Programmation orientée objet avec Greenfoot

Table des matières

Chapitre 1

Apprendre à connaître Greenfoot

1.1 Pour commencer

Faisons démarrer l'application Greenfoot et ouvrons le scénario `leaves-and-wombats`. Dans la fenêtre qui s'est ouverte, on peut observer :

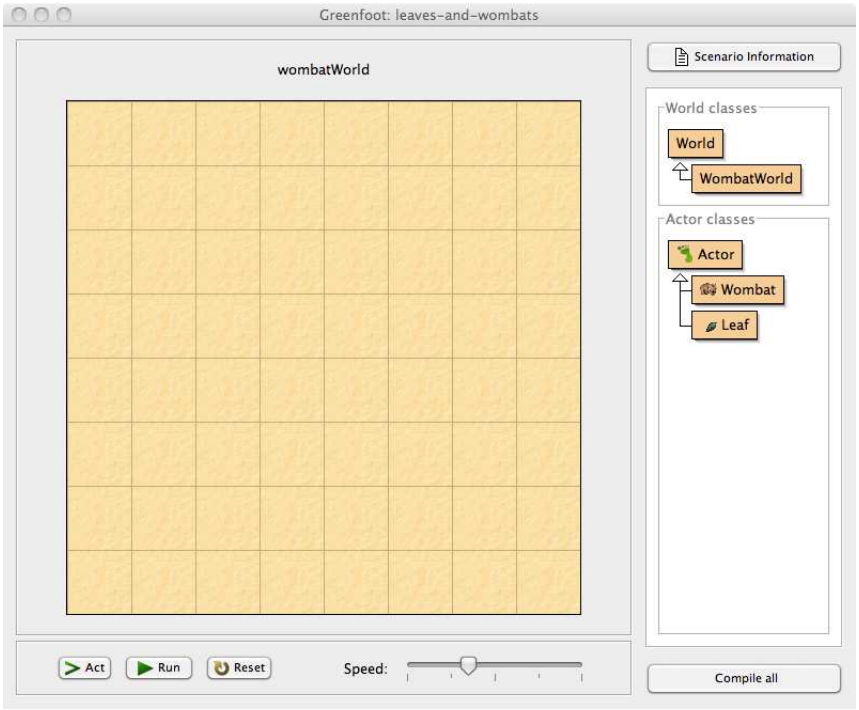


FIGURE 1.1 – La fenêtre principale de Greenfoot

- 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.

1.2 Objets et classes

Commençons par dire quelques mots du diagramme de classes que l'on voit sur la droite de la figure 1.1. On peut y observer les classes `World`, `WombatWorld`, `Actor`, `Leaf` et `Wombat`. 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

A 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.

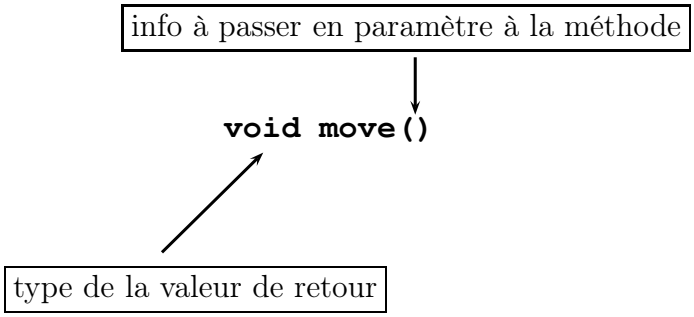
Dans Greenfoot, on peut :

- 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* ou la *signature* 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 ?

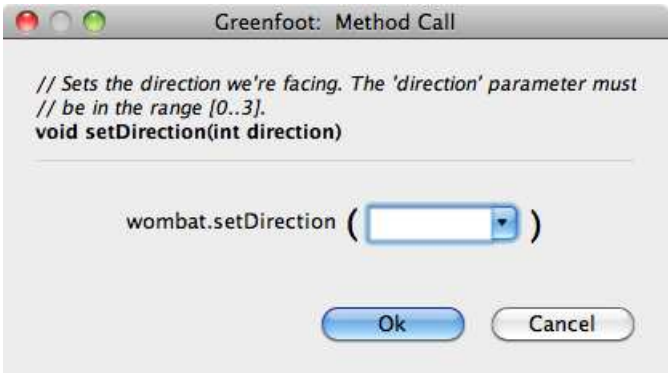
Les méthodes qui ont un type de valeur de retour différent de `void` ne font en général que nous donner de l'information sur un objet (*Peut-il bouger ? Combien de feuilles a-t-il mangées ?*), mais ne modifient pas cet objet. Le wombat est resté le même après la question sur le nombre de feuilles consommées. Les méthodes sans valeur de retour (de type `void`) sont en général des ordres donnés à l'objet pour lui faire faire quelque chose.

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` et `direction` écrits à la suite l'un de l'autre 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

A 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.

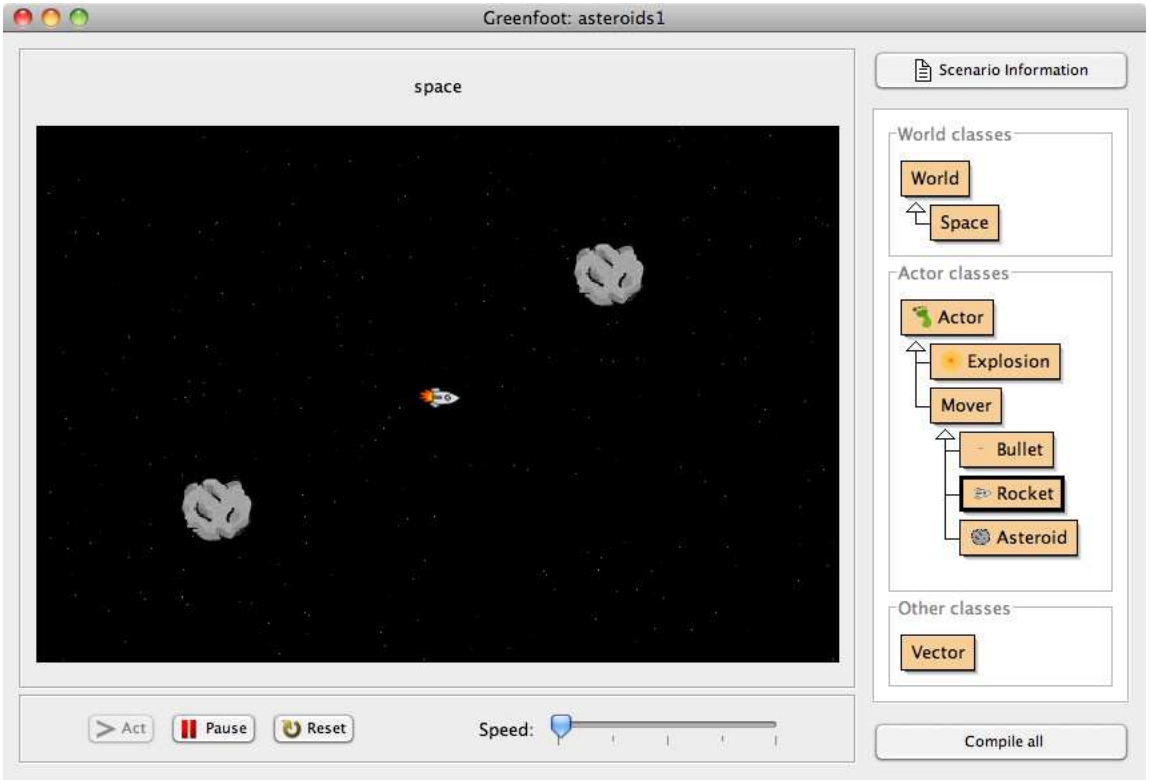
b) S'il n'y a rien à manger dans la case et que la voie est libre, le wombat avance d'une case.

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.

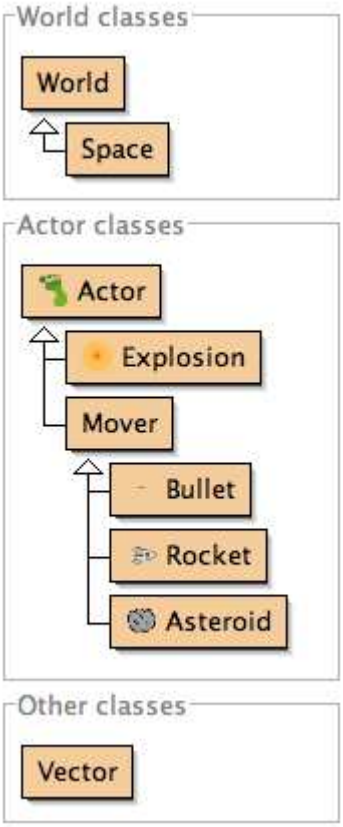
1.7 Un deuxième exemple

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.
A 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 n'allons 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

