que la théorie générale de la relativité d'Einstein décrit le mouvement des planètes de façon plus précise, mais la loi de Newton suffit à notre simulation élémentaire.

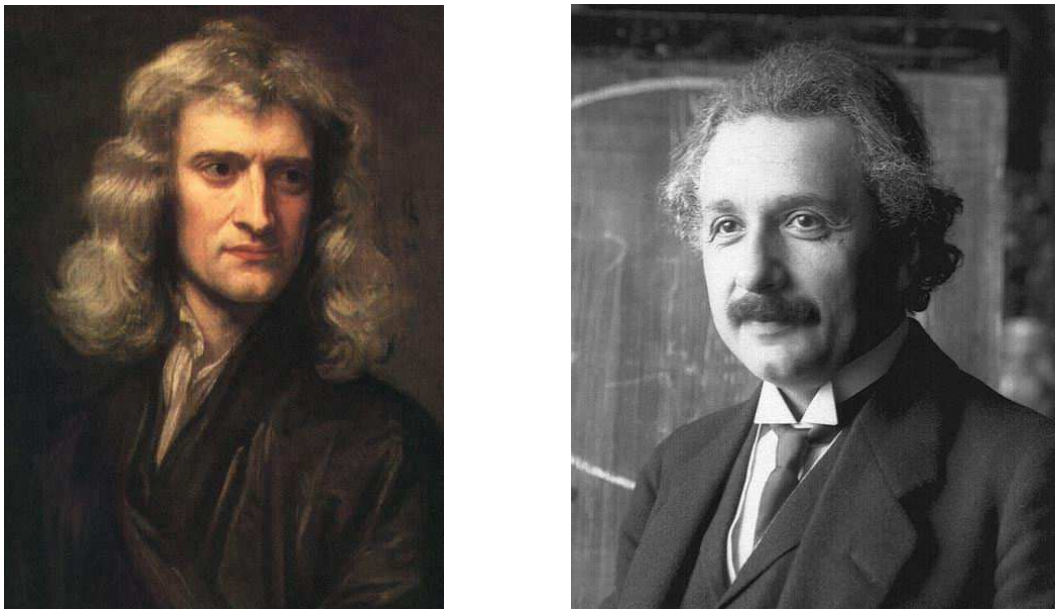


FIGURE 1 – Isaac Newton et Albert Einstein

Si le fait de devoir travailler avec des formules de physique vous inquiète, vous pouvez sans autre vous tranquilliser. Nous n'aurons pas besoin de nous plonger profondément dans la théorie de la gravité, et la formule que nous utiliserons est raisonnablement simple. De plus, à la fin du chapitre, nous transformerons ce scénario en une production artistique avec des effets visuels et sonores. Si votre intérêt est plutôt scientifique et technique, vous pourrez donner l'accent sur la physique. Si c'est l'art qui vous attire, vous pourrez vous concentrer sur cet aspect des choses.

6.1 Le point de départ : le laboratoire Newton

Nous allons commencer ce projet en nous intéressant à une version partiellement implémentée de ce scénario. Ouvrez le scénario Newtons-lab-1 qui se trouve dans le dossier book-scenarios. Vous verrez qu'une sous-classe de la classe `World` du nom de `Space` existe déjà. Nous avons également les classes `SmoothMover`, `Body` et `Vector`.

Exercice 6.1

Ouvrez le scénario Newtons-lab-1. Essayez-le (il s'agit de placer quelques corps célestes dans l'espace). Qu'observez-vous ?

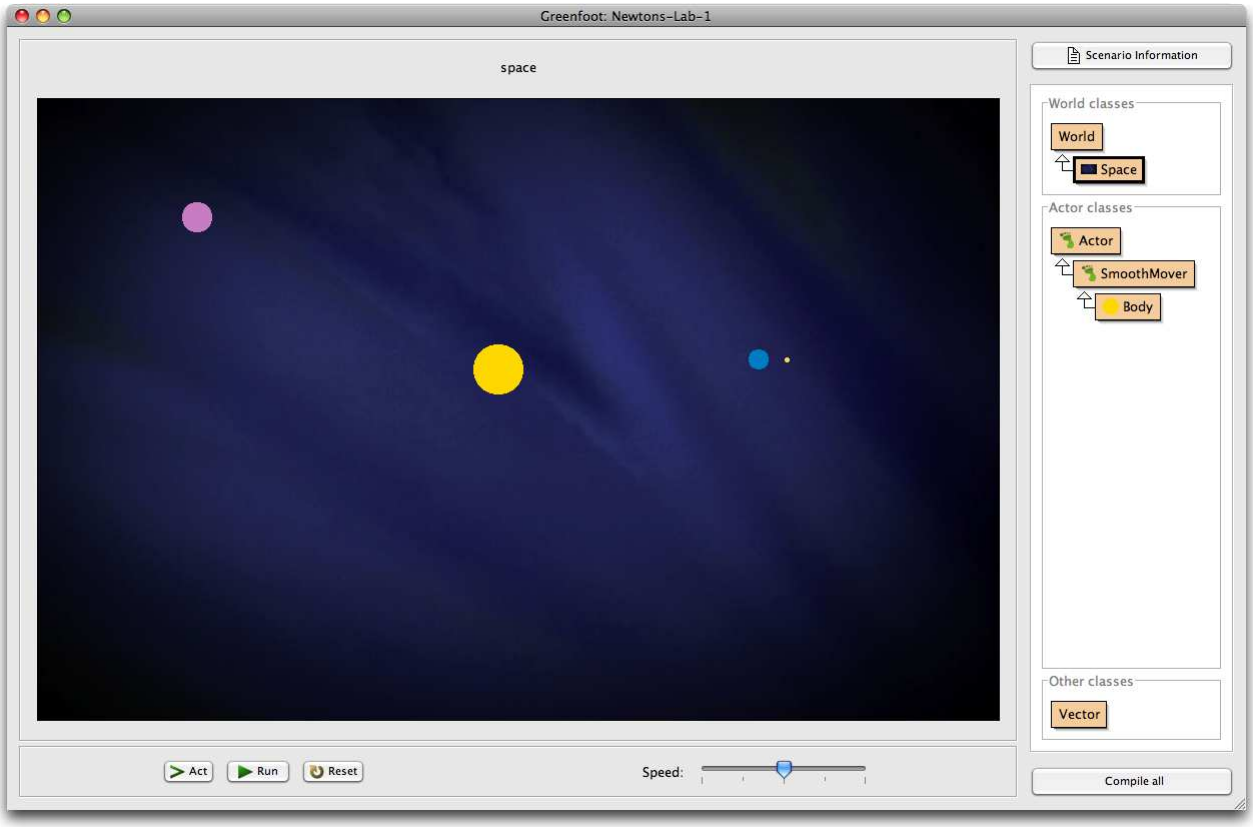


FIGURE 2 – Le scénario du laboratoire de Newton

Lorsque vous cliquez le bouton Run de ce scénario, vous constatez que vous pouvez placer des objets de type `Body` dans l'espace, mais que ces corps célestes ne bougent pas et qu'ils n'agissent pas de manière intéressante pour l'instant.

Avant de s'occuper d'étendre l'implémentation, observons ce scénario d'un peu plus près.

Un clic droit sur le titre du monde (le mot "space" en haut du rectangle de l'interface graphique), fait apparaître le menu contextuel de la classe `Space` qui nous donne accès aux méthodes publiques de cette classe.

On peut directement invoquer ces méthodes à l'aide du menu, comme le montre la figure ci-dessous :

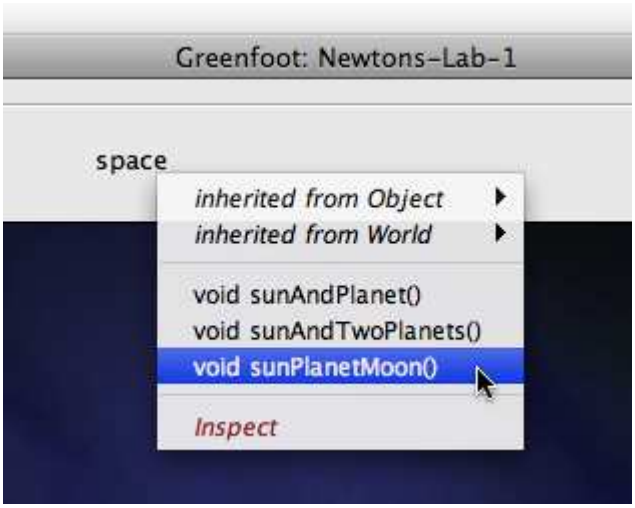


FIGURE 3 – Les méthodes de la classe `Space`

Exercice 6.2

Invoquez les différentes méthodes publiques de l'objet `space` de la classe `Space`. Que font-elles ?

Exercice 6.3

Faites un clic droit sur une étoile, puis une planète. Quelles sont les méthodes de ces objets ?

Exercice 6.4

Invoquer la méthode sunPlanetMoon depuis le menu contextuel de l'objet `space`. Trouver et noter la masse du soleil, d'une planète et de la lune.

Exercice 6.5

Lire le code source de la classe `Space` et observer comment les méthodes publiques y sont implémentées.

6.2 Deux classes auxiliaires : `SmoothMover` et `Vector`

Dans ce scénario, nous employons deux classes auxiliaires à usages multiples : `SmoothMover` et `Vector`. Ces classes permettent d'ajouter des fonctionnalités à un scénario donné et peuvent être utilisés dans différents contextes pour atteindre un but similaire. En fait, ces deux classes sont utilisées dans plusieurs projets Greenfoot.

La classe `SmoothMover` permet de "lisser" le mouvement d'un acteur en utilisant des nombres décimaux (de type `double`) plutôt que des entiers pour les coordonnées de l'acteur. Les champs de type `double` peuvent contenir des nombres comportant une partie décimale, dits "nombres à virgule", comme 2.4567. Ils permettent donc d'établir les coordonnées d'un point avec plus de précision.

En ce qui concerne la représentation à l'écran de l'objet, les coordonnées seront toujours arrondies aux entiers les plus proches, vu que la localisation de l'endroit où "peindre" l'objet à l'écran doit toujours être un donnée par un nombre entier de pixels. Nous aurons cependant une représentation interne décimale.

Un objet `SmoothMover` peut, par exemple, avoir 12.3 comme coordonnée sur l'axe des x. Si nous déplaçons maintenant cet acteur le long de cet axe par sauts de 0.6, ses positions successives seront :

$$12.3, \quad 12.9, \quad 13.5, \quad 14.1, \quad 14.7, \quad 15.3, \quad 15.9, \quad 16.5, \quad 17.1, \ldots$$

et ainsi de suite. Au niveau de l'affichage, les positions seront données par les entiers suivants :

$$12, \quad 13 \quad 14, \quad 14, \quad 15, \quad 15, \quad 16, \quad 17, \quad 17, \ldots$$

et ainsi de suite. En fin de compte, même si les nombres sont arrondis au moment de l'affichage, le mouvement semble plus fluide à l'observateur que s'il avait été codé à l'aide de champs de type `int`.

L'autre fonctionnalité ajoutée par la classe `SmoothMover` est un vecteur de mouvement. Tout objet d'une sous-classe de `SmoothMover` comprend un vecteur qui indique une direction courante et une vitesse de déplacement. Nous pouvons penser à un vecteur comme à une flèche invisible ayant une direction et une longueur, comme représenté ci-dessous : La classe `SmoothMover`

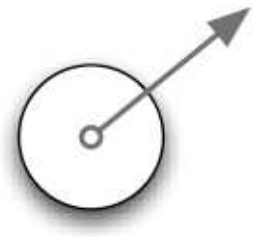


FIGURE 4 – Un objet `SmoothMover` et son vecteur de déplacement

dispose de méthodes pour modifier son vecteur de déplacement, et une méthode `move` qui fait se déplacer l'acteur selon l'état courant de son vecteur de déplacement.

Note : Les classes abstraites

Si vous faites un clic droit sur la classe `SmoothMover`, vous constaterez que vous ne pouvez pas créer d'objets de ce type. Le constructeur brille par son absence.

Si nous examinons le code source de cette classe, nous pouvons voir le mot-clef `abstract` dans l'entête de classe. Il est possible de déclarer des classes abstraites pour empêcher la création d'instances de ces classes. Les classes abstraites ne servent que comme super-classes d'autres classes et ne peuvent pas servir à créer des objets directement.

Exercice 6.6

Placer un objet de la classe `Body` dans l'espace. En examinant le menu contextuel de l'objet, déterminer quelles sont les méthodes de la classe `SmoothMover` dont cet objet hérite. Prendre note des noms de ces méthodes.

Exercice 6.7

L'un des noms des méthodes apparaît deux fois. Lequel ? En quoi diffèrent ces deux méthodes portant le même nom ?

Terminologie

En Java, il est parfaitement légal d'avoir deux méthodes portant le même nom, aussi longtemps que les listes de leurs paramètres sont différentes. Cela s'appelle de la **surdéfinition** ou **surcharge**. (Le nom de la méthode est **surchargé** ; il se réfère à plus d'une méthode.)

Lorsque nous appelons une méthode surdéfinie, c'est au moment de l'exécution du code que le système examine les paramètres fournis et décide en conséquence laquelle des méthodes a été appelée.

Nous dirons également que les méthodes n'ont pas la même signature.

La deuxième classe auxiliaire, `Vector`, implémente le vecteur lui-même et est utilisée par la classe `SmoothMover`. Notez bien le fait que la classe `Vector` ne fait pas partie du groupe des sous-classes de la classe `Actor`. Il ne s'agit pas d'un acteur ; un vecteur n'apparaîtra jamais en tant que tel dans le monde. Les objets de cette classe ne sont jamais créés que par d'autres objets de type `actor`.

Les vecteurs peuvent être représentés de deux façons : d'une part par une paire de longueurs (dx, dy) représentant un déplacement selon l'axe des x pour dx et un déplacement selon l'axe des y pour dy ; d'autre part par une paire de valeurs spécifiant la direction et la longueur du vecteur (α, ℓ).

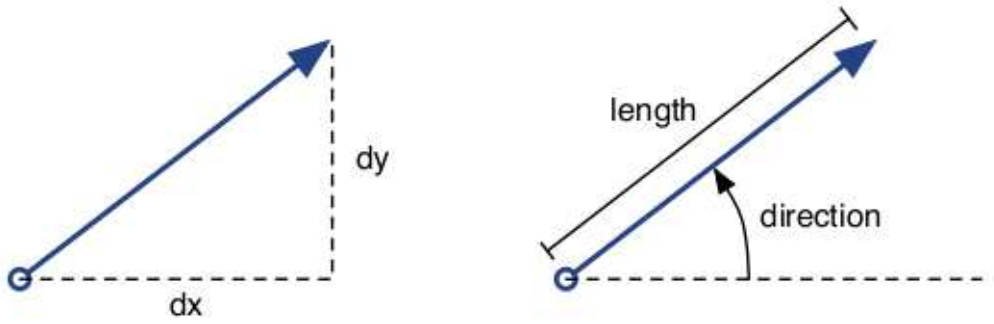


FIGURE 5 – Deux façons de représenter un vecteur

La première représentation qui utilise les deux déplacements selon x et y, s'appelle la représentation *cartésienne*. L'autre représentation, utilisant un angle et une longueur, s'appelle représentation *polaire*. Vous verrez ces deux noms dans le code source de la classe `Vecteur`.

Nous verrons par la suite que pour parvenir à nos fins, nous aurons parfois besoin de la représentation cartésienne et parfois de la représentation polaire. En effet, l'une est parfois plus facile à utiliser que l'autre dans un certain contexte. C'est pour cette raison que notre classe `Vector` est écrite de façon à pouvoir avoir recours à l'une ou l'autre représentation. Les conversions nécessaires seront faites automatiquement à l'intérieur de l'objet.

Exercice 6.8

Familiarisez-vous avec les méthodes des classes `SmoothMover` et `Vector` en ouvrant l'éditeur et en étudiant leur définition dans la vue *Documentation*, accessible depuis l'éditeur. (On peut passer à cette vue en utilisant le menu qui se trouve en haut à droite de la fenêtre de l'éditeur.) Vous pouvez aussi lire le code source si vous le désirez, mais ce n'est pas nécessaire à ce stade du projet.

Exercice 6.9

Placer une instance de la classe `Body` dans l'espace. Quelles sont les méthodes héritées de la classe `SmoothMover` que l'on peut appeler de façon interactive, par le biais du menu contextuel de l'objet ? Lesquelles ne peut-on pas appeler à ce stade ?

6.3 La partie déjà existante de la classe `Body`

Exercice 6.10

Ouvrir le code source de la classe `Body` et l'examiner.

```
public class Body extends SmoothMover
{
        // Du code laissé de côté...

    private double mass;

    /**
     * Construct a Body with default size, mass, movement and color.
     */
    public Body()
    {
        this (20, 300, new Vector(0, 1.0), defaultColor);
    }

    /**
     * Construct a Body with a specified size, mass, movement and color.
     */
    public Body(int size, double mass, Vector movement, Color color)
    {
        this.mass = mass;
        addForce(movement);
        GreenfootImage image = new GreenfootImage (size, size);
        image.setColor (color);
        image.fillOval (0, 0, size-1, size-1);
        setImage (image);
    }

    // Encore du code laissé de côté...
}
```

Après observation attentive de ce code, on peut relever deux aspects qu'il vaut la peine de discuter plus avant. Le premier est le fait que la classe dispose de deux constructeurs comme on peut le constater sur l'extrait de code ci-dessus. C'est un autre exemple de surcharge d'une méthode : il est parfaitement légal d'avoir deux constructeurs dans une classe si les deux listes de paramètres sont différentes.

Dans notre cas, l'un des constructeurs n'a pas de paramètres du tout et l'autre en a quatre.

Terminologie

Un constructeur sans paramètres est aussi appelé *constructeur par défaut*.

Le constructeur par défaut nous permet de créer des corps célestes de manière interactive sans devoir spécifier tous les détails. Le second constructeur permet la création d'un corps céleste ayant une masse, une taille, une couleur et un vecteur déplacement spécifiques. Ce constructeur est utilisé, par exemple, dans la classe `Space` pour créer le soleil, une planète et la lune.

Le second constructeur définit l'état initial de l'acteur en utilisant toutes les valeurs passées en paramètre. Le premier constructeur a l'air plus mystérieux, il ne comporte qu'une seule ligne de code :

```
this (20, 300, new Vector(90, 1.0), defaultColor);
```

Cette ligne de code ressemble à un appel de méthode à cela près qu'elle fait usage du mot-clef `this` à la place du nom d'une méthode. Au moment de cet appel, le constructeur lance l'exécution de l'*autre constructeur*, celui qui a quatre paramètres, et lui fournit quatre valeurs par défaut. Il n'est pas possible d'utiliser le mot-clef `this` de cette façon (comme un nom de méthode) à un autre endroit que dans le corps d'un constructeur pour appeler un autre constructeur dans le cadre de la construction initiale d'un objet.

Dans le code ci-dessus, on voit également la ligne

```
this.mass = mass;
```

dans laquelle on utilise également le mot-clef `this`. Il s'agit encore d'un exemple de surcharge : le même nom est utilisé pour deux variables (un paramètre et une variable d'instance). Lorsque nous écrivons l'instruction d'affectation ci-dessus, nous devons spécifier à quelle variable `mass` nous nous référons de chaque côté du signe égal.

Si on écrit `mass` tout court, c'est la déclaration la plus proche de cette variable qui sera utilisée ; on se réfère donc dans ce cas au paramètre du constructeur. Lorsque nous écrivons `this.mass`, par contre, nous spécifions que nous nous référons au champ `mass` de l'objet courant. Cette ligne attribue donc la valeur du paramètre local `mass` au champ `mass` de l'objet courant.

Exercice 6.11

Enlever le "`this.`" qui se trouve devant le mot `mass` dans la ligne de code examinée ci-dessus de façon à n'avoir plus que la ligne

```
mass = mass;
```

Le compilateur accepte-t-il cette ligne de code ? Peut-on exécuter le résultat de la compilation ? Que pensez-vous que ce code fasse ? Quel est son effet ?

Créer un objet et utiliser la fonction "Inspect" de cet objet pour examiner le champ `mass`.

Une fois l'expérience terminée, remettre le code dans l'état initial.

Le deuxième aspect de la classe `Body` qui mérite un commentaire est le suivant : Observons les deux lignes qui figurent tout en haut de la classe et que nous reproduisons ci-dessous :

```
private static final double GRAVITY = 5.8;
private static final Color defaultColor = new Color(255, 216, 0);
```

Ces deux déclarations ressemblent à des déclarations de variables d'instance, excepté en ce qui concerne l'utilisation des deux mots-clefs `static final` qui sont insérés après le mot-clef `private`.

Ce sont là deux *constantes*. Une constante a en commun avec une variable d'instance le fait que nous pouvons utiliser son nom dans notre code pour nous référer à sa valeur. Cette valeur ne pourra toutefois jamais changer, elle est *constante*. C'est le mot-clef `final` qui fait de la déclaration une déclaration de constante.

L'effet du mot-clef `static` est, quant à lui, de permettre le partage de la constante parmi tous les acteurs de la classe et de faire en sorte que nous n'ayons pas besoin d'une copie séparée de la constante pour chaque objet. Nous avons déjà rencontré le mot-clef `static` plus haut, au chapitre 3, dans le contexte des méthodes de classe. Tout comme une méthode de classe est associée à la classe elle-même (mais peut être appelée depuis les objets de cette classe), on peut accéder à un champ statique depuis l'instance de la classe dans la mesure où le champ est associé à la classe elle-même.

Dans le cas qui nous occupe, celui de la classe `Body`, les constantes déclarées sont d'une part une valeur pour la gravité[1] (utilisée dans la suite) et une couleur par défaut pour les corps célestes. Cette dernière constante est un fait un objet de type `Color` que nous étudierons de façon plus détaillée ultérieurement.

Il est recommandé de déclarer les champs qui ne changeront pas tout au long du programme comme des constantes. Grâce à cette précaution, le champ concerné ne pourra pas changer de valeur accidentellement quelque part dans le code.

6.4 Première extension : Créer le mouvement

Bon, assez observé le décor. Il est temps d'écrire du code et de faire en sorte que quelque chose se passe.

La première chose évidente à faire est de mettre les corps célestes en mouvement. Nous avons mentionné plus haut que la classe `SmoothMover` met à disposition une méthode `move()`, et donc, dans la mesure où un objet de `Body` est un `SmoothMover`, il peut avoir accès à cette méthode.

Exercice 6.12

Ajouter un appel à la méthode `move()` dans la méthode `act` de la classe `Body`. Tester l'effet. Quelle

1. Notre valeur pour la gravité n'a aucune relation directe avec une unité naturelle particulière utilisée en physique. C'est une constante arbitraire définie pour ce scénario. Dès que nous aurons commencé à implémenter l'application de la gravité à nos corps célestes, nous pourrons faire des expériences en changeant cette valeur.

est par défaut la direction du mouvement ? Quelle est la vitesse par défaut ?

Exercice 6.13

Créer plusieurs objets de type `Body`. Comment se comportent-ils ?

Exercice 6.14

Appeler les méthodes publiques de la classe `Space` (`sunAndPlanet()`, etc.) et cliquer le bouton Run du scénario. Comment se déplacent ces objets ? Comment sont définies leurs vitesses initiales et leurs directions de mouvement initiales ?

Exercice 6.15

Changer la direction des objets de type `Body` de sorte à ce qu'ils se déplacent par défaut vers la gauche. Après cette modification, un objet `Body` nouvellement créé par le constructeur devrait se déplacer vers la gauche à l'appel de la méthode `move()`.

Comme nous pouvons le constater à la suite de ces quelques exercices, il suffit de dire à nos corps célestes de se déplacer pour qu'ils le fassent. Ils ne vont toutefois se déplacer qu'en ligne droite. La raison en est que le mouvement (vitesse et direction) est contrôlé par le vecteur de mouvement et que rien ne modifie ce vecteur pour l'instant. Le mouvement d'un corps est donc constant.

6.5 Utiliser les bibliothèques des classes de Java

En lisant le code ci-dessus, nous avons rencontré la classe `Color` à la fois dans la classe `Space` et dans la classe `Body`. Le second constructeur de la classe `Body` attend un paramètre de type `Color`, et le code source de la classe `Space` crée des objets de type `Color` à l'aide d'expressions du type suivant :

```
new Color(248, 160, 86)
```

Les trois paramètres du constructeur de la classe `Color` sont les composants rouge, vert et bleu de cette couleur particulière. Toute couleur d'un écran d'ordinateur peut être décrite comme une composition de ces trois couleurs de base. Les questions que nous nous posons maintenant sont les suivantes : D'où vient cette classe ? Comment savoir de quels paramètres son constructeur a besoin ?

Un élément de réponse à cette question se trouve près du sommet de cette classe, au niveau de la ligne

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

La classe `Color` est l'une des nombreuses classes de la *Bibliothèque Standard des Classes de Java*. Le système Java vient avec une importante collection de classes utiles que nous pouvons utiliser sans autre. Petit à petit, nous apprendrons à connaître un bon nombre d'entre elles.

Nous pouvons voir la documentation de toutes les classes de cette bibliothèque en sélectionant Java Library Documentation dans le menu Help de Greenfoot. Cela ouvrira la documentation des bibliothèques de Java dans un navigateur Web, comme ci-dessous :

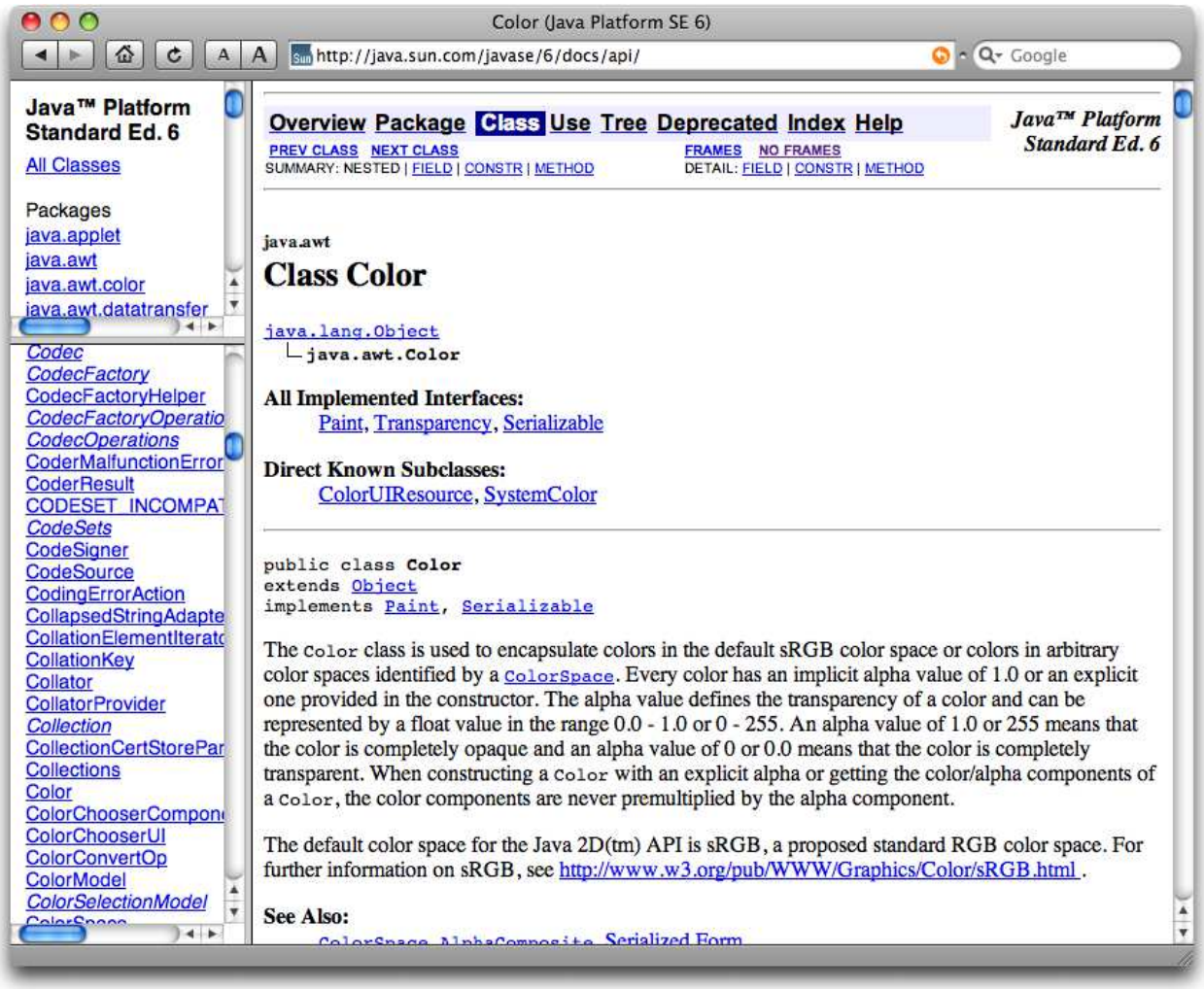


FIGURE 6 – La documentation de la bibliothèque des classes de Java

Exercice 6.16

Trouvez la classe `Color` dans la liste des classes. Sélectionnez-la. Examinez la documentation de cette classe. De combien de constructeurs dispose-t-elle ?

Exercice 6.17

Trouvez la description du constructeur que nous avons utilisé (celui à qui l'on doit fournir trois paramètres entiers). Quelle est le domaine dans lequel on peut choisir ces entiers ?

On peut constater qu'il y a littéralement des milliers de classes dans la bibliothèque des classes de Java. Pour faire un peu d'ordre dans cette longue liste, les classes sont groupées en *packages* ou *paquets*. Un paquet est un groupe de classes liées logiquement les unes aux autres. En haut de la documentation d'un classe, on peut voir dans quel paquet elle se trouve. En ce qui nous concerne, la classe `Color` se trouve dans le paquet `java.awt`.

Sitôt que nous désirons utiliser l'une des classes de la bibliothèque de Java dans l'un de nos scénarios, nous devons *importer* la classe, en utilisant une instruction d'importation, comme nous l'avons déjà vu plus haut. l'instruction d'importation consiste en le nom du paquet suivi par le nom de la classe, un point séparant ces deux éléments. Ainsi, pour pouvoir utiliser la classe `Color` du paquet `java.awt`, nous écrivons

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

Le fait d'importer une classe la rend utilisable à l'intérieur de notre scénario, comme si c'était l'une de nos propres classes. Après l'importation, on peut créer des objets de cette classe, appeler ses méthodes, en bref faire tout ce que l'on peut faire avec une classe quelconque.

La bibliothèque de Java est plutôt intimidante au départ, vu le nombre de classes qu'elle contient. Ne vous inquiétez pas, nous n'utiliserons qu'un nombre restreint de celles-ci, et nous les introduirons une à une au fur et à mesure de nos besoins.

Nous aurons toutefois besoin de l'une de ces classes très bientôt : dans le prochain paragraphe.