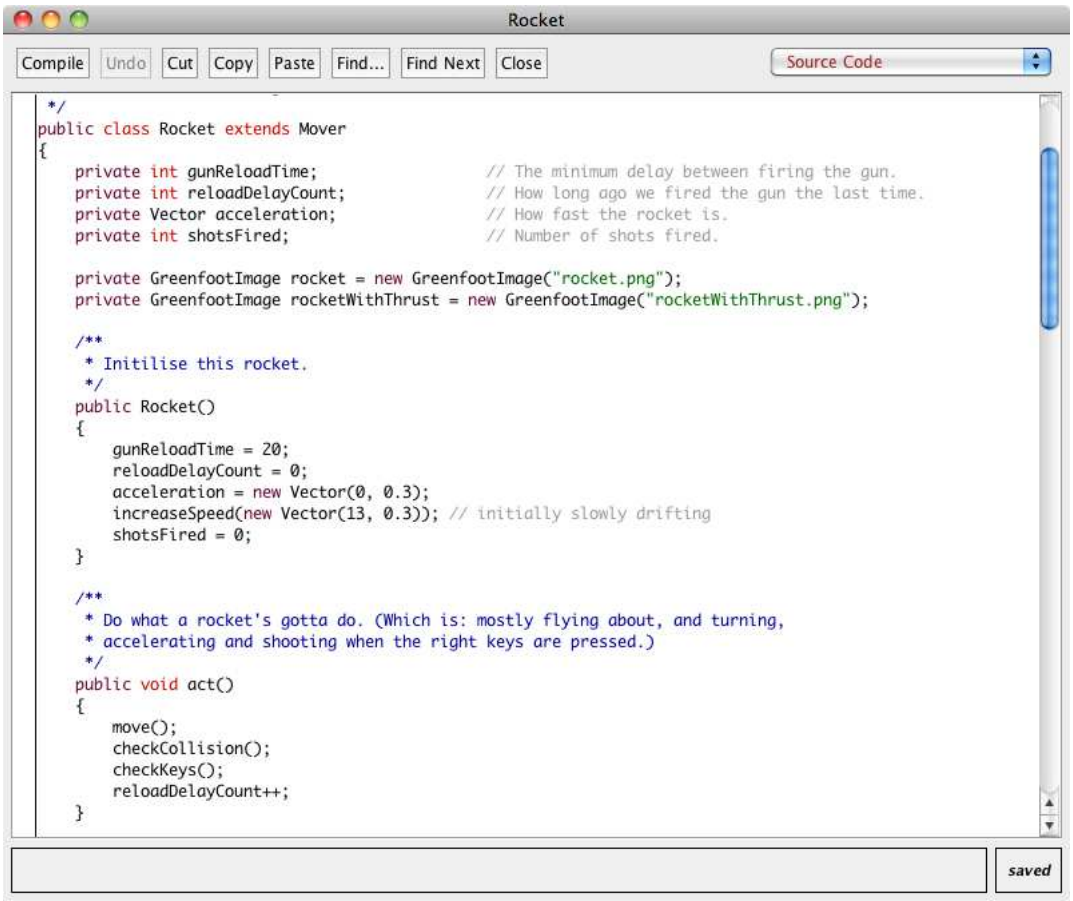
Les astéroïdes ont une `stabilité` qui leur est propre. Chaque fois qu'ils sont touchés par un projectile, leur stabilité décroît. Lorsqu'elle atteint zéro, ils cassent. Quelle est la valeur de la stabilité d'un astéroïde au moment de sa création ? De combien d'unités la stabilité d'un astéroïde diminue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



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.
A 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)

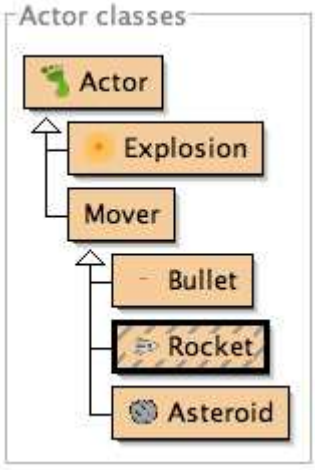
et de façon à ce que toutes les nouvelles fusées créées par la suite aient ce comportement amélioré.
Ouvrez l'éditeur à partir du menu contextuel de la classe `Rocket`. A environ 25 lignes du haut du texte, vous devriez trouver la ligne suivante :

```
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 des techniques de programmation

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.

1.12 Concepts du chapitre

- Les scénarios de Greenfoot sont constitués de *classes*.
- De nombreux *objets* peuvent être créés à partir d'une *classe*.
- Les objets ont des *méthodes* qui peuvent être invoquées pour faire agir les objets.
- Le *type de la valeur de retour* d'une méthode indique ce que la méthode renvoie après qu'elle a été appelée.
- Une méthode dont le type de valeur de retour est `void` ne renvoie pas de valeur.
- Les méthodes de type `void` représentent des *commandes* ; les autres représentent des *questions*.
- Un *paramètre* est un mécanisme qui permet de passer des données supplémentaires à une méthode.
- Les paramètres et les valeurs de retour ont un *type*. Un exemple de type de nombre est le type `int`. Pour une valeur prenant uniquement les valeurs `true` ou `false`, on dispose du type `boolean`.
- La description d'une méthode par type de valeur de retour, nom et liste de paramètres s'appelle *en-tête* ou *signature* de cette méthode.
- Les objets que l'on peut placer dans le monde s'appellent des *acteurs*. Ils sont créés à partir d'une sous-classe de la classe `Actor`.
- Une *sous-classe* est une classe qui représente une spécialisation d'une autre classe. Dans le diagramme de classe de Greenfoot, la relation qui lie une sous-classe avec la classe dont elle hérite est représenté par une flèche.
- Toute classe est définie par du *code source*. Ce code détermine ce qu'un objet de cette classe peut faire. Dans Greenfoot, on peut accéder au code source d'une classe en ouvrant l'éditeur de cette classe.
- Les ordinateurs ne comprennent pas le code source. Il doit être traduit en langage machine avant d'être exécuté. Cela s'appelle la *compilation*.

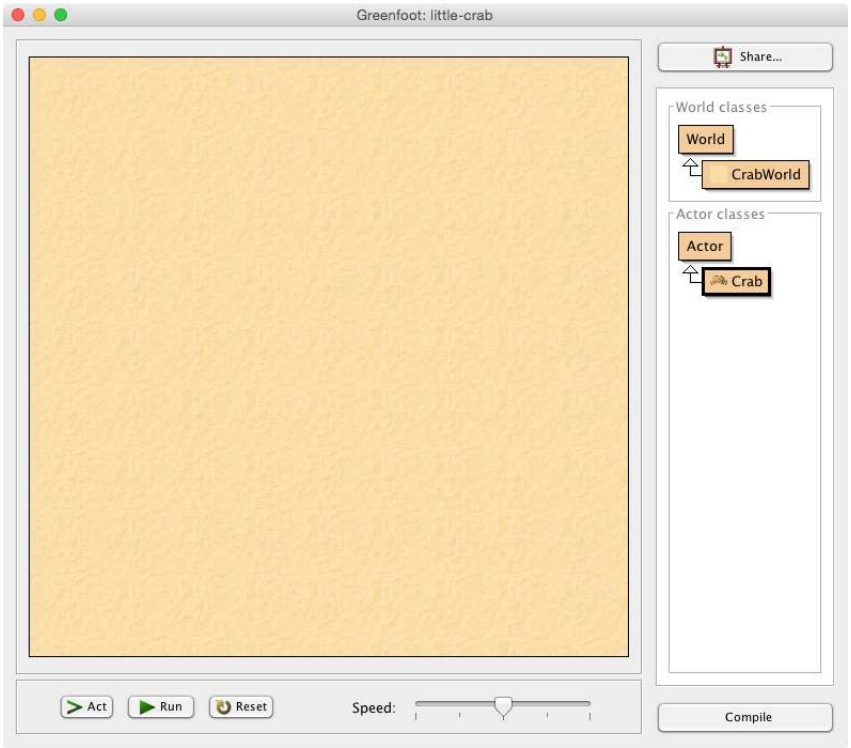
Chapitre 2

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 ? (Rappelez-vous que si les

icônes de classes situées sur la droite sont hachurées, c'est que vous devez compiler le scénario d'abord.)

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 les usuelles classes `World` et `Actor` de Greenfoot, ainsi que deux sous-classes appelées `CrabWorld` et `Crab`.
La hiérarchie, figurée par les flèches, dénote une relation *is-a*, appelée également *héritage*: Un crabe *est un* acteur et le monde du crabe `CrabWorld` *est un* monde.
Nous travaillerons tout d'abord seulement avec la classe `Crab`. Nous parlerons un peu plus loin des classes `Actor` et `CrabWorld`.
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.*;

/**
 * This class defines a crab. Crabs live on the beach.
 */
public class Crab extends Actor
{
    public void act()
    {
        // Add your action code here
    }
}
```

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(5);
```

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(5);
}
```

Exercice 2.2

Modifier la méthode `act` de la class `Crab` de façon à y inclure l'instruction

```
move(5);
```

comme décrit ci-dessus. Compiler le scénario en cliquant sur le bouton Compile et placer un crabe dans le monde. Cliquer ensuite sur les boutons Act et Run.

Exercice 2.3

Remplacer le nombre 5 par un autre autres. Tester avec des nombres plus grands et plus petits. Que signifie ce nombre, à votre avis ?

Exercice 2.4

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(5)` 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 l'instruction `move(5);` que nous avons écrite à l'intérieur de la méthode `act` s'exécute. Le nombre cinq figurant dans cette instruction désigne la distance que le crabe va franchir à chaque exécution de l'instruction, soit ici 5 pixels (vers la droite).
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.

Exercice 2.5

Pouvez-vous trouver un moyen de faire bouger le crabe à l'envers (vers la gauche) ?

Terminologie

L'instruction `move(5);` 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 et le nombre qui est à l'intérieur 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(3);
```

Le nombre 3 dans l'instruction spécifie de combien de degrés le crabe doit tourner. Cela s'appelle un *paramètre*. (Le nombre 5 utilisé pour l'appel à la méthode `move` plus haut est aussi 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(5)` par `turn(3)`. (Les valeurs des paramètres, 5 et 3 dans ce cas, sont choisies plutôt arbitrairement ; vous pouvez choisir d'autres valeurs.) La méthode `act` aura alors l'allure suivante :

```
public void act()
{
    turn(3);
}
```

Exercice 2.6

Remplacer `move()` par `turn(3)` 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 2.7

Comment faire pour que le crabe tourne dans l'autre sens, soit vers la gauche ?

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.*;

/**
 * This class defines a crab. Crabs live on the beach.
 */
public class Crab extends Actor
{
    public void act()
    {
        move(5);
        turn(3);
    }
}
```

Exercice 2.8

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

Terminologie

Le nombre placé entre parenthèses dans l'instruction `turn` – c'est à dire, le 3 de `turn(3)` – s'appelle un *paramètre*. Un paramètre est une information supplémentaire que nous devons fournir lorsque nous appelons certaines méthodes.
Certaines méthodes n'attendent aucun paramètre. Nous les appelons en écrivant le nom de la méthode suivie d'une paire de parenthèses sans rien à l'intérieur, par exemple `stop()`. D'autres méthodes, comme `turn` ou `move` demandent de l'information en plus : *De combien dois-je tourner ? De quelle distance dois-je avancer ?* Dans ce cas, nous devons fournir cette information sous la forme d'un paramètre à l'intérieur des parenthèses, `turn(17)` ou `move(11)`, 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(5)`, l'environnement Greenfoot émettra un message d'erreur lors d'un essai de compilation.

Exercice 2.9

Ouvrir l'éditeur pour voir le code source du crabe et ôter le point-virgule après `move(5)`. 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 2.10

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 ?

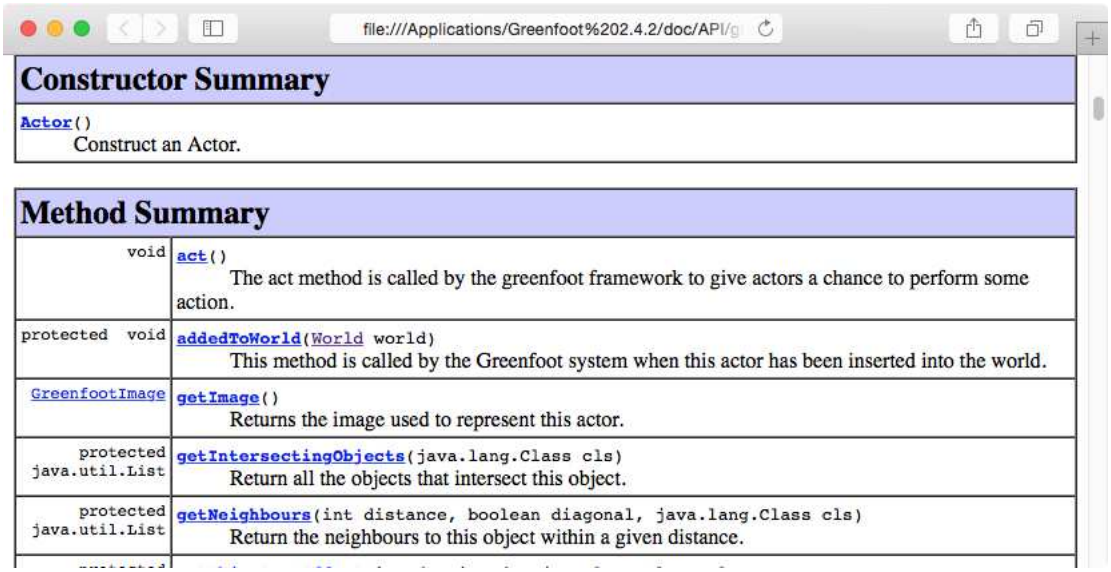


FIGURE 2.1 – La documentation de la classe `Actor`

Les méthodes `move` et `turn` que nous avons utilisées jusqu'à présent proviennent de la classe `Actor`. Un crabe est un acteur, comme indiqué par la flèche qui va de la classe `Crab` à la classe `Actor` dans le diagramme de classes ; il peut donc faire tout ce qu'un acteur fait en général. Un acteur décrit par la classe `Actor` 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 `Actor`.
Une question se pose maintenant : que peut faire d'autre notre acteur ?
Pour découvrir cela, nous pouvons ouvrir la classe `Actor`. Vous constaterez que lorsqu'on double clique l'icône de la classe `Actor` elle ne s'ouvre pas dans un éditeur texte comme la classe `Crab`, mais présente de la documentation dans un navigateur Web (voir figure 2.1). La raison de ce fait est que la classe `Actor` fait partie des classes prédéfinies de Greenfoot ; elle ne peut pas être modifiée. Mais nous pouvons tout de même employer les méthodes de la classe `Actor` et les appeler. La documentation nous dit quelles méthodes existent, quels sont leurs éventuels paramètres et ce qu'elles font. (On peut aussi consulter la documentation de nos autres classes en utilisant le menu déroulant qui se trouve dans le coin en haut à droite de la fenêtre de l'éditeur.)

Exercice 2.11

Ouvrir la documentation de la classe `Actor`. Trouver la liste des méthodes de la classe (« Method summary »). De combien de méthodes dispose cette classe ?

Exercice 2.12

Parcourez la liste de toutes les méthodes à disposition. Pouvez-vous en trouver une qui pourrait nous être utile pour déterminer si nous sommes « au bord de notre monde » ?

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

`boolean isAtEdge()`

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

`void move(int distance)`

Déplace cet acteur de la distance spécifiée dans la direction à laquelle il fait face.

`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 ; nous avons déjà rencontré ce concept 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 : on doit fournir à ces deux méthodes un seul paramètre de type `int` (un nombre entier). Pour la méthode `move`, c'est la distance dont il faut se déplacer et pour la méthode `turn`, c'est la quantité dont il faut tourner.
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. Le crabe se bornera à obéir mais ne nous donnera aucune réponse.
L'en tête de la méthode `isAtEdge()` est un peu différent :

```
boolean isAtEdge()
```

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 `isAtEdge()`, 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 2.13

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

Exercice 2.14

Faire se déplacer le crabe jusqu'au bord du monde à l'aide du bouton Run ou manuellement, et ensuite appeler la méthode `isAtEdge()` 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.*;
/**
* This class defines a crab. Crabs live on the beach.
*/
public class Crab extends Actor
{
    public void act()
    {
        if (isAtEdge())
        {
            turn(17);
        }
        move(5);
    }
}
```

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 (isAtEdge())
{
```

```
        turn(17);
    }
```

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

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

A la place de `condition`, on peut mettre toute expression qui est vraie ou fausse (telle notre appel de la méthode `isAtEdge()`), 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(5)` 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 2.15

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 2.16

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

Exercice 2.17

Placer l'instruction `move(5)` 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

Si vous observez le code source dans l'éditeur de texte de Greenfoot, vous remarquerez les rectangles colorés qui forment le fond de la fenêtre de l'éditeur, derrière les lignes de code. Ces rectangles marquent la *portée* d'une structure de code Java. Dans Greenfoot, on a donné différentes couleurs à différentes structures : par exemple, un rectangle vert marque le code associé à une classe, une méthode se verra colorée en jaune et l'instruction `if` est marquée par un gris-mauve pâle. Vous pouvez constater que les rectangles colorés, tout comme les instructions correspondantes, peuvent être imbriqués les uns dans les autres : une instruction `if` se trouve dans une méthode, une méthode est dans une classe.
Prêter attention à la coloration de la portée des structures de code est rentable à long terme ; cela peut vous aider à éviter des erreurs « classiques ». L'un de ces « rectangles de portée »

est en général défini dans votre code par une paire d'accolades (avec en général un en-tête qui précède l'accolade ouvrante qui définit quelle structure de code est associée à ce rectangle coloré). Il est courant de se tromper dans le nombre d'accolades ouvrantes ou fermantes, c'est à dire d'en avoir trop d'ouvrantes ou trop de fermantes. Dans ce cas de figure le code ne compilera pas.
La « coloration de portée » peut vous aider à détecter ce genre de problème. Vous apprendrez très rapidement à reconnaître à quoi doivent ressembler les rectangles de portée d'un programme écrit correctement et vous remarquerez que lorsque des accolades sont dépareillées, la coloration devient bizarre.
Avec la coloration de portée vient naturellement l'indentation du code.
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 décalées 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. L'éditeur de Greenfoot peut aussi vous aider si votre indentation n'est plus du tout correcte : La fonction Auto-Layout du menu Edit permet de tenter de réparer l'indentation de toute une classe.
Il est très important de faire attention à l'indentation de votre 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.

Exercice 2.18

Ouvrez le code source de votre classe `Crab`. Enlevez diverses accolades fermantes et ouvrantes et observez le changement dans la coloration de portée. Dans chacun de ces cas, pouvez-vous expliquer le changement de couleur ? Faites également quelques modifications dans l'indentation des accolades et celle du code en général et observez en quoi cela modifie l'aspect de votre programme.
Pour terminer l'exercice, replacez les accolades et corrigez l'indentation de votre code de façon à ce que votre programme soit à nouveau plaisant à regarder.

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 les méthodes `move(5)` ou `isAtEdge()`), 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 à l'existence du type `boolean`, une valeur qui peut être `true` ou `false`.

2.6 Concepts du chapitre

- Un *appel de méthode* est une instruction qui dit à un objet de réaliser une action. L'action est définie par une méthode de cet objet.
- De l'information supplémentaire peut être passée à aux méthodes ; cela se fait à l'intérieur des parenthèses qui font toujours partie de l'appel de méthode. La ou les valeurs placées dans les parenthèses s'appelle un ou des *paramètres*.
- Une suite d'instructions est exécutée de manière *séquentielle*, c'est à dire une instruction après l'autre, dans l'ordre dans lequel elles sont écrites.
- Lorsqu'une classe est compilée, le compilateur vérifie le code pour y détecter d'éventuelles erreurs. Si une erreur est trouvée, un *message d'erreur* est montré.
- Une sous-classe *hérite* de toutes les méthodes de sa classe mère ou superclasse. Cela signifie que la sous-classe dispose de toutes les méthodes définies dans la classe mère et qu'on peut les appeler depuis la sous-classe.
- Appeler une méthode *sans valeur de retour* (le type de valeur de retour est alors `void`) revient à faire exécuter une commande. Appeler une méthode *avec une valeur de retour* (le type de valeur de retour n'est pas `void` dans ce cas) revient à poser une question qui attend une réponse explicite.
- Une *instruction* `if` peut être utilisée pour écrire des instructions qui ne sont exécutées que lorsqu'une certaine condition est réalisée.

Chapitre 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 intéressants. 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 en premier lieu faire en sorte que certains acteurs de notre scénario présentent un 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 ( isAtEdge() )
        {
            turn(17);
        }

        if ( Greenfoot.getRandomNumber(100) < 10 )
        {
            turn(4);
        }

        move(5);
    }
}
```

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 4 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 4.

```
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

Tester 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.