Ce que nous voulons faire ensuite est "ajouter la gravité" à ce scénario dans le sens suivant : Sitôt qu'il y aura plus de deux corps célestes dans notre espace, les forces gravitationnelles exercées par les uns sur les autres devraient changer le mouvement de chacun.

6.6 Ajouter la force de gravitation

Commençons par observer ci-dessous la méthode `act` telle qu'elle est implémentée dans notre classe `Body`. Si vous n'avez pas fait l'exercice 6.12, l'appel à la méthode `move` ne s'y trouve pas. Vous pouvez dans ce cas l'ajouter maintenant.

```
/**
 * Act. That is: apply  the gravitation forces from
 * all other bodies around, and then move.
 */
public void act()
{
    move();
}
```

Alors que le code ne contient pour l'instant qu'un appel à la méthode `move`, le commentaire décrit de façon claire ce que nous voulons que la méthode réalise : Avant de bouger, nous devrions avoir appliqué à notre corps céleste les forces causées par l'attraction gravitationnelle de tous les autres objets de l'espace. Nous pouvons donner une esquisse de la tâche à réaliser en pseudo-code :

```
appliquer les forces exercées par les autres corps()
{
        obtenir tous les corps de l'espace;

        for chaque corps
```

```
            {
                    appliquer la gravité de ce corps sur le nôtre;
            }
    }
```

Vu que ce n'est pas une chose facile à réaliser, nous commençons par créer une méthode séparée pour cette tâche. Créer une méthode vide visant à la réalisation d'une tâche peut sembler une tâche triviale qui ne permet pas de réaliser grand-chose au départ, mais cela va nous aider énormément à décomposer notre problème en sous-problèmes plus faciles et cela nous aidera à structurer notre pensée.

```
/**
 * Act. That is: apply  the gravitation forces from
 * all other bodies around, and then move.
 */
public void act()
{
    applyForces();
    move();
}

/**
 * Apply the forces of gravity from all other celestial bodies in
 * this universe.
 */
private void applyForces()
{
    // work to do here
}
```

Note : les méthodes privées

L'en tête de la méthode que nous avons créée ci-dessus commence par le mot-clef `private` et non par le mot-clef `public` comme nous en avions pris l'habitude.

Les méthodes peuvent être `public` ou `private`. Lorsque les méthodes sont prévues pour être appelées depuis l'extérieur de la classe, de façon interactive par un utilisateur ou depuis une autre classe, elles doivent être de type `public`. Si l'on ne veut pouvoir les appeler que depuis les autres méthodes de la même classe, comme c'est le cas ici, elles doivent être déclarées `private`.

Les méthodes de type `private` ne sont ni visibles, ni accessibles depuis l'extérieur de la classe. Déclarer les méthodes prévues uniquement pour un usage interne à l'aide du mot-clef `private` est une bonne pratique de programmation. Cela aidera à éviter les erreurs et permettra de documenter l'utilité de la méthode de manière plus claire.

Nous devons ensuite travailler sur la manière d'accéder à tous les autres objets de notre monde.

La classe `World` de Greenfoot dispose de méthodes qui nous permettent d'accéder aux objets qui s'y trouvent.

Exercice 6.18

Consulter la documentation de la classe `World` dans la documentation des classes de Greenfoot. Trouver toutes les méthodes qui nous donnent accès aux objets qui se trouvent dans le monde. Par écrit, faire la liste de toutes ces méthodes.

Pour nous, la méthode la plus intéressante pour l'instant est la suivante :

```
java.util.List getObjects(java.lang.Class cls)
```

Cette méthode nous fournit une liste de tous les objets d'une certaine classe qui existent dans notre monde. Le paramètre de cette méthode est de type `java.lang.Class` ; il s'agit de la classe nommée "`Class`" du paquet `java.lang`[2]. Nous avons vu des paramètres de ce type plus haut, dans le chapitre 4, lorsque nous avons utilisé les méthodes `canSee` et `eat` de la classe `Crab` pour manger les vers. Nous pouvons utiliser un appel à la méthode `getObjects` pour obtenir la liste de tous les corps céleste que nous avons créés dans notre monde :

```
getObjects(Body.class)
```

Nous pouvons également fournir le mot-clef `null` en paramètre afin d'obtenir une liste des objets du monde, toutes classes confondues.

```
getObjects(null)
```

Le mot-clef `null` est une expression spéciale qui signifie *rien* ou *pas d'objet*. Lors de son utilisation dans une liste de paramètres, nous ne passons "aucun objet" comme paramètre. La valeur `null` peut également être affectée à une variable.

La méthode `getObjects` est toutefois une méthode de la classe `World`, elle doit donc être appelée depuis un objet de type `World`. Vu que nous sommes en train d'écrire notre code dans la classe `Body`, nous devons tout d'abord obtenir un objet de type `World` pour appeler cette méthode depuis celui-ci. Il y a heureusement une méthode de la classe `Actor` qui nous donne accès à l'objet dont nous avons besoin. Son en-tête est le suivant :

```
World getWorld()
```

Trouvez cette méthode dans la documentation des classes de Greenfoot et lisez sa description.

La méthode `getWorld` nous renverra un objet de type `World` et nous pourrons alors appeler la méthode `getObjects` sur l'objet résultant :

```
getWorld().getObjects(Body.class)
```

Ce type de code peut être utilisé depuis un acteur pour obtenir tous les objets de la classe `Body` qui existe dans le monde du scénario. Examinons maintenant d'un peu plus près le type de la valeur de retour de cette méthode.

2. le paquet `java.lang` est spécial en cela qu'il contient les classes les plus utilisées et que les classes qu'il contient sont importées automatiquement. Nous ne devons pas écrire d'instruction d'importation pour les classes de ce paquet

Le type de la valeur de retour de la méthode `getObjects` est `java.util.List`. Cela nous indique qu'il y a un type appelé `List` dans le paquet `java.util` de la bibliothèque des classes standard et que nous obtiendrons un objet de ce type comme valeur de retour de cette méthode.

Le type `List` mérite un examen plus approfondi.

6.7 Le type `List`

Il est important d'apprendre à utiliser des collections d'objets à la fois pour la programmation dans Greenfoot et pour la programmation en général. Plusieurs méthodes de Greenfoot renvoient des collections d'objets, en général sous forme de liste. Le type de l'objet retourné est alors le type `List` du paquet `java.util`.

Aparté : les interfaces

Le type `List` est un peu différent des autres types d'objets que nous avons rencontré : il ne s'agit pas d'une classe, mais d'une *interface*. Les interfaces sont des structures abstraites de Java qui permettent d'implémenter plusieurs classes différentes. Les détails de ce concept ne sont pas importants pour nous maintenant ; il nous est suffisant de savoir que nous pouvons utiliser les objets de type `List` de la même façon que les autres objets : Nous pouvons consulter la documentation Java à propos de ce genre de type et nous pouvons appeler des méthodes existantes sur les objets correspondants. Par contre, nous ne pouvons pas créer directement d'objet de type `List`.

Exercice 6.19

Consulter la documentation de la classe `java.util.List` dans la documentation des classes de Java. Quel est le nom des méthodes utilisées pour ajouter un objet à la liste, enlever un objet de la liste et trouver combien d'objets contient la liste à un moment donné ?

Exercice 6.20

Quel est le nom exact de ce type, tel qu'il est donné en haut de la documentation ?

Lorsque nous avons examiné la méthode `getObjects` dans le paragraphe précédent, nous avons constaté que sa valeur de retour est un objet de type `java.util.List`. Pour pouvoir stocker cet objet, il nous faudra donc déclarer une variable de ce type. Nous ferons cela dans notre méthode `applyForces`.

Par contre, le type `List` doit être traité différemment de ce que nous avons fait jusqu'à présent pour les autres types. La documentation nous donne le nom de type suivant :

```
Interface List<E>
```

En plus du mot *interface* à la place du mot *class*, nous observons une nouvelle notation : le `<E>` après le nom du type.

Formellement, on appelle ceci un *type générique*. Cela veut dire que le type `List` a besoin d'un autre type passé en paramètre. Le deuxième type précise le type des éléments qui seront contenus dans la liste.

Par exemple, si nous avons affaire à une liste de chaînes de caractères, nous devons écrire

```
List<String>
```

Si, par contre, nous nous occupons d'une liste d'acteurs, nous écrivons

```
Interface List<Actor>
```

Dans tous les cas, le type spécifié entre les symboles d'inégalité (<>) doit être un type d'objet connu. Dans le cas qui nous occupe, nous voulons travailler avec une liste de corps célestes, ce qui fait que nous écrirons notre déclaration comme suit :

```
List<Body> bodies = getWorld().getObjects(Body.class);
```

Après l'exécution de cette ligne, notre variable `bodies` contient une liste de tous les corps célestes qui existent à ce moment dans le monde. On se souviendra à toutes fins utiles qu'il est nécessaire ici d'ajouter une instruction d'importation du paquet `java.util.List` au sommet de la classe `Body`.

La méthode `applyForces` peut commencer à prendre corps.

```
private void applyForces()
{
        List<Body> bodies = getWorld().getObjects(Body.class);

        //...
}
```

6.8 La boucle `for-each`

L'étape suivante à réaliser, maintenant que nous disposons d'une liste de tous nos corps célestes, est d'appliquer l'effet de la force de gravitation exercée par chacun de ces corps à notre mouvement.

Nous allons réaliser cela en parcourant notre liste, ce qui nous permet de prendre nos éléments un par un pour appliquer la force exercée sur le corps depuis lequel on appelle la méthode `applyForces`.

Java dispose d'une instruction spécialisée pour le parcours d'une collection d'objets et nous pouvons utiliser cette boucle ici. Cette instruction s'appelle une boucle `for-each`, et elle s'écrit en respectant la structure suivante :

```
for (Type variable : collection)
{
        instruction;
        ...
}
```

Dans la structure de l'instruction, `Type` représente le type de chaque élément de la collection, `variable` est le nom d'une variable qui est déclarée à cet endroit, nous pouvons donc lui donner le nom que nous voulons, `collection` est le nom de la collection à parcourir et `instruction` représente le début de la liste d'instructions que nous voulons exécuter. Un exemple appliqué à notre liste dont le nom est `bodies` peut clarifier cette explication :

```
for (Body body : bodies)
{
        body.move();
}
```

Souvenez- vous que Java est sensible à la casse : le mot `Body` débutant par un "`B`" majuscule sera traité différemment du mot `body` par le compilateur. Le nom avec majuscule se réfère au nom de la classe et le nom débutant par une minuscule désigne une variable pointant sur un objet. Le mot au pluriel `bodies` est encore une variable qui contient toute la liste.

Nous pouvons lire l'en-tête de l'instruction `for-each` un petit peu plus facilement si nous lisons "pour tout" à la place du mot-clef `for`, "dans" à la place des deux points et "faire" à la place de l'accolade ouvrant le bloc d'instructions.On peut donc "lire" l'instruction ci-dessus comme :

Pour chaque `body` dans `bodies`, faire...

Cette façon de lire l'instruction nous donne également une indication de ce que va faire la boucle : Elle va exécuter les instructions comprises entre les accolades une fois pour chaque élément de la liste `bodies`. Si, par exemple, il y a trois éléments dans cette liste, les instructions seront exécutées trois fois. Au début de chaque itération, avant l'exécution des instructions, la variable `body`, déclarée dans l'en-tête de la boucle se verra assigner l'un des éléments de la liste. La séquence d'actions réalisée par le programme sera donc :

1) `body =` premier élément de la liste `bodies` ;
   exécuter les instructions du corps de la boucle ;
2) `body =` deuxième élément de la liste `bodies` ;
   exécuter les instructions du corps de la boucle ;
3) `body =` troisième élément de la liste `bodies` ;
   exécuter les instructions du corps de la boucle ;
   ...

La variable `body` sert à accéder à l'élément courant de la liste dans le bloc d'instruction de la boucle. Nous pouvons, par exemple, appeler une méthode de l'objet, comme dans l'exemple présenté ci-dessus, ou encore passer l'objet en paramètre à une méthode pour réaliser autre chose.

Nous pouvons maintenant utiliser ce type de boucle pour appliquer la force de gravité exercée par les autres corps célestes sur celui-ci :

```
for (Body body : bodies)
{
    applyGravity(body);
}
```

Dans cet extrait de code, nous prenons simplement chaque élément (stocké dans la variable `body`) et le passons en paramètre à une autre méthode appelée `applyGravity`, que nous allons écrire bientôt.

Nous devons ajouter encore quelque chose : Dans la mesure où `bodies` est une liste de tous les corps de l'espace, elle inclut aussi bien l'objet courant (celui sur lequel nous voulons appliquer la gravité). Nous ne devons pas appliquer la gravité d'un objet à lui-même, ce qui fait que nous ajoutons une instruction `if` qui permet de n'appeler la méthode `applyGravity` seulement si l'élément de la liste n'est pas l'objet courant lui-même.

Le résultat est présenté ci-dessous. Noter l'emploi du mot-clef `this` qui est utilisé ici pour se référer à l'objet courant.

```
/**
 * Apply the forces of gravity from all other celestial bodies
 * in this universe.
 */
private void applyForces()
{
    List<Body> bodies = getWorld().getObjects(Body.class);

    for (Body body : bodies)
    {
        if (body != this)
        {
            applyGravity (body);
        }
    }
}

/**
 * Apply the gravity force of a given body to this one.
 */
private void applyGravity(Body other)
{
    // work to do here
}
```

6.9 Appliquer la gravité

Nous disposons maintenant de code qui nous règle la question de l'accès à tous les objets de l'espace, mais il nous reste à appliquer effectivement la force de gravité à tous nos objets. Il nous faut maintenant écrire la méthode `applyGravity` (un autre exemple de méthode privée).

Cela nous est maintenant un peu plus facile qu'avant l'écriture de la méthode `applyForces` vu qu'il nous suffit de nous occuper d'un seul objet à la fois : l'objet courant, et un autre objet passé en paramètre. Nous devons maintenant appliquer la force de gravitation de l'objet "paramètre" à l'objet courant. C'est à ce moment que la loi de Newton entre en jeu.

La formule de Newton qui donne la force en fonction des masses des objets et de la distance qui les sépare ressemble à ceci :

$$F = \frac{m_1 \cdot m_2}{d^2} \cdot G$$

où F est la force, m_1 et m_2 sont les masses, d est la distance et G la constante universelle de la gravitation.

En d'autres termes, pour calculer la force que nous devrons appliquer à l'objet courant, nous devons multiplier la masse de cet objet avec la masse de l'autre objet et diviser ensuite par le carré de la distance qui sépare les deux objets. La valeur est finalement multipliée par la constante G. Peut-être vous souvenez-vous que nous avons déjà défini une constante pour cette valeur dans notre classe, et que cette constante s'appelle `GRAVITY`.

Si vous êtes sûr de vous ou que vous cherchez l'aventure, vous aurez peut-être envie d'essayer d'implémenter `applyGravity` vous-même. Vous devrez alors créer un vecteur dont l'origine est le corps céleste courant et qui est dirigé vers l'autre corps, dont la longueur est spécifiée par la formule ci-dessus. Si ce n'est pas le cas, voici l'implémentation complète de la méthode `applyGravity`.

```
/**
 * Apply the gravity force of a given body to this one.
 */
private void applyGravity(Body other)
{
    double dx = other.getExactX() - this.getExactX();
    double dy = other.getExactY() - this.getExactY();
    Vector force = new Vector (dx, dy);
    double distance = Math.sqrt (dx*dx + dy*dy);
    double strength = GRAVITY * this.mass * other.mass / (distance * distance);
    double acceleration = strength / this.mass;
    force.setLength (acceleration);
    addForce (force);
}
```

Cette méthode n'est pas si compliquée qu'elle en a l'air. Nous calculons tout d'abord les distances entre notre objet et l'autre objet en termes des coordonnées x et y (nous stockons les résultats dans les variables `dx` et `dy`). Nous créons ensuite un nouveau vecteur en utilisant ces valeurs. Ce vecteur a maintenant la bonne directions, mais n'a pas encore la bonne longueur. Nous calculons ensuite la distance entre les deux objets en utilisant le *théorème de Pythagore* ($c^2 = a^2 + b^2$ dans le triangle rectangle, voir la figure 7 plus bas).