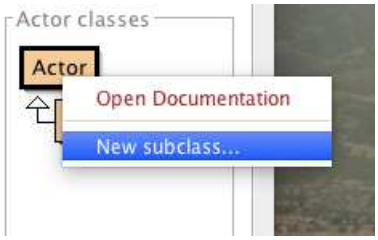
Exercice 3.7

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 sous-classe de la classe `Actor`. Souvenez-vous du fait que la relation qui lie une sous-classe avec sa classe est une relation *is a* : a worm *is an* actor.
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, comme sur la figure 3.1. 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

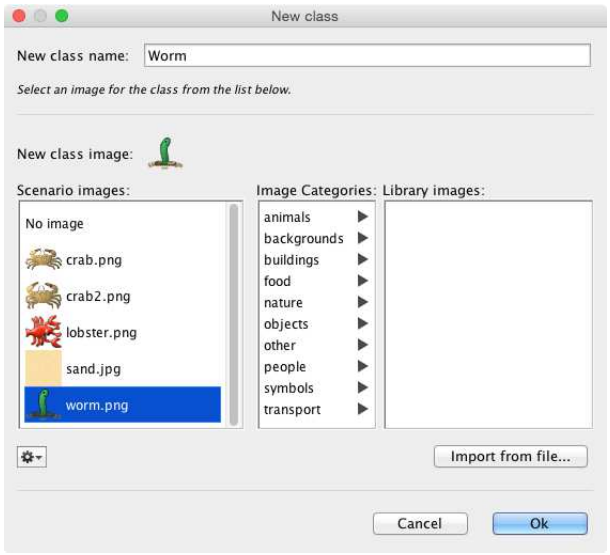


FIGURE 3.1 – Dialogue de création d'une sous-classe

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. A nouveau, nous consultons la liste des méthodes de la classe `Actor` pour savoir de quelles méthodes le ver a hérité directement de cette super-classe. Nous ouvrons donc la documentation de la classe `Actor` et nous voyons qu'il existe les deux méthodes suivantes :

> `boolean isTouching (java.lang.Class clss)`
> Check whether this actor is touching any other objects of the given class.
>
> `void removeTouching (java.lang.Class clss)`
> Remove one object of the given class that this actor is currently touching (if any exist).

Nous pouvons implémenter le type de comportement cherché en utilisant ces deux méthodes. La première teste si le crabe touche un ver. Cette méthode nous renvoie un

`boolean` (`true` ou `false`) que nous pouvons utiliser dans une instruction conditionnelle `if`. La seconde méthode permet d'éliminer 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 ( isTouching(Worm.class) )
{
    removeTouching(Worm.class);
}
```

Nous avons passé `Worm.class` en paramètre aux deux appels de méthode (`isTouching` et `removeTouching`). 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 ( isAtEdge() )
    {
        turn(17);
    }

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

    move(5);

    if ( isTouching(Worm.class) )
    {
        removeTouching(Worm.class);
    }
}
```

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 `isTouching` et `removeTouching`, 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 de nouveaux comportements à notre crabe : tourner sur soi-même 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 ( isTouching(Worm.class) )
    {
        removeTouching(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 ( isAtEdge() )
    {
        turn(17);
    }

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

    move(5);
    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 ( isTouching(Worm.class) )
    {
        removeTouching(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(5);
    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 `Actor`, 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 les 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 permet 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

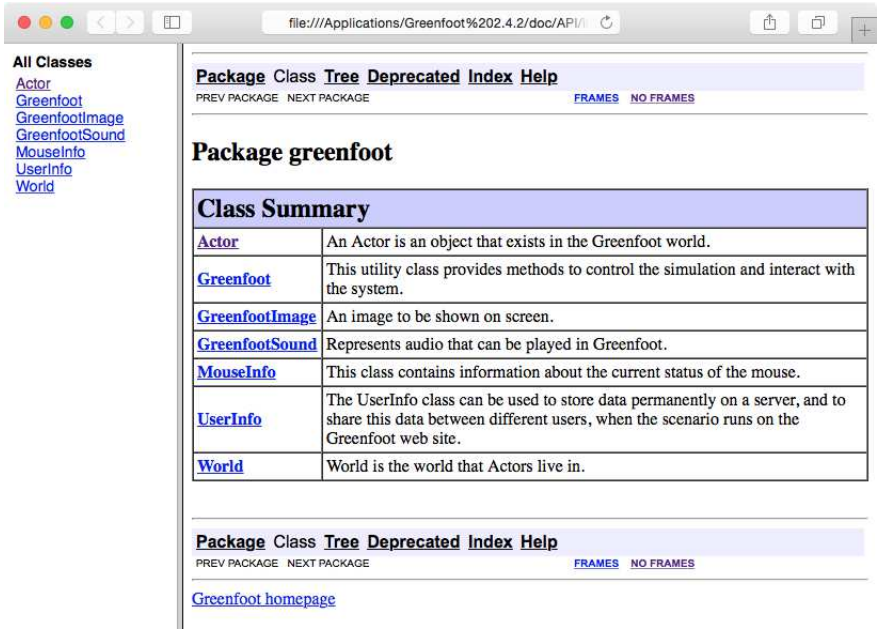
Si vous ne l'avez pas fait spontanément, faites en sorte que le code qui teste l'état de la touche « flèche gauche » ne soit pas écrit directement dans la méthode `act`, mais bien dans une méthode séparée s'appelant `checkKeypress`, par exemple. Un appel à cette méthode devra être présent dans la méthode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

A nouveau, choisissez Greenfoot Class Documentation depuis le menu Help afin d'ouvrir la documentation des classes de Greenfoot. Il y a une classe `GreenfootSound` pour contrôler les effets sonores, et également une méthode de la classe `Greenfoot` prévue pour nous simplifier la tâche. Tâche qui consiste bien entendu à produire rapidement et facilement du son. Dans la documentation de la classe `Greenfoot`, trouver les caractéristiques de cette méthode. 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 le nom d'un fichier (un `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 dossier 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 y 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");
```

Testez cette méthode !

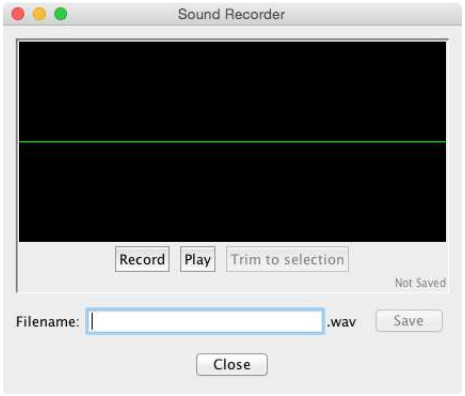
Exercice 3.25

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.

3.9 Fabriquer ses propres sons

Il y a plusieurs façons d'ajouter vos propres sons à un scénario Greenfoot. Vous pouvez, par exemple, trouver des effets sonores dans de nombreuses bibliothèques sur Internet ou produire vos sons en utilisant des logiciels spécialisés permettant d'enregistrer des sons ou des effets sonores.
Il est parfois un peu compliqué d'utiliser des sons car ceux-ci sont stockés dans de nombreux formats, et Greenfoot peut jouer certains sons mais pas d'autres. On peut en général faire jouer par Greenfoot des sons au format MP3, AIFF, AU et WAV. (Certains fichiers WAV ne pourront toutefois pas être joués – c'est compliqué...)
C'est pourquoi nous allons montrer ici la méthode la plus simple pour intégrer nos propres sons à notre scénario : les enregistrer nous-même directement dans Greenfoot.
Les deux sons que nous avons utilisés dans la section précédente ont été enregistrés en parlant tout simplement dans le microphone intégré à notre machine. Greenfoot dispose d'un enregistreur sonore intégré qui vous permet de faire de même.
Pour commencer, sélectionnez l'élément Show Sound Recorder du menu Controls. Cela fait apparaître la fenêtre de contrôle d'enregistrement du son dans Greenfoot, comme montré ci-dessous :



En utilisant cette fenêtre, vous pouvez maintenant enregistrer vos propres sons en utilisant le bouton Record et en parlant dans le microphone. [1] Appuyez sur le bouton Stop Recording lorsque vous avez terminé. Vous pouvez utiliser le bouton Play pour entendre votre son et vous assurez qu'il convient.
Il n'y a qu'une seule opération possible : Trim to selection. Elle permet de couper des parties superflues d'un extrait sonore au début ou à la fin de celui-ci, uniquement. Vous aurez souvent un peu de bruit parasite au début ou à la fin d'un enregistrement sonore, et cela ne fera pas bonne figure dans votre programme : un silence eu début donnera l'impression que votre son est joué en retard.

1. Evidemment, cela ne fonctionne que si votre ordinateur est équipé d'un microphone. Les portables les plus récents sont équipés d'un microphone intégré. Pour certains ordinateurs de bureau, vous devrez connecter un microphone externe. Si vous n'en avez pas, vous pouvez passer cette section.

Pour éliminer les parties non voulues, sélectionner ce que vous voulez conserver de l'extrait sonore avec la souris et appuyez ensuite sur le bouton Trim to selection.
Finalement, choisissez un nom pour votre son, entrez-le dans le champ de saisie du nom et cliquez le bouton Save. Le son sera sauvegardé au format WAV et recevra automatiquement une extension « .wav ». Si, par exemple, vous avez nommé votre son « blip », il sera enregistré sous le nom « blip.wav ». Greenfoot enregistrera automatiquement ce fichier son au bon endroit, à savoir dans le fichier « sounds » de votre dossier de scénarios.
Vous pourrez ensuite utiliser ce son via un appel à la méthode que nous avons déjà vue plus haut :

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

Il est temps maintenant d'essayer par vous-même.

Exercice 3.26

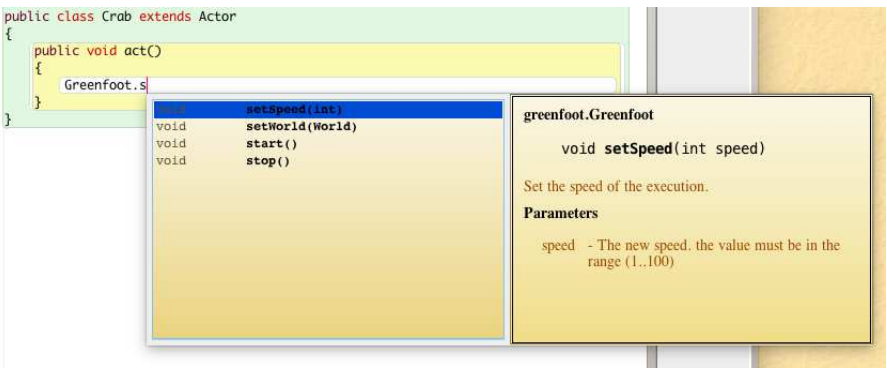
Si vous disposez d'un microphone intégré à votre machine, enregistrez vos propres sons à utiliser lorsque le crabe ou un ver se font manger. Enregistrez les sons, placez-les dans le dossier adéquat du dossier de scénario et utilisez-les dans votre code source.

3.10 Complétion automatique du code

Pour augmenter votre productivité, l'éditeur de Greenfoot permet la *complétion automatique du code* pour la saisie des appels de méthodes. Vous pouvez utiliser la complétion automatique chaque fois que vous êtes sur le point de taper le nom d'une méthode que vous désirez appeler. On active la complétion en formant la combinaison de touches CTRL-Espace.
Par exemple, si vous tapez