Tout ceci nous montre que la distance vaut la racine de la somme des carrés de `dx` et `dy`. Dans le code de la méthode `applyGravity`, nous utilisons la méthode `sqrt` de la classe `Math` pour calculer la racine carrée. (`Math` est une classe de java.lang, son importation est donc automatique.)

Exercice 6.21

Examiner la classe `Math` dans la documentation de Java. De combien de paramètres dispose la méthode `sqrt` ? Quels sont leurs types ? Quelle est le type de la valeur de retour de cette méthode ?

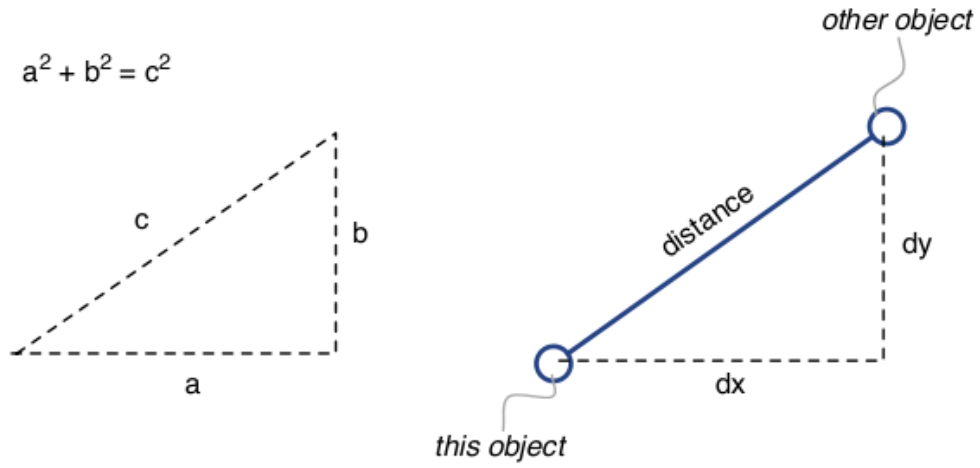


FIGURE 7 – Calcul de la distance à partir de `dx` et `dy`

Exercice 6.22

Dans la classe `Math`, trouver la méthode qui peut être utilisée pour trouver le maximum de deux entiers. Quel est son nom ?

La ligne de code suivante nous permet de calculer la norme de la force en utilisant la loi de la gravitation de Newton.

Il nous reste finalement à calculer l'accélération, vu que le le changement dans notre mouvement n'est pas seulement déterminé par la force de gravitation, mais aussi par la masse de notre objet : Plus notre objet est lourd, moins facilement il accélérera. L'accélération est calculée en utilisant la formule suivante :

$$a = \frac{F}{m}$$

où a est l'accélération, F la force et m la masse.

Une fois que nous avons calculé l'accélération, nous pouvons attribuer la longueur correcte à notre vecteur et ajouter la contribution de ce vecteur au mouvement de notre corps céleste. On peut le faire facilement en utilisant la méthode `addForce` qui fait partie intégrante de la classe `SmoothMover`.

Exercice 6.23

Établir la correspondance entre les variables de la classe `applyGravity` et celles de la formule de la gravitation de Newton, celles du théorème de Pythagore et celles de la formule du calcul de l'accélération ci-dessus. Quelle variable correspond à quelle partie de quelle formule ?

Avec ceci, notre scénario est terminé. On trouvera une implémentation du code décrit jusqu'ici dans le scénario Newtons-Lab-2.

Cette tâche a clairement nécessité plus de connaissances en maths et en physique que les autres que nous avons déjà effectuées. Si ce n'est pas votre domaine de préférence, ne vous faites pas de souci, nous retournerons à des projets moins "mathématiques" sous peu. Souvenez-vous : on peut faire ce qu'on veut en programmation. On peut s'occuper du côté mathématique des

choses, mais aussi réaliser des projets avec une forte composante artistique et créative.

6.10 Essayer le résultat

Maintenant que notre implémentation des forces de gravité est complète, il est temps de l'essayer. Nous pouvons commencer en utilisant les trois scénarios préfabriqués disponibles dans la classe `Space`.

Exercice 6.24

Dans le scénario contenant le code d'application de la gravité complet, essayer les trois méthodes d'initialisation de l'objet `space` de la classe `Space` (`sunAndPlanet()`, `sunAndTwoPlanets()`, et `sunPlanetMoon()`). Qu'observez-vous ?

Exercice 6.25

Faites plusieurs expériences en modifiant la gravité (la constante `GRAVITY` tout en haut de la classe `Body`).

Exercice 6.26

Faites plusieurs expériences en modifiant la masse et/ou le mouvement initial des corps célestes (définis dans la classe `Space`).

Exercice 6.27

Créez de nouveaux ensembles prédéfinis d'étoiles et de planètes et voyez comment ils interagissent. Pouvez-vous aboutir à un système stable ?

Piège

Faites attention en utilisant le constructeur de la classe `Vector`. Ce constructeur est surchargé : une version attend un `int` et un `double` comme paramètres, tandis que l'autre attend deux paramètres de type `double`. Ainsi,

```
new Vector(32, 12.0)
```

va appeler l'un des constructeurs, tandis que

```
new Vector(32.0, 12.0)
```

appellera l'autre constructeur, le résultat étant un tout autre vecteur !

Vous constaterez rapidement qu'il est très difficile de configurer les paramètres de sorte à ce que le système reste stable pendant longtemps. Certaines combinaisons masse/gravité résulteront en des collisions d'objets ; d'autres combinaisons feront s'échapper nos corps de leur orbite. Vu que nous n'avons pas implémenté la notion de collision d'objets, les corps en question vont simplement se traverser l'un l'autre. Lorsqu'ils seront très proches les uns des autres, toutefois, les forces deviendront énormes et ils seront parfois catapultés sur des trajectoires étonnantes.

Certains de ces effets sont similaires à ceux qui pourraient se produire dans la nature, malgré le fait que notre simulation est un peu approximative au vu de la loi simplifiée que nous avons utilisée. Le fait, par exemple, que les objets soient traités en séquence plutôt que simultanément a un effet sur le comportement de notre corps céleste et n'est pas représentatif de la réalité. Pour rendre la simulation plus réaliste, il nous faudrait calculer toutes les forces d'abord, sans bouger, puis faire se déplacer les corps ensuite, en tenant compte de tous les calculs d'un seul coup. Finalement, notre simulation ne modélise pas les forces de façon précise lorsque la distance entre deux corps devient très petite, ajoutant encore des effets non réalistes.

Nous pouvons nous poser la question de la stabilité en ce qui concerne notre système solaire dans la réalité. Bien que les orbites de nos planètes soient plutôt stable, il est difficile de prédire avec précision leur mouvement très loin dans le futur. Nous sommes plutôt certains qu'aucune des planètes ne va entrer en collision avec le soleil dans les millions d'années qui viennent, mais des petites variations d'orbites pourront se produire. Des simulations du même type que la notre (tenant compte de plus de détails et beaucoup plus précises, mais partant d'un principe similaire) ont été utilisées pour tenter de prédire les orbites futures. Nous avons constaté que ce genre de prédiction est très difficile à faire et les simulations plus complexes n'échappent pas à ce problème. En effet, des différences infimes dans les conditions initiales peuvent générer des différences énormes après quelques millions d'années. Si vous désirez en savoir plus, Wikipedia est un bon point de départ :

`http://en.wikipedia.org/wiki/Stability_of_the_Solar_System`

Au vu de la difficulté à définir un ensemble de paramètres créant un système qui reste stable ne serait-ce qu'un temps limité, nous pourrions être surpris par la stabilité de notre système solaire. Il y a une explication : Lorsque le système solaire s'est formé, la matière composant un nuage gazeux entourant le Soleil a formé des paquets de matière qui ont grossi en se cognant les uns les autres pour former des objets de plus en plus gros. Au départ, il y avait une quantité incalculable de paquets de matière sur orbite. Petit à petit, certains sont tombés sur le Soleil et d'autre se sont perdus dans les profondeurs de l'espace. Ce processus s'est terminé lorsque les morceaux ont été bien séparés les uns des autres, sur des orbites stables pour l'essentiel.

Il serait possible d'inventer une simulation qui modélise cette situation. Si nous créions un modèle de croissance des planètes à partir de millions de petits paquets de matière, nous pourrions observer le même effet : la formation de quelques planètes de grande taille sur des orbites plutôt stables. Il nous faudrait pour cela une simulation beaucoup plus complexe et détaillée et beaucoup de temps : Il nous faudrait un temps extrêmement long pour simuler cette situation, même sur un ordinateur d'une grande puissance.

6.11 Gravité et musique

Avant de laisser le scénario Newton's Lab derrière nous, il nous reste quelque chose à faire : ajouter de la musique, disons du bruit au minimum.

L'idée d'ajouter du son à un projet simulant la gravité a été ici inspirée par générateur de musique *Kepler's Orrery*, voir :

`http://www.art.net/~simran/GenerativeMusic/Kepler.html`

L'idée est la suivante : Nous ajoutons un certain nombre d'*Obstacles* dans notre monde. Lorsque les obstacles sont touchés par nos planètes, il produisent un son. Nous créons ensuite quelques planètes ou étoiles, les laissons se mettre en mouvement, et observons ce qui se passe.

Nous ne discuterons pas de l'implémentation en détail. En lieu et place, nous vous laisserons l'étudier vous-même, et nous bornerons à mettre l'accent sur les caractéristiques les plus intéressantes. Vous trouverez l'implémentation de cette idée dans le scénario Newtons-Lab-3.

Exercice 6.28

Ouvrir le scénario *Newtons-Lab-3* et lancer l'exécution. Lire le code source et essayer de comprendre comment cela fonctionne.

On donne ci-dessous la liste des changements les plus intéressants que nous avons réalisés à partir de la version précédente pour pouvoir créer ce nouveau scénario :

- Nous avons ajouté une nouvelle classe `Obstacle`. Vous pouvez distinguer facilement les objets de cette classe à l'écran. Les obstacles ont deux images : le rectangle orange que l'on voit la plupart du temps, et une version plus claire du rectangle qui apparaît lorsqu'ils sont touchés. Le but est de créer un effet d'éclairage. Les obstacles sont aussi associés à un fichier son, tout comme les touches du piano du scénario du chapitre 5. En fait, nous réutilisons ici les fichiers sons du scénario du piano, les sons restent les mêmes.
- Nous avons modifié la classe `Body` de façon à ce que les astres rebondissent sur les bords de l'écran. Cela donne un meilleur effet pour ce genre de scénario. Nous avons également augmenté un petit peu la gravité pour obtenir des mouvements plus rapides et modifié le code de façon à ce que nos astres ralentissent automatiquement dès que leur vitesse est trop grande. Sans cela, ils pourraient indéfiniment s'accélérer les uns les autres.
- Nous avons finalement ajouté dans la classe `Space` le code nécessaire à la création d'une rangée fixe d'obstacles et celui générant cinq planètes de taille, masse et couleur aléatoires.

L'implémentation de ces trois changements contient quelques extraits de code qu'il mérite de commenter ici.

- Dans la classe `Obstacle`, nous utilisons la méthode `getOneIntersectingObject`pour tester si un obstacle est touché par une planète. Le code s'écrit comme suit :

```
Object body = getOneIntersectingObject(Body.class);

if (body != null)
{
        ...
}
```

  La méthode `getOneIntersectingObject` qui est définie dans la classe `Actor` est à disposition de tout acteur de Greenfoot. Sa valeur de retour est un objet de type `Actor`. La méthode retourne l'objet en question s'il y a une intersection avec l'objet courant et `null` s'il n'y a pas d'intersection avec un acteur à ce moment. L'instruction conditionnelle `if` qui suit l'appel de la méthode teste si la variable `body` contient `null` permet de savoir s'il y a une intersection entre l'objet courant et un autre objet.
  C'est un exemple de *détection de collision* dont nous discuterons plus en détail au chapitre suivant.

- Nous avons enfin ajouté deux méthodes à la classe `Space`, portant les noms suivants : `createObstacles` et `randomBodies`. La première crée les obstacles et leur associe un nom de fichier son, d'une façon similaire à celle utilisée pour créer les touches du piano du chapitre 5. La seconde utilise une boucle `while` pour créer un certain nombre d'objets de type `Body`. Ces objets sont initialisés à partir de valeurs aléatoires. La boucle `while` fait un compte à rebours depuis un certain nombre jusqu'à 0, pour créer le bon nombre d'objets. Cela vaut la peine d'étudier ce nouvel exemple de boucle.

Exercice 6.29

Changer le nombre par défaut de corps célestes qui sont créés au moment de l'ouverture du scénario.

Exercice 6.30

Jouer avec les paramètres de mouvement pour voir s'il est possible de rendre le mouvement des planètes plus harmonieux. Les paramètres en question sont : la valeur de la constante `GRAVITY` ; la valeur de l'accélération utilisée lors d'un rebond contre le bord (valant 0.9 pour l'instant) ; le seuil de vitesse dont la valeur courante est 7 et l'accélération (0.9) utilisés dans la méthode `applyForces` pour ralentir les objets rapides ; et, finalement, la masse initiale des planètes fixée dans la classe `Space`.

Exercice 6.31

Créer dans votre scénario un arrangement d'obstacles différent.

Exercice 6.32

Utiliser d'autres sons pour les obstacles.

Exercice 6.33

Utiliser d'autres images pour les obstacles.

Exercice 6.34

Faire en sorte que les planètes changent de couleur chaque fois qu'elles rebondissent contre une paroi.

Exercice 6.35

Faire en sorte que les planètes changent de couleur chaque fois qu'elles touchent un obstacle.

Exercice 6.36

Créer un nouveau type d'obstacle qui s'allume et s'éteint chaque fois qu'il est touché par une planète. Une fois allumé, l'obstacle clignote et produit un son à intervalles fixes.

Exercice 6.37

Faire en sorte que le déroulement de l'action puisse être influencé par les touches du clavier. Par exemple, une pression sur la flèche droite pourrait ajouter une petite force dirigée vers la droite à tous les objets de type `Body`.

Exercice 6.38

Permettre la création d'autres planètes durant l'exécution du scénario. Un clic de souris dans l'univers durant l'exécution devrait faire apparaître une nouvelle planète à cet endroit.

Il y a un nombre incalculable d'autres manières de rendre ce scénario plus intéressant et plus agréable à regarder. Inventez quelques améliorations et implémentez-les !

6.12 Résumé des techniques de programmation

Dans ce chapitre, nous avons touché à un certain nombre de nouveaux concepts. Nous avons vu un nouveau scénario, Newton's lab, qui est une simulation les étoiles et les planètes dans l'espace. Les simulations forment un sujet intéressant d'une manière générale et nous y reviendrons au chapitre 9.

Nous avons vu l'utilisté des deux classes auxiliaires `SmoothMover` et `Vector`, qui nous ont toutes deux aidés à faire bouger nos objets de manière plus sophistiquée.

Parmi les sujets que nous avons introduits dans ce chapitre, l'un des plus importants à été l'utilisation de classes supplémentaires, provenant des classes standard de la bibliothèque Java. Nous avons utilisé les classes `Color`, `Math` et `List` provenant de cette bibliothèque. Nous utiliserons d'autres classes de ce type dans les chapitres suivants.

Un autre ajout à notre boîte à outils a été l'utilisation d'une nouvelle boucle : l'instruction `for-each`. On utilise cette boucle pour réaliser une action concernant chaque objet de d'une collection Java telle qu'une liste. Nous réutiliserons plus tard des extraits de code de ce genre.

Les boucles `for-each` sont particulièrement utiles pour traiter séquentiellement les objets d'une collection. Il ne peuvent toutefois pas être utilisés sans cette collection et ne fournissent pas un indice donnant le numéro de l'élément en cours de traitement. S'il nous faut un indice ou une boucle indépendante de toute collection, nous devons alors utiliser une boucle `for` ou une boucle `while` à la place.

Nous avons finalement utilisé des méthodes très utiles de l'API de Greenfoot comme `getObjects` de la classe `World` ou `getOneIntersectingObject` de la classe `Actor`. Cette dernière méthode nous conduit dans la domaine plus général de la détection de collision dont nous discuterons plus en détail dans le chapitre suivant qui sera consacré à l'étude du scénario Asteroids que nous avions déjà rencontré au chapitre 1.

7 Détection de collisions : le scénario **Asteroids**

Dans ce chapitre, nous n'allons pas introduire beaucoup de nouveaux concepts, mais nous allons plutôt reprendre certains de ces concepts et approfondir notre compréhension des sujets que nous avons abordés dans les deux chapitres précédents. Nous allons nous intéresser à nouveau à un scénario que nous avons rencontré très tôt dans ce livre : Asteroids.

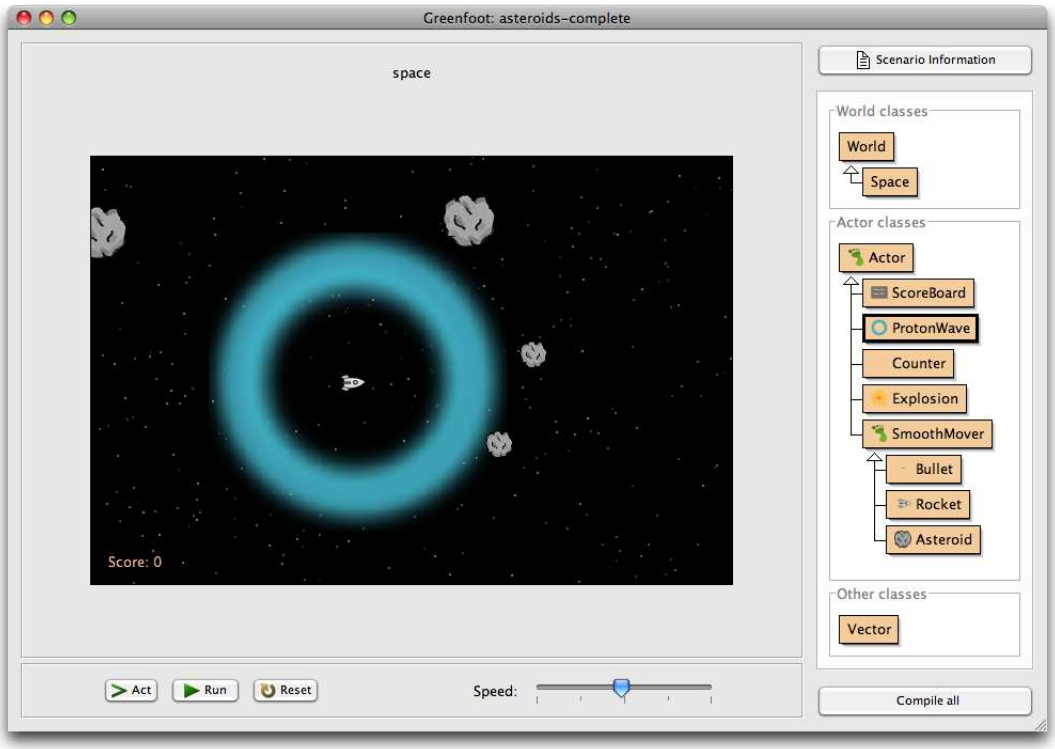


FIGURE 8 – Le nouveau scénario Asteroids, avec la vague protonique

La version du scénario Asteroids que nous utiliserons ici est légèrement différente de celle à laquelle nous nous somme intéressés plus haut. Il dispose de quelques ajouts comme une « vague protonique » et un compteur de score, mais il n'est pas complètement implémenté. Il lui manque encore des fonctions importantes, et le travail du chapitre sera de les programmer.

Nous utiliserons cet exemple pour étudier plus à fond le mouvement et la détection de collision. En termes de concepts de programmation Java, nous allons continuer à travailler avec des boucles et des collections.

7.1 Investigation : À quoi avons-nous affaire ?

Il nous faut commencer ce projet en examinant le code du scénario de base. Nous disposons d'une solution partiellement implémentée, qui s'appelle asteroids-1, dans le dossier chapter07 des scénarios du livre. Faites bien attention à utiliser la version du dossier chapter07 et non celle du chapitre 1.

Exercice 7.1

Ouvrir le scénario asteroids-1 du dossier chapter07 des scénarios du livre. Déterminer ce que ce

scénario peut faire et ce qu'il ne peut pas faire.

Exercice 7.2

Faire une liste de tout ce qu'il faudrait ajouter à ce projet.

Exercice 7.3

Quelle touche du clavier sert à tirer un projectile ?

Exercice 7.4

Placer une explosion dans le scénario en créant « à la main » un objet de la classe `Explosion`. Cela fonctionne-t-il ? Décrire ce qui se passe.

Exercice 7.5

Placer une vague protonique dans l'espace. Cela fonctionne-t-il ? Décrire ce qui se passe.

En examinant le scénario et ses possibilités vous aurez sans doute observé qu'il manque des fonctions essentielles :

- La fusée ne bouge pas. On ne peut pas la faire tourner ou la faire avancer.
- Il ne se passe rien lorsqu'un astéroïde rencontre la fusée. Il passe au dessus de la fusée, sans l'endommager.
- Il est donc impossible de perdre à ce jeu pour l'instant. Le jeu ne se termine jamais et le score final n'apparaît pas à l'écran.
- Les classes `Scoreboard`, `Explosion`, et `ProtonWave` que nous pouvons observer dans le diagramme des classes ne semblent pas tenir de rôle dans le scénario.

Nous pouvons faire une chose, toutefois : tirer des projectiles sur les astéroïdes. Si vous n'avez pas encore trouvé comment, cherchez maintenant. Les astéroïdes touchés par un projectile se cassent en plus petits morceaux ou disparaissent s'ils sont déjà de suffisamment petite taille.

Le but du jeu est évidemment d'éliminer tous les astéroïdes de l'écran en évitant que notre fusée soit touchée par l'un d'entre eux. Afin de rendre le jeu plus intéressant, nous voulons également ajouter une autre arme, la vague protonique. Et nous voulons afficher le score tout au long de la partie. Pour réaliser tout cela, nous avons du pain sur la planche !

- Nous devons implémenter le mouvement de la fusée. Elle peut, pour l'instant, tirer des projectiles mais rien d'autre. Il faut que nous puissions la faire avancer et tourner.
- Nous devons faire en sorte que la fusée explose lorsqu'elle est touchée par un astéroïde.
- Lorsque la fusée explose, nous voulons faire apparaître le score final à l'écran.
- Nous voulons pouvoir générer une vague protonique. Elle doit se former autour de la fusée, être de petite taille au départ et grandir petit à petit, détruisant les astéroïdes qu'elle rencontre.

Mais avant de commencer à programmer ces fonctions, nous allons commencer avec une petite chose d'ordre cosmétique : Disposer des étoiles dans notre univers.

7.2 Disposer les étoiles

Dans tous nos scénarios précédents, nous avons utilisé une image de fond fixe pour notre objet de la classe `World`. L'image était stockée dans un fichier sur notre disque.

Dans ce scénario, nous voulons introduire une technique différente pour créer des images de fond : les dessiner à la volée.

Le scénario Asteroid n'utilise pas de fichier pour stocker l'image de fond. Un monde d'un scénario Greenfoot qui n'a pas d'image de fond associée se voit attribuer par défaut une image de fond entièrement remplie de blanc.

Exercice 7.6

Cherchez dans le code source du constructeur de la classe `Space` du scénario quelles lignes de code génèrent le fond noir.

En observant le scénario asteroids-1, nous pouvons voir que le fond est complètement noir. Nous avons trouvé les trois lignes suivantes dans le code source :

```
GreenfootImage background = getBackground();
background.setColor(Color.BLACK);
background.fill();
```

Exercice 7.7

Enlevez ces trois lignes de code de votre classe. Pour le faire, il suffit de commenter ces lignes. Qu'observez-vous ? Veillez à décommenter les lignes ensuite.

La première ligne permet d'obtenir l'image courante associée à notre monde. Il s'agit de l'image blanche générée automatiquement par l'environnement Greenfoot. Nous avons alors une référence à l'image de fond de notre monde qui est stockée dans la variable `background`.

L'objet que nous avons stocké dans la variable est de type `GreenfootImage` ; nous avons déjà rencontré cette classe précédemment.

Exercice 7.8

Consulter la documentation de la classe `GreenfootImage`. Quel est le nom de la méthode utilisée pour dessiner un rectangle ? Quelle est la différence entre `drawOval` et `fillOval` ?

La deuxième ligne de l'extrait de code donné plus haut assigne à l'image `background` la couleur noire. Cette attribution n'a aucun effet immédiat, elle ne change pas la couleur de l'image. Elle détermine par contre la couleur qui sera utilisée dans toutes les opérations de dessin qui suivront. Le paramètre est une constante de la classe `Color`, que nous avons rencontrée au chapitre précédent.

Exercice 7.9

Regarder à nouveau la documentation de la classe `Color`. (Vous souvenez-vous de quel paquet elle fait partie ?) Pour combien de couleurs cette classe définit-elle des constantes ?

La troisième ligne de code de notre extrait permet le remplissage de notre image de fond avec

la couleur choisie. Notons le fait qu'il ne nous est pas nécessaire de spécifier à nouveau que cette image doit servir de fond à notre monde. Lorsque nous appelons la méthode `getBackground()`, nous obtenons une *référence* à l'image de fond et cette même image garde son statut d'image de fond. Elle n'est pas supprimée par le simple fait que nous stockons une référence à cette image dans une variable.

Lorsque nous dessinons sur cette image, nous dessinons directement sur le fond de notre monde.

Notre tâche est maintenant de dessiner un certain nombre d'étoiles sur cette image de fond.

Exercice 7.10

Dans la classe `Space`, créer une nouvelle méthode nommée `createStars`. Cette méthode doit avoir un paramètre de type `int`, nommé `number` permettant de spécifier le nombre d'étoiles à créer. La méthode n'a pas de valeur de retour. Le corps de la méthode reste pour l'instant vide.

Exercice 7.11

Écrire un commentaire pour cette nouvelle méthode. Le commentaire doit décrire ce que fait la méthode et expliquer à quoi sera utilisé le paramètre.

Exercice 7.12

Ajouter un appel à cette nouvelle méthode dans le constructeur de la classe `Space`. On peut pour l'instant se dire que 300 étoiles forment une quantité raisonnable. Libre à vous de modifier ce nombre par la suite jusqu'à l'obtention d'un résultat vous satisfaisant.

Exercice 7.13

Compiler la classe `Space`. À ce stade, vous ne devez observer aucun effet (notre nouvelle méthode ne comporte pas d'instructions), mais la compilation de la classe doit se faire sans message d'erreur.

Dans la méthode `createStars`, nous allons maintenant écrire du code permettant de dessiner un certain nombre d'étoiles sur l'image de fond. Le nombre de ces étoiles n'est autre que le nombre stocké dans le paramètre de la méthode.

Pour y parvenir, nous allons utiliser un autre type de boucle : la boucle `for`.

Nous avons déjà vu plus haut deux types de boucles : l'instruction `while` et l'instruction `for-each`. L'instruction `for` utilise le même mot-clef que l'instruction `for-each`, mais est structurée différemment. Voici cette structure :

```
for (initialisation; condition-de-boucle; incrémentation)
{
        instruction;
        ...
}
```

Un exemple de boucle `for` peut être observé dans la méthode `addAsteroids` de la classe `Space`.

Exercice 7.14

Examiner la méthode `addAsteroids` de la classe `Space`. Que fait-elle ?

Exercice 7.15

Observer la boucle `for` qui figure dans cette méthode. À partir de l'en-tête de la boucle, écrire les expressions correspondant à la partie `initialisation`, `condition-de-boucle` et `incrémentation` dans la définition de la structure de l'instruction donnée plus haut.

L'instruction correspondant à la partie `initialisation` d'une boucle `for` est exécutée une seule fois au départ de la boucle. La `condition-de-boucle` est ensuite testée : Si elle est vraie, le corps de la boucle est exécuté. Finalement, après que le corps de la boucle a été complètement exécuté, le programme passe à l'exécution de la partie `incrémentation` de l'en-tête de boucle.

Après cela, la boucle recommence : la condition est évaluée à nouveau et les instructions figurant dans le corps de la boucle sont exécutées à nouveau si la condition est vraie. Cela continue ainsi jusqu'à ce que l'évaluation de la condition de boucle nous renvoie `false`. L'instruction d'initialisation n'est jamais exécutée à nouveau.

Une boucle `for` peut être très facilement remplacée par une boucle `while` dont la structure est décrite ci-dessous :

```
initialisation;
while (condition-de-boucle)
{
        instruction;
        ...
        incrémentation;
}
```

Les deux instructions de boucle (`for` et `while`) font exactement la même chose. La différence principale réside dans le fait que l'initialisation et l'incrémentation font partie de l'en-tête de la boucle `for`. Cela permet de placer tous les éléments qui définissent le comportement de la boucle à un seul endroit, permettant une lecture plus facile de l'instruction de boucle.

L'instruction `for` est surtout utile lorsque nous connaissons déjà au début de la boucle le nombre de fois que nous voulons l'exécuter.

L'instruction `for` qui se trouve dans la méthode `addAsteroids` s'écrit

```
for (int i = 0; i < count; i++)
{
    int x = Greenfoot.getRandomNumber(getWidth()/2);
    int y = Greenfoot.getRandomNumber(getHeight()/2);
    addObject(new Asteroid(), x, y);
}
```

ce qui nous montre un exemple typique de boucle `for` :

- La partie d'initialisation déclare et initialise une variable de boucle. Cette variable s'appelle souvent `i`, et on l'initialise à 0 dans de nombreux cas.
- La condition de la boucle teste si notre variable de boucle est encore inférieure à une certaine limite, ici le contenu de la variable `count`. Tant que ce sera le cas, le corps de la boucle sera exécuté.

– La partie incrémentation ne fait qu'ajouter 1 à notre compteur.

Il existe différentes manières d'écrire une boucle `for`, mais cet exemple présente une manière de faire très courante.

Exercice 7.16

Dans votre classe `Space`, réécrire la boucle `for` de la méthode `addAsteroids` sous la forme d'une boucle `while`. Vérifier que la méthode produit le même résultat qu'avant la modification.

Exercice 7.17

Modifier la méthode `addAsteroids` à nouveau, pour la remettre dans son état initial.

Exercice 7.18

Implémenter le corps de la méthode `createStars` que vous avez créée plus haut. La méthode doit contenir les instructions permettant de

– placer dans une variable locale une référence à l'image de fond ;

– générer pour chaque étoile que l'on veut dessiner une paire de coordonnées `x` et `y`, choisir le blanc comme couleur de dessin et tracer un ovale de 2 pixels de large centré en `(x; y)` ;

– disposer dans l'espace un nombre d'étoiles égal au nombre passé en paramètre à la méthode en utilisant une boucle `for`.