```
Greenfoot.
```

dans l'éditeur et que le curseur est juste après le point, et que vous tapez ensuite CTRL-Espace, une boîte de dialogue apparaît, donnant la liste de toutes les méthodes que vous pouvez appeler dans ce contexte (ici, toutes les méthodes de la classe `Greenfoot`). Si ensuite vous commencez à saisir le début du nom d'une méthode, comme sur la figure ci-dessous où on a tapé la lettre « s », la liste des méthodes proposées se réduit à celles qui commencent par la suite de symboles que vous avez déjà entrés.



Vous pouvez sélectionner des méthodes dans cette liste en utilisant les flèches du clavier ou avec la souris, et les insérer dans votre code en utilisant la touche Return.
Utiliser la complétion de code est utile pour examiner la liste des méthodes à disposition, pour trouver une méthode dont vous connaissez l'existence mais dont vous avez oublié le nom exact, pour consulter la documentation associée, pour déterminer le nombre de paramètres ou simplement pour diminuer le temps de saisie et accélérer la quantité de symboles tapés à la minute.

3.11 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 notation « 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.
Encore un concept très important que nous avons rencontré : la possibilité de définir nos propres méthodes. Nous avons vu comment définir des méthodes pour permettre l'exécution de différentes sous-tâches, et comment les appeler depuis d'autres méthodes de la même classe.

3.12 Concepts du chapitre

- Lorsque nous désirons appeler une méthode qui n'est pas définie dans notre classe ou qui est héritée, nous devons spécifier la classe ou l'objet qui dispose de cette méthode avant d'écrire le nom de la méthode, et ajouter un point juste après. Cela s'appelle la *notation « point »*.
- Les méthodes qui « appartiennent » à des classes, par opposition à des méthodes d'objets, sont marquées par le mot-clef `static` dans leur en-tête. Elles sont aussi appelées *méthodes de classe*.
- Une *définition de méthode* contient le code associé à une nouvelle action pour les objets de la classe courante. Cette action n'est pas exécutée immédiatement, mais la méthode peut être appelée plus tard pour exécuter le code figurant dans la définition.
- Des *commentaires* sont écrits dans le code source ; ils fournissent des explications concernant ce code aux lecteurs humains. Ils sont ignorés par le compilateur.

– La *documentation de l'API* donne une liste de toutes les classes et méthodes à disposition dans Greenfoot. Nous aurons souvent besoin d'y consulter la liste des méthodes prédéfinies.

Chapitre 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 donnerons quelques idées dans ce sens à la fin du chapitre. Nous allons maintenant exposer en détail et mettre en pratique un certain nombre d'améliorations du jeu.

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 présents à chaque début de partie et que cela se fasse automatiquement.
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 pourrez utiliser 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 usuelle `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 dans Greenfoot.
On trouve ensuite l'en-tête de classe et un commentaire (le bloc de lignes de couleur bleutée qui commence avec des astérisques ; nous avons déjà vu cela au chapitre précédent). 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 son 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 créer automatiquement nos acteurs. Si nous insérons le code de création d'un acteur dans le constructeur, ce code sera également exécuté. On écrit par exemple :

```
public CrabWorld()
{
    super(560, 560, 1);
    Crab myCrab = new Crab();
    addObject(myCrab, 250, 200);
}
```

Ce code va automatiquement créer un nouveau crabe et nous le placera à la position $x = 250$, $y = 200$ dans notre monde. La position $(250; 200)$ est le pixel qui se trouve à 250 pixels du bord gauche du monde, et 200 pixels du haut. L'origine de notre système de coordonnées, soit le point $(0; 0)$, se trouve en haut à gauche de notre monde (Figure 4.1).
Nous utilisons quatre nouvelles choses ici : une variable, une affectation, le mot-clef `new` pour créer le nouveau crabe, et la méthode `addObject`. Dans les paragraphes qui suivent, nous allons discuter de ces éléments les uns après les autres.

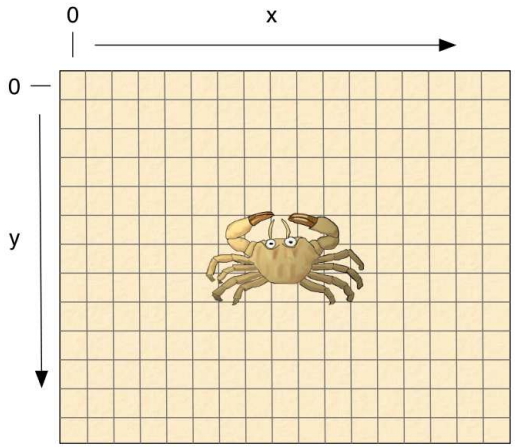


FIGURE 4.1 – Le système de coordonnées du monde

4.2 Créer de nouveaux objets

Si nous voulons ajouter un crabe dans le monde, la première chose dont nous avons besoin est un crabe. Dans notre cas, le crabe et un objet de la classe `Crab`. Jusque là, nous avons créé nos objets de façon interactive, en faisant un clic droit sur la classe `Crab` et en sélectionnant l'élément `new Crab()` du menu pop-up.
Nous voulons maintenant que le code de notre constructeur crée le nouvel objet crabe pour nous automatiquement.
En Java, le mot-clef `new` nous permet de créer de nouveaux objets à partir de toute classe existante. Par exemple, l'expression

```
new Crab()
```

crée une nouvelle instance de la classe `Crab`. L'expression utilisée pour créer de nouveaux objets commence toujours par le mot-clef `new`, suivi par le nom de la classe que nous désirons créer et une liste de paramètres, qui est vide dans notre exemple. La liste des paramètres nous permet de passer des paramètres au constructeur de notre nouvel objet. Dans la mesure ou nous n'avons pas explicité de constructeur dans notre classe `Crab`, la liste des paramètres par défaut est vide. (Peut-être avez-vous remarqué que l'élément que nous sélectionnons dans le menu pop-up de la classe pour créer des objets correspond précisément à cette instruction.)
Dans le code du constructeur donné plus haut, vous trouvez l'expression `new Crab()` qui forme la moitié de droite de la première ligne que nous avons insérée dans ce constructeur. Lorsque nous créons un nouvel objet, nous devons faire quelque chose avec lui. Dans notre cas, nous *l'affectons* à une *variable*.

4.3 Les variables

Lorsqu'on programme, on a souvent besoin de stocker de l'information pour la conserver et l'utiliser plus tard. On peut le faire en utilisant des *variables*.
Une variable est un morceau d'espace de stockage. Un nom est donné à chaque variable, pour permettre de s'y référer par la suite. Lorsque nous dessinons des diagrammes de nos

objets ou des schémas explicatifs de code, nous représentons les variables à l'aide de boîtes vides avec leur nom sur la gauche. On a schématisé ci-dessous une variable dont le nom est `age`. (Nous pourrions avoir envie de stocker l'âge du crabe.)

En Java, les variables ont toujours un *type*. Le type de la variable nous indique quelle sorte de données peut être stockée dans celle-ci. Par exemple, une variable de type `int` peut stocker des nombres entiers, une variable de type `boolean` peut stocker les valeurs `true` ou `false`, exclusivement, et une variable de type `Crab` peut stocker des « objets crabes ». Dans notre code source, lorsque nous avons besoin d'une variable, nous pouvons en créer une en écrivant une *déclaration de variable*. Une déclaration de variable est très simple : nous écrivons simplement le type et le nom de la variable, et terminons par un point-virgule. Par exemple, si nous voulons une variable `age` pour y stocker des nombres entiers comme montré ci-dessus, nous pouvons écrire

```
int age;
```

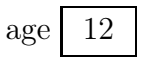
Cela créera notre variable `age`, prête à stocker des valeurs de type `int`.

4.4 L'affectation

Une fois que nous disposons d'une variable, nous sommes prêts à stocker quelque chose dedans. Cela se fait à l'aide d'une *instruction d'affectation*.
Une affectation en Java s'écrit à l'aide du signe égal : =.
Par exemple, pour stocker le nombre 12 dans notre variable `age`, nous pouvons écrire

```
age = 12;
```

Il est judicieux de lire les instructions d'affectation de droite à gauche : *La valeur 12 est stockée dans la variable* `age`. Après l'exécution de cette instruction d'affectation, notre variable contiendra la valeur 12. Dans nos diagrammes, nous représentons cela en écrivant la valeur dans la boîte, comme ci-dessous :



La forme générale d'une instruction d'affectation est la suivante :

```
variable = expression;
```

C'est à dire qu'à gauche du signe égal se trouve toujours le nom de la variable, et qu'à droite se trouve une expression qui est évaluée et dont la valeur est stockée dans la variable. Souvent, dans nos programmes, nous voulons déclarer une variable et y stocker une valeur immédiatement. C'est pourquoi nous trouverons souvent la déclaration d'une variable suivie directement de son affectation :

```
int age;
age = 12;
```

Vu la fréquence à laquelle cela se produit, Java nous permet d'écrire ces deux instructions ensemble sur une seule ligne :