Tester la méthode. Si des étoiles apparaissent dans votre monde, c'est que tout s'est bien passé.

Exercice 7.19

Créer des étoiles dont la luminosité est variable. On peut le faire en générant un nombre aléatoire compris entre 0 et 255 (l'intervalle des valeurs autorisées pour une couleur RGB) et créer un nouvel objet de type `Color` qui utilise ce nombre pour définir la valeur des trois composants : rouge, vert et bleu. Utiliser la même valeur pour tous les composants de la couleur garantit que le résultat est un gris neutre d'un certain niveau. On pourra alors utiliser cette nouvelle couleur pour dessiner nos étoiles. On prendra bien soin de créer une nouvelle couleur pour chaque étoile.

Ces exercices sont plutôt difficiles. Si vous êtes en difficulté, vous pouvez toujours vous inspirer de la solution. Une implémentation de tout ceci est fournie dans le scénario asteroids-2. Vous pouvez également, si vous le désirez, ignorer les exercices de ce paragraphe, passer à la suite et traiter ces problèmes plus tard.

7.3 Tourner

Dans le paragraphe précédent, nous avons fait de gros efforts pour améliorer l'aspect visuel de notre scénario. Nous avons utilisé une boucle `for` pour créer des étoiles apparaissant sur l'image de fond, un travail conséquent pour peu d'effet. Le jeu en valait la chandelle, toutefois : nous utiliserons la boucle `for` à de nombreuses reprises par la suite.

Nous voulons maintenant créer une fonction essentielle à notre programme : Nous voulons faire avancer notre fusée. Le premier pas dans cette direction est d'arriver à la faire tourner lorsque la flèche droite ou gauche du clavier subit une pression.

Exercice 7.20

Examiner la classe `Rocket`. Trouver le code qui gère les entrées du clavier. Quel est le nom de la méthode contenant ce code ?

Exercice 7.21

Ajouter une instruction qui fait tourner la fusée à gauche tant que la touche « flèche gauche » est enfoncée. Durant chaque cycle d'action, la fusée doit tourner de 5 degrés. On pourra utiliser les méthodes `getRotation` et `setRotation` de la classe `Actor` pour obtenir le résultat escompté.

Exercice 7.22

Ajouter une instruction qui fait tourner la fusée vers la droite lorsque la touche « flèche droite » est enfoncée. Tester le fonctionnement !

Si vous avez réussi à faire ces exercices, vous devriez maintenant pouvoir faire tourner votre fusée en appuyant sur les flèches appropriées. Vu que la fusée tire dans la direction dans laquelle elle pointe, elle peut tirer dans toutes les directions.

La tâche suivante est de la faire avancer.

7.4 Voler vers l'avant

Notre classe `Rocket` est une sous-classe de la classe `SmoothMover`, que nous avons déjà vu au chapitre précédent. Cela implique que la fusée dispose d'un vecteur qui permet de contrôler son mouvement, et qu'elle a une méthode `move()` qui la fait se déplacer selon son vecteur de mouvement.

Notre première étape est de faire usage de cette méthode `move()`.

Exercice 7.23

Dans la méthode `act` de la classe `Rocket`, ajouter un appel à la méthode `move()`, héritée de la classe `SmoothMover`. Tester. Que peut-on observer ?

Ajouter l'appel à la méthode `move()` dans notre méthode `act` est un premier pas important, mais ne produit pas grand-chose en lui-même. Cet appel de méthode fait se déplacer la fusée selon son vecteur de mouvement, mais comme nous n'avons pas donné de valeur initiale à ce vecteur, il a une longueur nulle pour l'instant, et donc il ne se produit aucun déplacement.

Pour changer cela, introduisons pour commencer une petite quantité de dérive automatique, de sorte à ce que la fusée se déplace un peu dès le départ. Cela rend le scénario plus intéressant à jouer, car cela empêche le joueur de rester immobile longtemps.

Exercice 7.24

Ajouter une petite quantité de mouvement initiale à la fusée à l'aide de son constructeur. Pour le faire, créer un nouveau vecteur de petite taille (j'ai utilisé 0.3 pour ma version du scénario) dont la direction est arbitraire. Utiliser ensuite la méthode `addForce` de la classe `SmoothMover` en lui passant ce vecteur en paramètre pour ajouter cette force à la fusée. On prendra bien soin d'utiliser un entier comme premier paramètre du constructeur de `Vector` pour faire appel au bon constructeur ;

on se souvient en effet du fait que le constructeur de la classe `Vector` est surchargé.

Tester. Si tout s'est bien passé, la fusée devrait dériver d'elle même au lancement du scénario. Ne rendez-pas cette dérive trop importante. Faites le nombre d'essais nécessaires à l'obtention d'un mouvement initial convenablement lent.

Nous voulons ajouter ensuite le contrôle du mouvement par le joueur. Le plan est le suivant : une pression sur la flèche « haut » déclenche le booster de la fusée et la fait partir vers l'avant.

Pour faire tourner la fusée, nous avons utilisé les instructions suivantes :

```
if (Greenfoot.isKeyDown("left"))
{
        setRotation (getRotation() - 5);
}
```

Pour le mouvement vers l'avant il nous faut une structure légèrement différente. La raison en est que, pour la rotation telle que présentée ci-dessus, nous ne devons agir que si une touche est enfoncée.

La gestion du mouvement vers l'avant est une autre affaire : Lorsque nous appuyons sur la « flèche vers le haut », nous voulons changer l'image de la fusée pour montrer que le réacteur s'allume. Lorsque nous relâchons la pression sur la touche, il faudra encore que l'image de la fusée revienne à la normale. Nous devons donc structurer notre suite d'instructions comme suit :

```
lorsque la flèche haut est enfoncée:
        changer l'image pour montrer que le réacteur s'enclenche;
        ajouter le mouvement;

lorsque la flèche haut est relâchée:
        remettre l'image normale;
```

Changer d'image est relativement simple, vu que le scénario contient déjà deux images différentes, prévues à cet effet : rocket.png et rocketWithThrust.png. Ces deux images sont déjà chargées dans deux champs en haut du code de la classe `Rocket`.

Vu que nous devons réagir dans deux cas, lorsque le joueur enfonce la touche « haut » et lorsqu'elle ne subit aucune pression, nous allons définir et appeler deux méthodes séparées pour gérer le contrôle de cette fonction de jeu.

Dans la méthode `checkKeys`, nous pouvons insérer l'appel de méthode suivant :

```
ignite (Greenfoot.isKeyDown("up"));
```

Nous devrons ensuite écrire une méthode appelée `ignite` ayant les propriétés suivantes :

- Elle reçoit en paramètre un booléen (disons `boosterOn`) qui indique si le booster doit être enclenché ou non.
- Si la valeur du paramètre est `true`, l'image courante devient rocketWithThrust.png et un appel à la méthode `addForce` permet l'ajout d'un vecteur au mouvement de la fusée. Ce vecteur devrait avoir la même direction que la fusée que l'on obtient à l'aide de `getRotation` et devrait avoir une longueur fixe (disons 0.3).

– Si la valeur du paramètre est `false`, l'image doit être rocket.png.

Exercice 7.25

Ajouter l'appel à la méthode `ignite` à votre méthode `checkKeys` exactement comme indiqué ci-dessus.

Exercice 7.26

Définissez un squelette de méthode (une méthode dont le corps est vide) pour la méthode `ignite`. Cette méthode doit prendre un paramètre booléen et son type de valeur de retour doit être `void`. Prenez bien soin d'écrire un commentaire. Tester ! Le code doit pouvoir être compilé sans erreur, mais il n'y a pas pour l'instant de résultat visible.

Exercice 7.27

Implémentez la méthode `ignite` comme suggéré ci-dessus.

Dans cette implémentation, on observe le recours systématique à la méthode `setImage` chaque fois que la méthode `ignite` est appelée, même lorsque cela n'est pas nécessaire : si le booster est éteint et qu'il était également éteint lors de l'appel précédent de la méthode `act`, il n'est pas nécessaire d'appeler `setImage` vu que l'image n'a pas changé. Cela ne pose pas de problème dans la mesure ou le coût de l'opération est faible ; il n'est donc pas crucial de chercher à éviter cet appel systématique.

Une fois les exercices de ce paragraphe terminés, vous avez atteint le stade suivant : il vous est possible de faire voler votre fusée dans toutes les directions de l'espace et de tirer sur les astéroïdes.

La solution des exercices présentés dans ce chapitre jusqu'ici est donnée par le scénario asteroids-2 qui se trouve dans le dossier des scénarios du livre.

7.5 Collisions avec les astéroïdes

Le défaut le plus évident de notre jeu spatial à ce stade est que nous pouvons passer au travers des astéroïdes sans que rien ne se passe. Nous ne pouvons donc pas perdre, pour l'instant, ce qui ne rend pas le jeu très palpitant. Nous allons remédier à cela maintenant.

Il faut que notre fusée explose lorsqu'elle entre en collision avec un astéroïde. Si vous avez fait les exercices précédents, vous avez déjà constaté que nous disposons dans notre scénario d'une classe `Explosion` pleinement fonctionnelle. Il suffit de placer une explosion quelque part dans l'espace pour obtenir un effet convaincant.

On peut alors donner une description grossière de la tâche à résoudre :

```
If (nous avons percuté un astéroïde)
    {
        éliminer la fusée de l'espace;
        disposer une explosion à sa place;
        montrer le score final (game over);
    }
```

Avant de considérer la résolutions des trois tâches ci-dessus, nous allons préparer notre code source pour faciliter l'implémentation, comme nous l'avons fait plus haut dans le chapitre, pour d'autres fonctions. Nous suivons la même stratégie que précédemment : écrire le code de chaque tâche particulière dans une méthode qui lui est propre. Cela permet de structurer le code et d'en améliorer la lisibilité. Il faudrait commencer l'implémentation de toute nouvelle fonction ainsi. L'exercice qui suit permet de réaliser cela.

Exercice 7.28

Créer le squelette d'une nouvelle méthode (le corps de la méthode doit être vide) dans la classe `Rocket` pour détecter les collisions avec les astéroïdes. Nommer cette méthode `checkCollision`. Cette méthode peut être privée, n'a pas de valeur de retour et n'attend pas de paramètre.

Exercice 7.29

Dans la méthode `act` de la classe `Rocket`, ajouter un appel à la méthode `checkCollision`. Vérifiez que tout se passe bien en compilant et en exécutant le scénario.

La première tâche à réaliser est de détecter si nous avons percuté un astéroïde. La classe `Actor` de Greenfoot contient plusieurs méthodes différentes permettant de détecter les collisions qui fonctionnent sur divers modèles. L'appendice C présente un résumé de ces différentes méthodes de détection de collision et expose brièvement leur mode de fonctionnement. C'est peut-être le bon moment pour jeter un coup d'oeil à ces différentes méthodes. À partir d'un certain stade, il vous faudra être familiarisé avec toutes les méthodes de détection de collision.

Les méthodes `getIntersectingObjects` ou `getOneIntersectingObject` semblent convenir plutôt bien pour ce qui nous occupe. Ces méthodes détectent l'intersection de deux objets sitôt qu'ils ont un de leurs pixels en commun. C'est tout à fait ce dont nous avons besoin.

Il y a toutefois un petit problème causé par les éventuels pixels transparents de nos images.

Les images de Greenfoot sont toujours des rectangles. Lorsque nous voyons des images qui semblent non rectangulaires, comme notre fusée, c'est à cause de certains pixels de l'image qui sont *transparents* (invisibles, ils ne contiennent aucune couleur). En ce qui concerne le programme, ils sont toutefois toujours part intégrante de l'image.

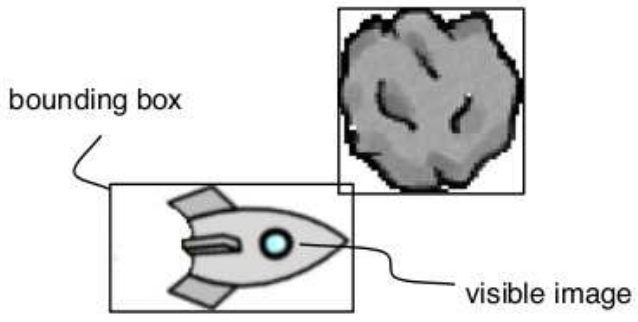


FIGURE 9 – Deux images d'acteurs et leurs « bounding box »

La figure ci-dessus nous montre deux images qui s'intersectent malgré le fait que leurs parties visibles ne se touchent pas. On voit que chaque image est en réalité dans un rectangle, sa « bounding box ». Le contour rectangle de cette boîte est la frontière de l'image. On observe

ici que l'image de la fusée est un peu plus grande que nécessaire vers l'arrière. La raison en est la suivante : il faut que l'image de la fusée soit de la même taille que l'image rocketWithThrust qui va remplir le vide arrière avec des flammes.

Il nous faut donc être conscients du fait que les méthodes mentionnées ci-dessus nous annoncent une collision alors que subsiste à l'écran une petite distance entre les objets.

Pour le jeu des astéroïdes, nous choisissons d'ignorer ce problème. Tout d'abord parce que les distances qui nous concernent ici sont faibles et que donc les joueurs ne se rendront probablement compte de rien. D'autre part, il est facile d'inventer une rationalisation à ce comportement : voler trop près d'un astéroïde détruira le vaisseau à cause de la force de gravitation.

Il est parfois indispensable de pouvoir tester si les parties visibles (non-transparentes) de deux objets s'intersectent. C'est possible, mais bien plus difficile. Nous ne discuterons pas de cela ici.

Reste à choisir entre les deux méthodes de la classe `Actor` que nous avons mentionnées plus haut. Leurs signatures respectives sont :

```
List getIntersectingObjects (Class cls)

Actor getOneIntersectingObject (Class cls)
```

On doit passer aux deux méthodes un paramètre de type `Class`, ce qui veut dire que nous pouvons tester l'intersection avec un type d'objet donné, si nous le désirons. La différence entre ce deux méthodes est que l'une renvoie la liste courante des objets avec lesquels nous avons une intersection non vide, alors que l'autre ne renvoie qu'un seul objet de type `Actor`. Dans le cas où l'intersection compte plus d'un objet, la seconde méthode choisit aléatoirement un objet dans la liste et le renvoie.

La seconde méthode nous suffit pour parvenir à nos fins. Il ne fait en effet, pour notre jeu, aucune différence si nous percutons un, deux ou plusieurs astéroïdes simultanément. La fusée explosera de la même façon. La seule question à nous poser est la suivante : avons-nous percuté un quelconque astéroïde ?

Nous allons donc utiliser la seconde méthode. Dans la mesure où elle renvoie un objet de type `Actor` plutôt qu'une collection de type `List`, elle est un peu plus simple à utiliser que l'autre. Elle va nous renvoyer un acteur si l'intersection est non vide et `null` dans le cas contraire. Nous pouvons tester si la valeur de retour est `null` pour savoir si nous avons percuté un astéroïde :

```
Actor actor = getOneIntersectingObject (Asteroid.class);

if (actor != null)
{
        ...
}
```

Exercice 7.30

Ajouter à votre méthode `checkCollision` un test comme ci-dessus permettant de détecter les intersections avec les astéroïdes.

Exercice 7.31

Ajouter dans le corps de l'instruction `if` de l'exercice précédent le code qui fait apparaître une explosion dans l'espace à l'endroit occupé par la fusée à ce moment et qui enlève ladite fusée de notre monde. Il faudra penser à utiliser la méthode `getWorld()` pour pouvoir utiliser les méthodes permettant d'ajouter et d'enlever des objets.

Pour le dernier des deux exercices ci-dessus, nous pouvons utiliser les méthodes `getX()` et `getY()` de la fusée pour obtenir la position courante de celle-ci. Nous employons ensuite ces deux coordonnées pour placer l'explosion.

Une tentative de résolution pourrait ressembler à l'extrait de code ci-dessous :

```
World world = getWorld();
world.removeObject(this);
world.addObject(new Explosion(), getX(), getY());
```

À première vue, ce code semble raisonnable, mais il provoquera une erreur.

Exercice 7.32

Tester le code ci-dessus, tel qu'il est écrit. Peut-on compiler le code sans erreur ? Le cas échéant, peut-on faire exécuter le résultat par la machine virtuelle ? À quel moment survient le problème et quelle est la nature du message d'erreur ?

La raison pour laquelle le code ci-dessus provoque une erreur à l'exécution est la suivante : nous faisons appel aux méthodes `getX()` et `getY()` *après* avoir supprimé la fusée du monde ; et, lorsqu'un acteur est enlevé du monde, il perd ses coordonnées qui n'existent que lorsqu'il est partie intégrante du monde. C'est pourquoi l'appel aux méthodes `getX()` et `getY()` conduit ici à l'échec.

Pour remédier à cela, rien de plus simple : il suffit d'intervertir les deux dernières ligne de l'extrait de code ci-dessus. On commence par placer l'explosion et on enlève la fusée ensuite.

Exercice 7.33

Il s'agit d'un exercice difficile que vous pouvez sans autre laisser de côté dans un premier temps. Vous pourrez y revenir plus tard.

L'explosion utilisée ici est une explosion plutôt simple. Elle suffit pour l'instant, mais si vous voulez créer des jeux plus élaborés visuellement, elle peut-être améliorée. Une manière plus sophistiquée de programmer des explosions est présentée dans une vidéo d'un tutoriel Greenfoot, que l'on trouve sur le site suivant :

`http://www.greenfoot.org/doc/videos.html`

Créer une explosion de ce genre pour votre scénario.

7.6 Casting

Notre jeu commence à devenir intéressant. Vous avez probablement constaté que le compteur de score ne fait rien pour l'instant (nous nous occuperons de ce fait plus tard) et que rien ne se

passe hormis l'explosion et la disparition de la fusée lorsque l'on perd. Nous allons donc ajouter maintenant une grande inscription « GameOver » qui apparaît au moment où la fusée explose.

Cela est relativement facile à faire: Il y a déjà une classe `ScoreBoard` prévue dans notre scénario et nous allons l'utiliser sans autre.

Exercice 7.34

Créez un objet de la classe `ScoreBoard` et placez-le dans l'espace.

Exercice 7.35

Examinez le code source de la classe `ScoreBoard`. Combien de constructeurs y trouve-t-on ? En quoi diffèrent-ils ?

Exercice 7.36

Examinez le code source de la classe `ScoreBoard` : Changez le texte qui s'affiche sur le panneau ; changez la couleur du texte ; changez la couleur du fond et celle du cadre ; changez la taille de la police de caractère de sorte à ce que votre texte soit d'une taille adaptée ; changez la largeur du panneau du score de sorte à ce qu'il convienne à votre texte.

Comme vous avez pu le constater, le panneau du score contient le texte « Game Over » et le score final, même si le score n'est pour l'instant pas décompté correctement (nous nous occuperons de cela plus tard).

En l'ouvrant dans l'éditeur, on peut constater que la classe `Space` dispose déjà du squelette d'une méthode `gameOver` qui devra s'occuper de créer et d'afficher un objet de la classe `ScoreBoard`.

Exercice 7.37

Trouvez et examinez le code de la méthode `gameOver` de la classe `Space`. Que fait cette méthode, pour l'instant ?

Exercice 7.38

Dans la classe `Space`, implémenter la méthode `gameOver`. Elle devrait créer un nouvel objet de type `ScoreBoard`, en utilisant le constructeur qui prend un paramètre entier. Pour l'instant, nous utiliserons 999 comme score et nous programmerons le décompte du score plus tard. Placer le panneau du score exactement au centre de la fenêtre du monde.

Exercice 7.39

Une fois implémentée, testez votre méthode `gameOver`. Souvenez-vous : vous pouvez appeler les méthodes des classes des mondes en faisant un clic-droit dans le monde.

Exercice 7.40

Comment être vraiment sûr que le panneau du score est placé au centre du monde sans coder « en dur » les coordonnées (c'est à dire, sans utiliser les nombres 300 et 200 directement comme coordonnées) ? On pourra faire usage de la hauteur et de la largeur du monde. Faites-le pour votre propre scénario.

Bien. Il semble qu'une grande partie du travail a été préparée pour nous avec soin. Nous n'avons maintenant plus qu'à appeler la méthode `gameOver` lorsque nous voulons que le jeu s'arrête.

L'endroit de notre code où nous désirons voir le jeu s'arrêter se trouve dans la méthode `checkCollision` de la classe `Rocket` : Si nous détectons une collision, la fusée doit exploser (sur ce plan là, c'est en ordre) et le jeu doit se terminer.

Exercice 7.41

Ajouter un appel à la méthode `gameOver` dans votre méthode `checkCollision`. Compiler. Que peut-on observer ? Vous vous heurtez certainement à un message d'erreur ; lequel ?

Le simple fait d'ajouter un appel à la méthode `gameOver` crée un problème. Ce problème est fondamental et il mérite un examen approfondi. Voyons le code à ce stade en supposant que nous nous sommes bornés à ajouter un appel à la méthode `gameOver` dans le code existant de la méthode `checkCollision`. Ce code est rejeté par le compilateur.

```
private void checkCollision()
{
    Actor actor = getOneIntersectingObject (Asteroid.class);

    if (actor != null)
    {
        World world = getWorld();
        world.addObject(new Explosion(), getX(), getY());
        world.removeObject(this);
        world.gameOver();
    }
}
```

Lorsque nous essayons de compiler ce code, nous obtenons le message d'erreur suivant :

```
Cannot find symbol - method gameOver()
```

Ce message nous indique que le compilateur ne peut pas trouver de méthode portant ce nom. Nous savons pourtant qu'une telle méthode existe dans notre classe `Space`. Nous savons aussi que l'appel à la méthode `getWorld()` nous donne une référence pointant sur notre objet `Space`. Alors, quel est le problème ?

Le problème réside dans le fait que le compilateur n'est aussi futé que nous l'aimerions. La méthode `getWorld()` est définie dans la classe `Actor` et sa signature est la suivante :

```
World getWorld()
```

L'en tête nous indique que la méthode nous renvoie un objet de type `World`. Dans les faits, l'objet qu'elle nous retourne est d'un type plus spécifique : c'est un objet de type `Space`.

Cela n'est pas une contradiction. Notre monde peut être de type `World` et de type `Space` en même temps, vu que la classe `Space` est une sous-classe de `World` (`Space` *is a* `World` ; on dira aussi que le type `Space` est une *sous-catégorie* du type `World`).

L'erreur provient de la différence suivante : `gameOver` est définie dans la classe `Space`, alors que `getWorld` nous donne un résultat de type `World`. Le compilateur ne tient compte que du type de valeur de retour de la méthode que nous sommes en train d'appeler (`getWorld`). À cause de cela, le compilateur ne cherche la méthode `gameOver` qu'à cet endroit, et il ne la trouve pas. C'est pourquoi nous obtenons le message d'erreur.

Pour résoudre ce problème, nous devons indiquer explicitement au compilateur que ce monde qui nous obtenons est en fait de type `Space`. Nous pouvons le faire en utilisant une opération de *cast*.

```
Space space = (Space) getWorld();
```

Le *casting* est la technique qui consiste à donner explicitement au compilateur un type plus précis que celui qu'il peut déterminer par lui-même pour notre objet. Dans notre cas, le compilateur peut déterminer que l'objet retourné par `getWorld` est de type `World`, et nous lui précisons qu'il appartient en fait à la classe `Space`. Nous le faisons en écrivant le nom de la classe (`Space`) entre parenthèses avant l'appel de méthode. Après cela, nous pouvons appeler les méthodes définies dans la classe `Space` :

```
space.gameOver();
```

Il faut noter que l'opération de cast ne change pas le type de l'objet auquel elle s'applique. Notre monde est de type `Space` indépendamment de tout casting. Le problème est que le compilateur n'est pas conscient de ce fait. Le casting nous permet de fournir une information supplémentaire au compilateur.

Revenons à notre méthode `checkCollision`. Une fois que nous avons effectué notre opération de cast sur notre monde et que nous avons stocké le résultat dans une variable de type `Space`, nous pouvons appeler toutes les méthodes de notre objet : celles définies dans `Space` et celles définies dans `World`. Ainsi, les appels déjà écrits aux méthodes `addObject` et `removeObject` sont toujours valables et l'appel à `gameOver` ne provoquera plus de message d'erreur.

Exercice 7.42

Implémenter l'appel à la méthode `gameOver` en utilisant le casting de l'objet `World` en un objet de type `Space`, comme exposé ci-dessus. Tester. Tout devrait maintenant fonctionner et le panneau du score devrait s'afficher au moment de l'explosion de la fusée.

Exercice 7.43

Que se passe-t-il si l'on utilise l'opérateur de cast à mauvais escient ? Essayer de « caster » l'objet monde en un objet de la classe `Asteroid` à la place de `Space`. Cela fonctionne-t-il ? Qu'observez-vous ?

Le travail effectué jusque là nous permet d'afficher notre panneau « game Over », avec un score incorrect, toutefois. Nous laisserons l'exercice permettant d'afficher un score correct pour la fin du chapitre. Si vous voulez vraiment vous en occuper maintenant, vous pouvez passer maintenant aux exercices de la fin du chapitre. En ce qui nous concerne, nous allons nous occuper de la vague protonique maintenant.

7.7 Ajouter de la puissance de feu : La vague protonique

Notre jeu s'est bien amélioré. La dernière chose dont nous discuterons en détail dans ce chapitre est l'ajout d'une seconde arme : la « vague protonique ». Cela devrait permettre d'ajouter un peu de variété à notre jeu. L'idée est la suivante : Notre vague protonique une fois lancée, rayonne autour de notre fusée, endommageant ou détruisant tous les astéroïdes se trouvant sur son chemin. Vu qu'elle balaye toutes les directions simultanément, c'est une arme beaucoup plus performante qu'un projectile. Pour que le jeu ne devienne pas trop facile à jouer, nous devrons probablement restreindre le nombre ou la fréquence des utilisations de la vague protonique.

Exercice 7.44

Placez une vague protonique et lancez l'exécution de votre scénario. Qu'observez-vous ?

L'exercice ci-dessus nous montre que nous disposons d'un acteur `protonWave` dont l'image est celle d'une vague de taille maximale. Toutefois, cette vague ne grandit pas, ne bouge pas, ne disparaît pas et ne provoque aucun dommage aux astéroïdes.

Notre première tâche sera de faire grandir la vague protonique. Nous commencerons par afficher une vague de toute petite taille et nous la ferons grandir ensuite jusqu'à ce qu'elle atteigne sa taille maximale que nous avons déjà pu observer.

Exercice 7.45

Examinez le code source de la classe `ProtonWave`. Quelles sont ses méthodes existantes ?

Exercice 7.46

Quel est la fonction de chacune de ces méthodes ? Reprendre le commentaire de chaque méthode et l'augmenter pour que l'explication donnée soit plus détaillée.

Exercice 7.47

Essayez d'expliquer ce que la méthode `initializeImages` fait et comment cela fonctionne. Donner l'explication sous forme de texte ou de diagrammes, si cela vous convient mieux.

7.8 Faire grandir la vague

Nous avons vu la classe `ProtonWave` dispose d'une méthode `initializeImages` qui permet la création de 30 images de tailles différentes et qui les stocke dans un tableau. Ce tableau, nommé `images`, contient l'image la plus petite à l'emplacement indexé par 0, et la plus grande à l'emplacement 29, comme illustré ci-dessous :

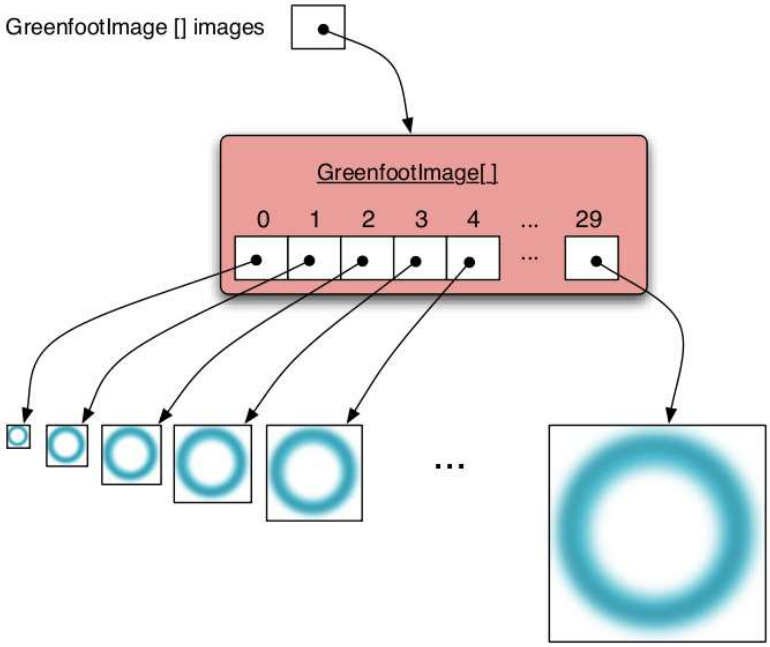


FIGURE 10 – Le tableau d'images

Les images sont créées en chargeant une image de base (wave.png) et en générant ensuite le nombre voulu de copies de cette image à l'aide d'une boucle `while`. Observons dans l'extrait de code ci-dessous le changement du facteur d'échelle à chaque itération de la boucle.

```
public static void initializeImages()
{
    if(images == null)
    {
        GreenfootImage baseImage = new GreenfootImage("wave.png");
        images = new GreenfootImage[NUMBER_IMAGES];
        int i = 0;
        while (i < NUMBER_IMAGES)
        {
            int size = (i+1) * ( baseImage.getWidth() / NUMBER_IMAGES );
            images[i] = new GreenfootImage(baseImage);
            images[i].scale(size, size);
            i++;
        }
    }
}
```

La méthode `initializeImages` emploie la méthode `scale` de la classe `GreenfootImage` pour faire la mise à l'échelle. Elle utilise également un boucle `while` pour remplir le tableau. Il s'agit toutefois d'un exemple dans lequel la boucle `for` que nous avons rencontrée au début du chapitre conviendrait parfaitement.

Exercice 7.48

Réécrire la méthode `initializeImages` en utilisant une boucle `for` plutôt qu'une boucle `while`.

En pratique, le choix de l'instruction de boucle n'est pas très important ici ; nous vous l'avons fait changer surtout pour vous permettre d'exercer l'écriture de l'instruction `for`. C'est pourtant un cas dans lequel l'instruction `for` représente le choix approprié, car nous connaissons le nombre d'itérations (c'est le nombre d'images) et nous pouvons faire usage du compteur de boucle pour calculer la taille des images. Ici, l'avantage sur la boucle `while` réside dans le fait que tous les éléments de la boucle sont placés ensemble dans l'en-tête de la boucle (initialisation, condition et incrément), et donc nous courrons moins le risque d'en oublier l'une ou l'autre.