```
int age = 12;
```

Cette ligne crée la variable entière `age` et lui affecte la valeur 12, en une seule fois. Cela fait exactement la même chose que lorsqu'on exécute les deux instructions séparément, à la suite l'une de l'autre.
L'affectation écrase toute valeur précédemment stockée dans une variable. Ainsi, si nous avons notre variable `age` qui contient maintenant la valeur 12, et que nous écrivons ensuite

```
age = 42;
```

la variable `age` contiendra la valeur 42 après exécution de l'instruction. La valeur 12 est écrasée, et nous ne pouvons plus la récupérer.

4.5 Les variables dont le type est un objet

Nous avons mentionné plus haut que les variables peuvent non seulement stocker des nombres, mais aussi des objets.
Le langage Java distingue les *types primitifs* et les *types objet*. Les types primitifs sont en nombre limité. Ce sont des types de données utilisés couramment tels que : `int`, `boolean` et `char`. Il n'y en a pas beaucoup ; on trouve facilement sur le net la liste des différents types primitifs de Java.
Toute classe dans Java définit également un type, que l'on appelle un *type objet*. C'est le cas de notre classe `Crab`, qui nous donne un « type `Crab` » ou encore celui de la classe `Lobster` qui définit un type du même nom et ainsi de suite. Nous pouvons déclarer des variables à l'aide de ces types que nous avons créés :

```
Crab myCrab;
```

Notons ici que comme plus haut, nous notons le type en premier, suivi par le nom de la variable que nous choisissons au moment de la déclaration, le tout suivi d'un point-virgule.
Une fois que nous disposons d'une variable de type objet, nous pouvons y stocker un objet. Nous allons combiner cette instruction avec celle qui permet la création d'un objet de type `Crab`, ce que nous avons déjà vu au paragraphe 4.2.

```
Crab myCrab;
myCrab = new Crab();
```

Comme plus haut, nous pouvons combiner déclaration et affectation en écrivant une seule ligne de code :

```
Crab myCrab = new Crab();
```

Cette seule ligne de code a trois effets :

- Une variable de type `Crab` dont le nom est `myCrab` est créée.
- Un objet de type `Crab` est créé.
- L'objet nouvellement créé est affecté à la variable `myCrab`.

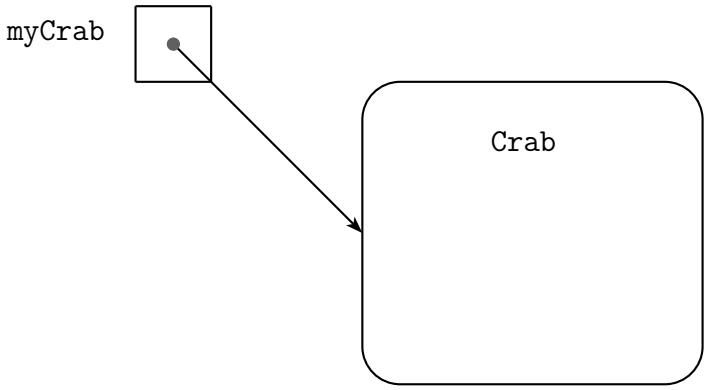


FIGURE 4.2 – Une variable de type objet qui fait référence à un objet

Lorsque l'on écrit une instruction d'affectation, le côté droit de l'affectation, soit ce qui se trouve à droite du signe égal est toujours exécuté en premier : l'objet de type crabe est créé. Ensuite, l'affectation à la variable qui se trouve à gauche du signe est réalisée. Dans nos diagrammes, nous dessinons les variables qui stockent les objets en utilisant une flèche, comme sur la figure 4.2. Dans ce cas, la variable `myCrab` contient une référence à l'objet de type `Crab`. Le fait que les variables objets contiennent toujours des *références* aux objets et non les objets directement deviendra important plus tard ; c'est pourquoi nous prendrons garde à toujours représenter les variables de ce type de cette façon.
Le type de variable et le type de la valeur qui lui est affectée doivent toujours correspondre. On peut affecter une valeur de type `int` à une variable de type `int`, et on peut affecter un objet de type `Crab` à une variable de ce type. Mais on ne peut pas affecter un objet de type `Crab` à une variable de type `int`. [1]

Exercice 4.1

Ecrire la déclaration d'une variable de type `int` dont le nom est `score`.

Exercice 4.2

Déclarer une variable nommée `isHungry` de type `boolean`, et lui affecter la valeur `true`.

Exercice 4.3

Déclarer une variable nommée `year`, et lui affecter la valeur 2015. Lui attribuer ensuite la valeur 2016.

Exercice 4.4

Ecrire la déclaration d'une variable de type `Crab` dont le nom est `littleCrab` et lui affecter un nouvel objet de type crabe.

1. Lorsque nous disons « les types doivent correspondre », cela ne veut pas forcément dire qu'ils doivent être littéralement identiques. Il y a des situations dans lesquelles des types différents se correspondent tout de même. Par exemple, nous pouvons affecter un `Crab` à une variable de type `Actor`, car la classe `Crab` est une sous-classe de la classe `Actor`. Nous reparlerons de ce genre de subtilités.

Exercice 4.5

Ecrire la déclaration d'une variable de type `Control` dont le nom est `inputButton`. Créer un objet de type `Button` et affecter cet objet à la variable.

Exercice 4.6

Quelle est l'erreur commise dans l'écriture de l'instruction : `int myCrab = new Crab();` ?

4.6 Utiliser des variables

Une fois que nous avons déclaré une variable et que nous lui avons affecté une valeur, nous pouvons l'utiliser à l'aide du nom de cette variable.
Par exemple, le code ci-dessous permet de déclarer et d'affecter une valeur à chacune de deux variables entières :

```
int n_1 = 7;
int n_2 = 13;
```

Nous pouvons alors les utiliser à droite du symbole d'affectation d'une autre instruction :

```
int sum = n_1 + n_2;
```

Après cette instruction, la variable `sum` contient la somme des contenus des variables `n_1` et `n_2`. Si nous écrivons

```
n_3 = n_1;
```

le contenu de la variable `n_1`, la valeur 7 en l'occurence, sera copié dans `n_3`, en supposant que la variable `n_3` a été déclarée auparavant. Les deux variables contiennent maintenant la même valeur, à savoir le nombre entier 7.

Exercice 4.7

Ecrire la déclaration d'une variable de type `int` dont le nom est `surface`. Ecrire ensuite une instruction d'affectation qui attribue à cette variable le produit du contenu de deux autres variables nommées `longueur` et `largeur`.

Exercice 4.8

Ecrire la déclaration d'une variable de type `int` dont le nom est `children`. Ecrire ensuite une instruction d'affectation qui attribue à cette variable la somme du contenu de deux autres variables nommées `daughters` et `sons`.

Exercice 4.9

Déclarer deux variables `x` et `y` dont le type est `int`. Affecter les valeurs 23 et 17 à `x` et `y`, respectivement. Ecrire ensuite le code qui permet de permuter les deux valeurs. La valeur de `x` doit alors être 17 et celle de `y` 23.

4.7 Ajouter des objets à notre monde

Nous savons maintenant comment créer un nouveau crabe et comment le stocker dans une variable. Il nous reste à ajouter ce nouveau crabe à notre monde. Dans l'extrait de code présenté à la page 45, nous avons vu que nous pouvons employer la ligne suivante :

```
addObject(myCrab, 250, 200);
```

La méthode `addObject` est une méthode de la classe `World` qui nous permet d'ajouter un « objet acteur » à notre monde. 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.
Nous venons de créer un nouveau crabe et de le stocker dans notre variable `myCrab`. Nous pouvons maintenant utiliser ce crabe comme premier paramètre de la méthode `addObject`, en nous référant à la variable « dans laquelle » il est contenu. Les paramètres `x` et `y` restants permettent de spécifier les coordonnées de l'endroit où nous souhaitons placer l'objet.
L'ensemble des instructions (déclaration de variable, création d'objet, affectation, et ajout de l'objet au monde courant) a l'allure suivante :