Le champ `images` et la méthode `initializeImages` sont statiques (on utilise le mot-clef `static` dans leur définition). Comme nous l'avons brièvement mentionné au chapitre 1, cela signifie que le champ `images` est stocké dans la *classe* `ProtonWave` et non dans les instances individuelles. En conséquence, tous les objets que nous créerons à partir de cette classe partageront cet ensemble d'images et il ne sera pas nécessaire de créer un ensemble d'images de plus pour chaque objet supplémentaire. C'est bien plus efficace que d'utiliser un ensemble d'images à chaque fois différent.

La copie et la mise à l'échelle de ces images durent assez longtemps (entre un dixième de seconde et une demi-seconde sur les ordinateurs couramment utilisés aujourd'hui). Cela ne semble pas très long, mais ça l'est suffisamment pour introduire un délai perceptible et ennuyeux lorsque cela se fait durant le milieu d'une partie. Pour résoudre ce problème, le code de la méthode se trouve à l'intérieur d'une clause `if` :

```
if (images == null)
{
        ...
}
```

Cette instruction `if` assure que la partie principale de la méthode (le corps du `if`) ne s'exécute qu'une seule fois. À la première exécution, le tableau `images` vaudra `null` et la méthode sera complètement exécutée. Cela nous initialisera notre champ `images` à autre chose que `null`. À partir de ce point, le test de la condition `if` sera la seule instruction qui sera exécutée et le corps de l'instruction sera ignoré. La méthode `initializeImages` sera appelée chaque fois qu'une vague protonique sera créée par le constructeur, mais le travail substantiel ne sera fait qu'une seule fois. [1]

Maintenant que nous avons une idée assez claire de ce qui concerne le code source et les champs

1. La méthode est en fait appelée pour la première fois par le constructeur de la classe `Space`, et s'exécute donc même avant que la première vague protonique ne soit créée. Cela évite également le délai pour la création de cette première vague. L'appel qui est inclus dans le constructeur de la classe `ProtonWave` est une simple mesure de sécurité : Si cette classe est utilisée une fois dans un autre projet et que cette méthode n'est pas appelée à l'avance, tout fonctionnera tout de même.

déjà existants, nous pouvons finalement nous mettre au travail et faire en sorte que quelque chose se produise.

Ecrivons les différents points que aimerions voir se réaliser :

- Nous voulons que la vague montre la plus petite image possible au moment de sa création.
- À chaque itération de la méthode `act`, nous voulons faire grandir la vague (montrer l'image du tableau qui suit immédiatement l'image courante).
- Nous voulons enfin que la vague protonique disparaisse (soit supprimée du monde), après avoir atteint la plus grande taille possible.

Les exercices ci-dessous permettront d'obtenir tout cela.

Exercice 7.49

Dans le constructeur de la classe `ProtonWave`, affecter la plus petite image possible à la vague protonique. Vous pourrez passer `images[0]` en paramètre à la méthode `setImage`.

Exercice 7.50

Créer une variable d'instance nommé `imageCount` de type `int`, et initialisez-le à 0. Nous utiliserons ce champ comme compteur d'image. La valeur courante sera l'indice de l'image qui est affichée à ce moment là.

Exercice 7.51

Écrire le squelette d'une nouvelle méthode appelée `grow`. Cette méthode ne prendra aucun paramètre et ne renverra aucune valeur non plus.

Exercice 7.52

Appeler la méthode `grow` depuis la méthode `act` de la classe `ProtonWave` (même si elle ne fait rien pour l'instant).

Nous y sommes presque. La seule chose qui reste à faire et l'implémentation de la méthode `grow`. Grossièrement, l'idée est la suivante :

```
montrer l'image correspondant à l'indice imageCount;
incrémenter imageCount;
```

Nous devrons également ajouter une instruction conditionnelle `if` qui permettra de vérifier si `imageCount` n'a pas dépassé le nombre d'images moins un. Dans ce cas, il suffit de supprimer la vague protonique du monde et nous en avons terminé.

Exercice 7.53

Implémenter la méthode `grow` en suivant les indications données ci-dessus.

Exercice 7.54

Testez votre vague protonique. Si vous créez une vague de manière interactive et que vous la placer dans le monde durant l'exécution du scénario, vous pourrez constater l'effet d'expansion de l'image.

Exercice 7.55

Ajoutez du son. Un fichier son nommé proton.wav est inclus dans le scénario ; il suffit de le faire jouer. Vous pouvez placer l'instruction qui fait jouer le son dans le constructeur de la classe `ProtonWave`.

Maintenant que nous disposons d'une vague protonique fonctionnelle, nous allons faire en sorte que la fusée en soit armée.

Exercice 7.56

Dans la classe `Rocket`, créer le squelette d'une méthode nommée `startProtonWave` sans paramètres. Doit-elle avoir une valeur de retour ?

Exercice 7.57

Implémenter cette méthode : elle doit placer un nouvel objet de type `ProtonWave` dans l'espace, là où se trouve la fusée à ce moment.

Exercice 7.58

Dans la méthode `CheckKeys`, écrire un appel conditionnel à la méthode `startProtonWave`. La condition est réalisée lorsque la touche « Z » est enfoncée. Tester.

Exercice 7.59

Vous constaterez très rapidement que la vague protonique peut être lancée beaucoup trop souvent maintenant. Un système de contrôle de la cadence de tir a déjà été intégré à la fusée pour le tir des projectiles (il utilise la constante `gunReloadTime` et l'attribut `reloadDelayCount`). Étudiez ce code et implémentez quelque chose de similaire pour la vague protonique. Testez différentes valeurs permettant d'espacer les émissions de la vague protonique jusqu'à trouver un intervalle qui convienne.

7.9 Interagir avec des objets dans un certain rayon

Nous disposons maintenant d'une vague protonique que nous pouvons émettre en pressant une touche, mais il nous reste un problème à régler : Cette vague protonique ne fait rien aux astéroïdes.

Nous voulons maintenant ajouter le code qui permet d'endommager les astéroïdes touchés par la vague protonique.

Exercice 7.60

Préparation pour cette nouvelle fonction : Dans la classe `ProtonWave`, ajouter le squelette d'une méthode appelée `checkCollision`. La méthode n'a ni paramètre, ni valeur de retour. Appeler cette méthode depuis la méthode `act`.

Exercice 7.61

Le but de cette nouvelle méthode est de tester si la vague touche un astéroïde et de l'endommager si c'est le cas. Rédiger le commentaire de la méthode.

Cette fois, nous ne pourrons pas utiliser la méthode `getIntersectingObjects`, dans la mesure où les parties invisibles situées au coins de l'image de la vague protonique sont plutôt grandes et que les astéroïdes seraient détruits bien avant que la vague semble les atteindre, si nous utilisions cette méthode.

Nous allons devoir utiliser la méthode `getObjectsInRange` qui permet de détecter les collisions d'une autre façon.

La méthode `getObjectsInRange` nous retourne une liste d'objets situés dans un rayon donné par rapport au centre de l'objet depuis lequel la méthode a été appelée :

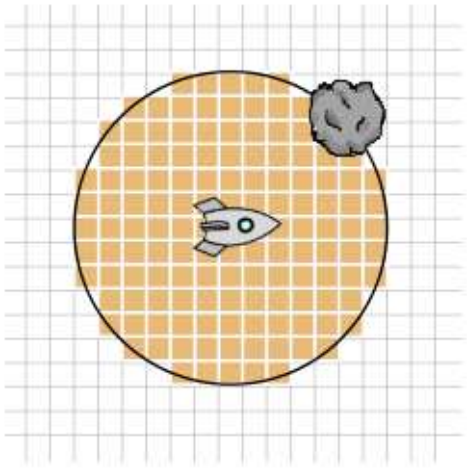


FIGURE 11 – Zone de détection d'un rayon donné

```
List getObjectsInRange (int radius, Class cls)
```

Lors de l'appel de méthode, on peut spécifier la classe des objets que l'on cherche à détecter comme pour les autres méthodes ; on doit par contre spécifier un rayon (en nombre de cellules). Cette méthode nous retournera ensuite une liste de tous les objets de la classe concernée qui se trouvent à l'intérieur du « disque » déterminé par le rayon.

Pour déterminer quels objets sont dans la zone circulaire, les centres des objets sont utilisés. Par exemple, un astéroïde sera détecté par l'appel de méthode

```
getObjectsInRange(20, Asteroid.class)
```

si la distance entre son centre et le centre de la fusée représente moins de 20 cellules de largeur. En ce qui concerne cette méthode, la taille de l'image importe peu.

Nous pouvons maintenant implémenter notre méthode `checkCollision` à l'aide de cette technique de détection.

Notre vague protonique aura des images dont la taille devient de plus en plus grande. Durant chaque cycle d'action, nous pouvons utiliser la taille de l'image courante pour faire déterminer le rayon à passer en paramètre à notre test de collision. Nous pouvons trouver la taille de l'image courante en appelant les méthodes suivantes :

```
getImage().getWidth()
```

Nous utilisons ensuite la moitié de cette taille pour notre rayon, vu qu'il s'agit précisément du rayon et non du diamètre...

Exercice 7.62

Dans `checkCollision`, déclarer une variable locale nommée `range` et lui attribuer la moitié de la taille courante de l'image.

Exercice 7.63

Ajouter un appel à la méthode `getObjectsInRange` qui retourne tous les astéroïdes qui sont dans la zone correspondant au contenu de la variable `range`. Assigner le résultat à une variable de type `List<Asteroid>`. N'oubliez pas d'ajouter l'instruction d'importation pour le type `List`.

Après ces deux exercices, nous disposons d'une liste de tous les astéroïdes qui sont à portée de notre vague protonique. Nous voulons maintenant endommager chacun de ces astéroïdes.

La classe `Asteroid` dispose d'une méthode nommée `hit` que nous pouvons utiliser pour réaliser cela. Cette méthode est déjà utilisée pour faire du dégât lorsque l'astéroïde est touché par un projectile de type `Bullet`, et nous pouvons l'utiliser ici à nouveau.

Nous allons utiliser une boucle `for-each` pour parcourir la liste d'astéroïdes que nous a fournie l'appel à la méthode `getObjectsInRange`. (Si vous ne vous sentez pas très à l'aise pour l'écriture d'une boucle de ce genre, il vous faut relire le paragraphe 6.8).

Exercice 7.64

Trouver la méthode `hit` dans la classe `Asteroid`. Quels sont ses paramètres ? Que retourne-t-elle ?

Exercice 7.65

La classe `ProtonWave` a une constante définie en haut de la classe et appelée `DAMAGE`, qui spécifie combien de dommage elle doit causer. Trouver la déclaration de cette constante. Quelle est sa valeur ?

Exercice 7.66

Dans la méthode `checkCollision`, écrire une boucle `for-each` qui parcourt la liste d'astéroïdes obtenue grâce à l'appel de la méthode `getObjectsInRange`. Dans le corps de la boucle, appeler `hit` pour chaque astéroïde en utilisant la constante `DAMAGE` pour quantifier le dommage causé.

Une fois que vous avez terminé ces exercices, tester la méthode. Si tout s'est bien passé, vous devriez avoir maintenant une version jouable de ce jeu qui vous permette non seulement de tirer sur les astéroïdes, mais encore d'émettre des vagues protoniques détruisant d'un seul coup plusieurs astéroïdes. Vous observerez que vous devrez mettre un délai d'attente assez long entre les émissions de deux vagues protoniques successives, sous peine de rendre le jeu inintéressant parce que trop facile.

Cette version du jeu, incluant tous les changements faits dans les derniers paragraphes, se trouve dans le dossier des scénarios du livre, sous le nom asteroids-3. Vous pouvez utiliser ce scénario pour comparer vos solutions à un « corrigé » ou pour vous aider si vous êtes bloqués dans un exercice.

7.10 Pour aller plus loin

Nous voici à la fin de la discussion détaillée du développement de ce scénario dans ce chapitre. Il y a toutefois un grand nombre d'améliorations possibles à ce jeu. Il y en a qui sont plutôt évidentes, et d'autres que vous aurez envie d'inventer par vous-même.

Vous trouverez ci-dessous des suggestions pour aller plus loin dans le travail de ce scénario. Beaucoup d'entre elles ont indépendantes les unes des autres ; peu importe l'ordre dans lequel vous les réaliserez. Choisissez d'abord celles qui vous intéressent le plus et produisez également quelques améliorations de votre cru.

Exercice 7.67

Mettez en place le décompte du score. Vous avez déjà constaté l'existence du compteur de score, mais il n'est pas encore utilisé. Le compteur est défini dans la classe `Counter`, et un objet de ce type est créé dans la classe `Space`. En gros, vous allez devoir mener à bien la tâche suivante : Ajouter une méthode à la classe `Space` appelée, disons, `countScore` ; elle devra ajouter un score donné au compteur de score. Il faudra ensuite appeler cette méthode depuis la classe `Asteroid` chaque fois qu'un astéroïde est touché (vous pourriez ajouter des quantités différentes au score si l'astéroïde est en train de se casser ou que l'on est en train d'éliminer les derniers petits morceaux).

Exercice 7.68

Ajouter de nouveaux astéroïdes lorsqu'il n'y en a plus. Commencez peut-être le jeu avec deux astéroïdes seulement et faites en réapparaître un de plus chaque fois que les astéroïdes ont tous été détruits. Après le premier tour, il apparaît trois astéroïdes, quatre après le deuxième tour, etc.

Exercice 7.69

Ajouter un compteur de niveau. Chaque fois que les astéroïdes ont tous été éliminés, vous montez un niveau. Vous obtenez peut-être de plus hauts scores dans les niveaux élevés, où la difficulté est plus grande.

Exercice 7.70

Ajouter un son de fin de niveau. Ce son devra être joué chaque fois qu'un niveau se termine.

Exercice 7.71

Ajouter un indicateur montrant l'état de recharge de la vague protonique, de sorte à ce que le joueur puisse voir quand elle sera prête à être lancée à nouveau. L'indicateur pourrait se présenter sous la forme d'un compteur, ou être représenté graphiquement d'une certaine façon.

Exercice 7.72

Ajouter un bouclier. lorsque le bouclier est déployé, il reste en place durant un petit intervalle de temps fixe. Lorsque le bouclier est présent, il est visible à l'écran et la fusée peut entrer en collision avec les astéroïdes sans dommage.

Il y a bien sûr un nombre incalculable d'extensions possibles. Inventez-en quelques uns, implémentez-les est publiez-les dans la « Greenfoot Gallery ».

7.11 Résumé des techniques de programmation

Dans ce chapitre, nous avons travaillé à compléter un jeu de destruction d'astéroïdes qui était à moitié écrit au départ. En le faisant, nous avons à nouveau rencontré plusieurs éléments de programmation que nous avions déjà vu auparavant, incluant les boucles, les listes et les détections de collisions.

Nous avons vu un nouveau style de boucle, la boucle `for` et nous l'avons utilisée pour dessiner des étoiles et générer les images de la vague protonique. Nous avons aussi utilisé à nouveau l'instruction `for-each` lorsque nous avons mis en place les effets de la vague protonique.

Deux méthodes de détection de collision ont été utilisées : `getOneIntersectingObject` et `getObjetsInRange`. Les deux ont leurs avantages dans certaines situations. La seconde nous renvoyant une liste d'acteurs, nous avons du traiter les listes à nouveau.

Comprendre les concepts de boucle et de liste est plutôt difficile au départ, mais très important en programmation ; vous devriez donc revoir attentivement ces aspects de votre code si vous n'êtes pas encore à l'aise pour ce qui est de leur emploi. Plus vous pratiquerez ces concepts, plus vous aurez de la facilité à les manipuler et à programmer en général. Après avoir employé ces instructions durant quelques temps, vous constaterez que vous ne voyez plus en quoi elles représentent de la difficulté.