```
Crab myCrab = new Crab();
addObject(myCrab, 250, 200);
```

Nous pouvons utiliser un objet de type `Crab` pour le paramètre de type `Actor`, car un crabe est un acteur. La class `Crab` est en effet une sous-classe de la classe `Actor`.

Exercice 4.10

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 4.11

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 4.12

Ajouter le code créant 2 vers de sable placés à différents endroits du monde `CrabWorld`.

4.8 Sauvegarder le monde

Nous allons maintenant montrer une façon plus simple d'obtenir le même résultat.
Il vous faut tout d'abord supprimer le code que vous avez écrit dans le dernier groupe d'exercices, de façon à ce que les objets ne soient plus créés automatiquement. Après cette opération, le monde devrait être vide lorsque vous compilez votre scénario. Faites ensuite les exercices suivants :

Exercice 4.13

Compiler votre scénario. Placer ensuite un crabe, trois homards et dix vers dans votre monde, de façon interactive.

Exercice 4.14

Faites un clic droit sur l'arrière-plan du monde. Le menu contextuel du monde apparaît alors ; sélectionnez l'élément « Save the world » de ce menu(voir figure 4.3).

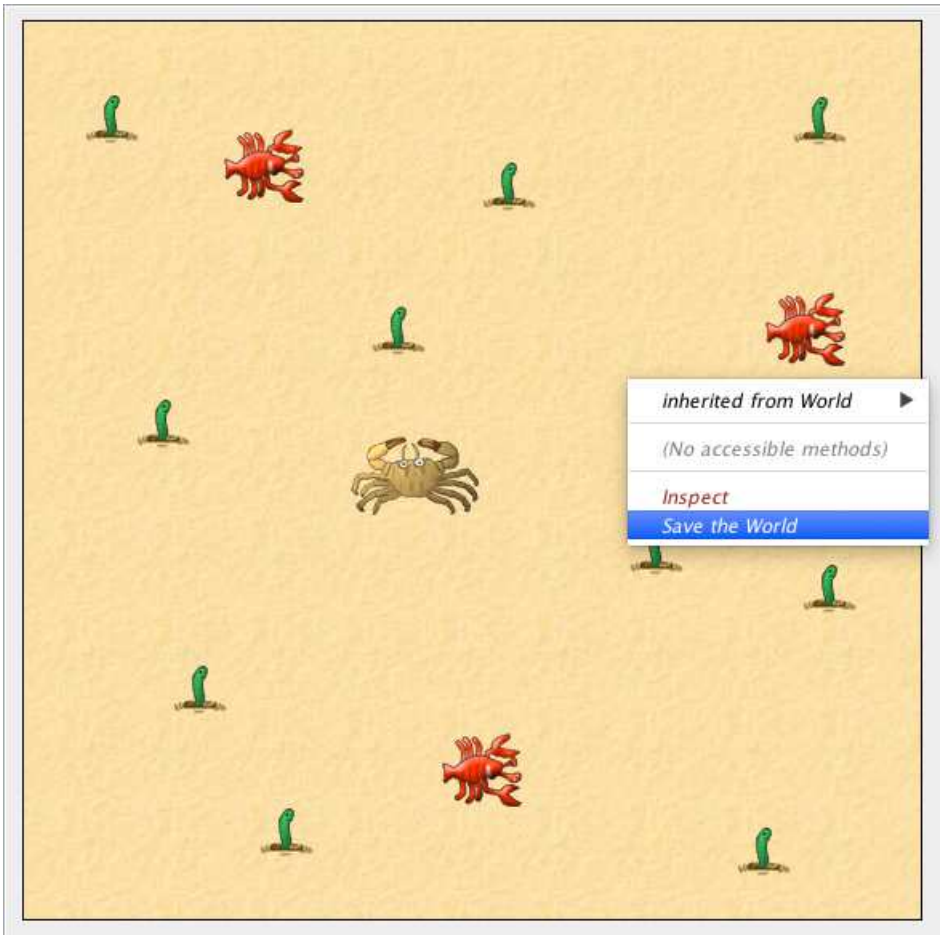


FIGURE 4.3 – La fonction qui permet d'enregistrer le monde

Lorsque vous placez un certain nombre d'objets dans votre monde et que vous sélectionnez ensuite la fonction « Save the world », vous observerez que le code source de la classe `CrabWorld` s'ouvre dans l'éditeur et qu'un certain nombre de lignes de codes se sont ajoutées à la classe. Etudiez ce code avec attention.

Vous pourrez constater que ce code est structuré de la façon suivante :

- Le constructeur contient maintenant un appel à une nouvelle méthode nommée `prepare`.
- Une définition de méthode pour cette méthode a été ajoutée.
- Le corps de la méthode `prepare()` contient du code qui crée et ajoute tous les acteurs que nous venons de créer et d'ajouter « à la main ».

Que se passe-t-il ici ?
Lorsque nous créons des objets de façon interactive et que nous cliquons ensuite sur « Save the world », Greenfoot écrit ensuite du code dans notre classe `CrabWorld` pour permettre de recréer plus tard la situation que nous venons de construire à la main. Il le réalise en créant une méthode appelée `prepare()` et en plaçant dans le constructeur un appel à cette méthode.
La conséquence est que maintenant, chaque fois que nous cliquons sur le bouton Compile ou Reset, les acteurs sont à nouveau créés dans la configuration voulue.
Les exercices précédents consacrés à l'écriture manuelle du code permettant de créer et de placer les acteurs nous aident à comprendre comment cette méthode fonctionne. Dans de nombreux cas, nous n'aurons pas à écrire ce code manuellement et pourrons utiliser la fonction « Save the world » à la place ; il est toutefois important de comprendre le processus en détail. Dans certaines occasions, nous devrons écrire ce type de code à la main pour obtenir quelque chose de plus sophistiqué.

4.9 Animer des images

Maintenant que nous avons réussi à obtenir automatiquement la mise en place des personnages de notre jeu, nous pouvons passer un peu de temps à améliorer les détails.
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 utiliser des variables en plus et savoir comment travailler avec des images dans Greenfoot.

4.10 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 4.15

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 un objet de type `GreenfootImage` à partir d'un fichier image comme décrit ci-dessus, et l'affecter à une variable de type `GreenfootImage`. Nous utiliserons ensuite la méthode `setImage` de l'acteur pour l'attribuer à cet acteur. Voici un extrait de code qui permet de réaliser cela :

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

Pour remplacer cette deuxième image par l'image d'origine, nous écrivons :

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

Cela permet de créer les objets images à partir des fichiers crab.png et crab2.png et de les affecter aux variables `image1` et `image2`, respectivement. Nous pouvons ensuite utiliser ces variables pour attribuer l'une ou l'autre image à notre acteur. Pour créer un effet d'animation, nous devons simplement faire en sorte que le passage d'une image à l'autre se fasse en alternance. Il faut donc que ces deux extraits de code soient exécutés en boucle, l'un après l'autre : le premier, le second, le premier, le second, et ainsi de suite.
Nous pourrions immédiatement ajouter du code de ce genre dans notre méthode `act`. Toutefois, avant de le faire, nous allons discuter d'une amélioration : nous voulons séparer la création de nos images de l'affectation de ces images à notre crabe.

La raison en est l'efficacité. Lorsque notre programme tourne avec l'animation des images, nous allons changer l'image un très grand nombre de fois, plusieurs fois par seconde. En utilisant le code tel qu'écrit ci-dessus, nous constaterions que le programme charge les images à partir des fichiers et crée les objets correspondants un grand nombre de fois également. Cela n'est pas nécessaire ; c'est même du gaspillage de ressources. Il suffit de créer les objets images une seule fois et de passer de l'un à l'autre autant de fois qu'on le désire. En d'autres termes, nous voulons séparer notre code en deux morceaux distincts, comme ci-dessous :

- Code à exécuter une seule fois à la création du crabe :

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

- Code à exécuter en boucle tant que le crabe se déplace, en fonction de l'image courante :
  `setImage(image1);`
  ou
  `setImage(image2);`

En conséquence, nous créerons d'abord les images et les stockerons dans une variable ; plus tard, nous pourrons utiliser les images mises en mémoire sans devoir les créer à nouveau à chaque fois que nous voulons changer l'image qui s'affiche.
Pour réaliser cela, nous devons ajouter un nouveau concept que nous n'avons pas encore utilisé : *la variable d'instance*.

4.11 Variables d'instance (champs)

Java permet l'utilisation de plusieurs sortes de variables. Celles que nous avons utilisées précédemment s'appellent des *variable locales*, et celles dont nous discuterons maintenant sont des *variables d'instance*. Les variables d'instance sont parfois appelées *champs*.
La première différence entre ces deux types de variable est qu'elles ne se déclarent pas au même endroit : les variables locales sont déclarées dans les méthodes, alors que les variables d'instance sont déclarées dans la classe, mais avant toute méthode, comme on peut le voir sur l'extrait de code ci-dessous :

```
public class Crab extends Actor
{
    private GreenfootImage image1; // Variable d'instance
    private GreenfootImage image2; // Variable d'instance
    private int wormsEaten;        // Variable d'instance
    public void act()
    {
        boolean isAlive; // Variable locale
        int n;           // Variable locale
    }
        // ... acting code omitted ...
}
```

Une autre différence visible est que les variables d'instance ont le mot-clef `private` placé devant elles, comme on peut le constater sur l'extrait de code précédent.
Il est plus important toutefois de noter la différence de comportement entre ces deux types de variables. Plus spécifiquement, c'est la *durée de vie* de chacun de ces types qui va clairement les différencier.
D'une part, les variables locales appartiennent à la méthode dans laquelle elles sont déclarées, et disparaissent sitôt que cette méthode termine son exécution. Chaque fois que nous appelons la méthode, les variables sont déclarées à nouveau, et peuvent être utilisées tant que la méthode est en cours d'exécution, mais elles ne « survivent » pas entre les différents appels de méthode. Les valeurs ou les objets qui y sont stockés ne sont plus disponibles à la fin de l'exécution. [2]
Les variables d'instances, d'autre part, appartiennent à l'objet dans lequel elles sont déclarées et sont à disposition pendant toute la durée de vie de cet objet. Elles peuvent être utilisées encore et encore, durant de multiples appels de différentes méthodes. Ainsi, si nous voulons qu'un objet conserve de l'information durant un temps conséquent, la variable d'instance est ce qu'il nous faut.
Les variables d'instance sont définies au sommet de la classe, [3] juste après l'en-tête de classe, en utilisant le mot-clef `private` suivi par le type de la variable et le nom de celle-ci :

```
private variable-type variable-name;
```

Dans notre cas, dans la mesure où nous désirons stocker des objets de type `GreenfootImage`, le type de la variable est `GreenfootImage` et nous utilisons les noms `image1` et `image2` comme précédemment :

```
public class Crab extends Actor
{
    private GreenfootImage image1; // Variable d'instance
    private GreenfootImage image2; // Variable d'instance

        // ... méthodes ...
}
```

Exercice 4.16

Avant d'ajouter ce code à la classe `Crab`, faites un clic droit sur un objet crabe dans votre monde et sélectionnez Inspect dans le menu contextuel du crabe. Prenez note des noms des variables qui sont montrées dans la fenêtre associée à cet objet crabe.

Exercice 4.17

Pourquoi pensez-vous que le crabe dispose de variables à ce niveau, malgré le fait que nous n'en avons pas déclaré une seule ?

2. Plus précisément, les variables locales appartiennent à *l'étendue*, en anglais *scope*, dans laquelle elles sont déclarées et n'existent que jusqu'à la fin de cette étendue. Souvent, il s'agit d'une méthode, mais si une variable est déclarée dans une instruction `if`, par exemple, elle disparaît à la fin de ce `if`.

3. Java ne force pas la déclaration des variables d'instance au tout début de la classe, mais nous procéderons toujours ainsi, car c'est une bonne pratique et cela nous aide à trouver facilement les déclarations de variables lorsque nécessaire.

Exercice 4.18

Ajoutez les déclarations pour les deux variables destinées à stocker les images, en prenant exemple sur les extraits de code du chapitre. Assurez-vous du fait que la classe compile sans erreur.

Exercice 4.19

Après avoir ajouté les variables et recompilé le scénario, inspectez à nouveau votre objet. Prenez note des noms des variables et de leurs valeurs, montrées dans les cases blanches.

Dans le diagramme de la figure 4.4, on représente l'objet à l'aide d'un rectangle aux bords arrondis, et les variables d'instance à l'aide de petits rectangles disposés à l'intérieur. Notez bien que la déclaration de ces deux variables de type `GreenfootImage` ne nous donne pas directement deux objets de ce type. Cela ne nous fournit que deux *espaces vides* dans lesquels on pourra stocker nos objets. Nous devons ensuite créer les deux objets images

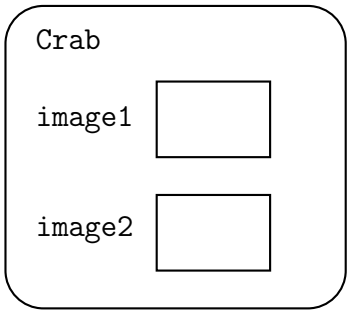


FIGURE 4.4 – Un objet de type crabe avec deux variables d'instance vides

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

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

Il ne nous reste plus maintenant qu'à créer les deux images et à les affecter à nos variables d'instance :

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

Après ces deux instructions, nous disposons de trois objets (un crabe et deux images), et les variables du crabe contiennent des références aux objets. Cette situation est illustrée sur la figure 4.5. 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.

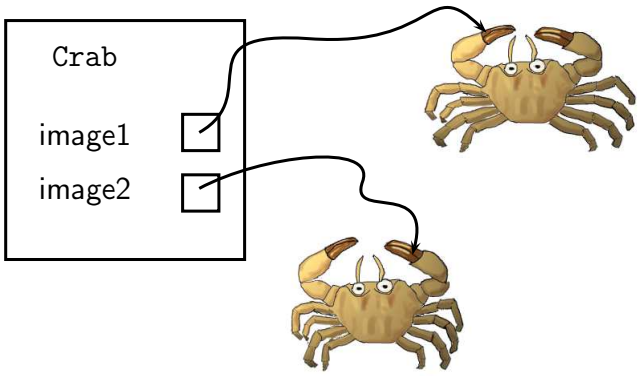


FIGURE 4.5 – Un objet crabe avec deux variables pointant sur des images

4.12 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.*;

// comment ommitted

public class Crab extends Actor
{
    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'inclut 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 4.5.
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 4.20

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 4.21

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 ce qui avait été noté précedemment.

4.13 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.

4.14 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 4.22

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 4.23

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 4.24

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

4.15 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 ( isTouching(Worm.class) )
    {
        removeTouching(Worm.class);
        Greenfoot.playSound("slurp.wav");

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

Exercice 4.25

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

Exercice 4.26

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.

4.16 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 4.27

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.

4.17 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.

4.18 Concepts du chapitre

- Un *constructeur* d'une classe est une méthode spéciale qui est exécutée chaque fois qu'une nouvelle instance est créée.
- Au niveau du code d'un programme, les objets Java peuvent être créés à l'aide du mot-clef `new`
- Les *variables* peuvent être utilisées pour stocker de l'information afin de l'utiliser plus tard.
- Les variables peuvent être créées au moyen d'une *déclaration de variable*.
- Nous pouvons stocker des valeurs dans les variables en utilisant une *instruction d'affectation*.
- Les variables de *type primitif* permettent de stocker des nombres, des booléens et des caractères ; les variables de type *objet* permettent de stocker des objets.
- Les objets sont stockés dans des variables. Celles-ci gardent une *référence* à l'objet.
- L'image visible à l'écran d'un acteur de Greenfoot se trouve dans une variable d'instance de type `GreenfootImage`. Cette variable d'instance est héritée de la classe `Actor`.
- Les *Variables d'instance* ou *champs* sont des variables attachées à un objet plutôt qu'à une méthode.
- *Durée de vie d'une variable d'instance* : La variable en question perdure aussi longtemps que l'objet qui la contient existe.
- *Durée de vie d'une variable locale* : La variable en question ne survit pas à la fin de l'exécution de la méthode dans laquelle elle a été déclarée.
- Nous pouvons tester si deux choses sont *égales* en utilisant un double symbole d'égalité : `==`.
- *L'instruction* `if/else` permet d'exécuter un morceau de code lorsqu'une condition est vérifiée, et un autre morceau de code lorsque cette condition est fausse.

4.19 Un peu de pratique

Cette fois, nous vous proposons quelques exercices supplémentaires avec des appels de méthode, incluant une nouvelle méthode héritée de la classe `Actor` ; il s'agit également de pratiquer un peu plus l'utilisation de variables.

Exercice 4.28

Dans cet exercice, on va rendre le homard un peu plus dangereux. La classe `Actor` dispose d'une méthode appelée `turnTowards`. Utiliser cette méthode pour faire tourner les homards en direction du centre de l'écran de temps en temps. Varier la fréquence à laquelle cela se produit, et varier également la vitesse de déplacement des homards et du crabe.

Exercice 4.29

Ajouter un compteur de temps au crabe. On peut le faire en ajoutant une variable de type `int` qui est incrémentée chaque fois que le crabe agit. Dans les faits, on compte les cycles de

la méthode `act`. Cette variable doit-elle être une variable locale ou une variable d'instance ? Pourquoi ?

Exercice 4.30

Faites une partie de votre jeu. Après avoir gagné, c'est à dire mangé 8 vers, inspectez l'objet crabe et déterminez combien de temps il vous a fallu. Quel a été le nombre de cycles nécessaires à votre victoire.

Exercice 4.31

Déplacer votre compteur temporel de la classe `Crab` vers la classe `CrabWorld`. Il est plus judicieux de placer le contrôle temporel dans le monde plutôt que de le confier à un acteur. Il est facile de déplacer la variable. Pour déplacer l'instruction qui permet l'incrémentation du temps, il vous faudra définir une méthode `act` dans la classe `CrabWorld`. Les sous-classes de `World` peuvent disposer d'une méthode `act` tout comme les sous-classes de `Actor`. Il suffit de copier l'en-tête de la méthode `act` de la classe `Crab` pour créer une nouvelle méthode `act` dans la classe `CrabWorld` et d'y placer votre instruction d'incrémentation.

Exercice 4.32

Transformer le compteur de temps en un contrôleur de durée de jeu. C'est à dire : Choisir une durée pour la variable de gestion du temps, 500 par exemple et décrémenter cette variable à chaque fois que la méthode `act` est invoquée. Ce qui revient à faire un compte à rebours temporel. Si le « timer » atteint zéro, le jeu doit se terminer avec un son de type « temps écoulé ». Faire des tests avec différentes valeur de timer.

Exercice 4.33

Examiner la méthode `showText` de la classe `World`. De combien de paramètres dispose-t-elle ? Quels sont ces paramètres ? Quelle est la valeur de retour de cette méthode ? Que fait-elle ?

Exercice 4.34

Faire afficher le compte à rebours à l'écran durant le jeu à l'aide de la méthode `showText`. On peut le faire directement dans la méthode `act` de la classe `CrabWorld`. Il faut utiliser une instruction similaire à celle donnée ci-dessous :

```
showText("Time left: " + time, 100, 40);
```

où `time` est le nom donné au timer.

Chapitre 5

Compter le score

Un élément qui manque clairement à notre exemple du crabe est le décompte du score. Nous en avons réalisé une partie : le crabe conserve un décompte interne du nombre de vers qu'il a mangés, en utilisant une variable de type entier. Nous avons vu que les variables jouent un rôle important dans la comptabilisation du score, mais nous n'avons pas encore de solution complète. Premièrement, nous pourrions désirer que d'autres actions contribuent à faire gagner ou perdre des points, et deuxièmement, nous n'avons pas encore d'affichage du score à l'écran. Pour changer un peu, nous discuterons de tout ceci dans le cadre d'un exemple différent. Après avoir lu ce chapitre, vous pourrez, si vous le désirez, reprendre le jeu du crabe et lui ajouter également un système de décompte du score ; vous devriez en être capable d'ici la fin de ce chapitre. Pour cette discussion, nous nous

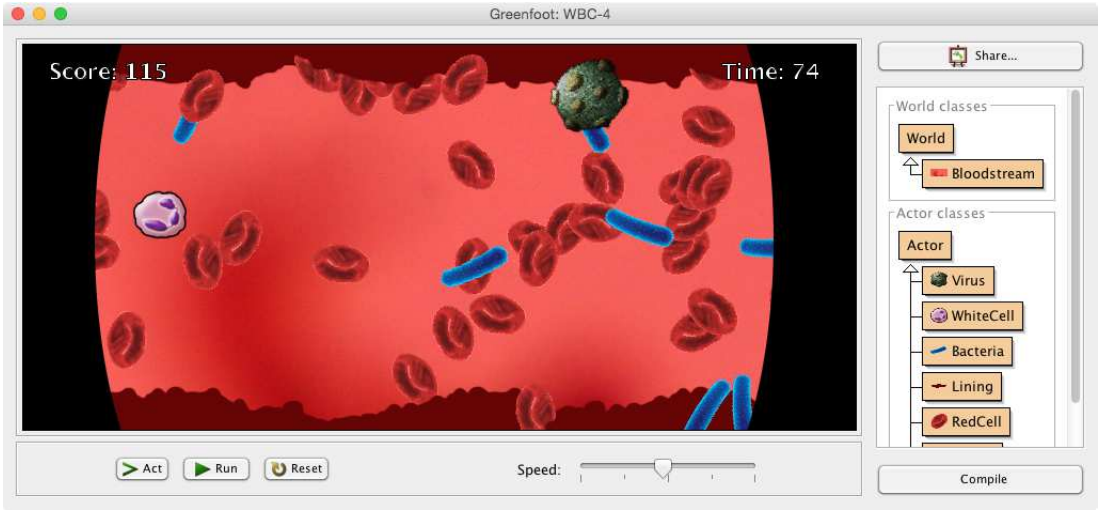


FIGURE 5.1 – Le scénario WBC (« White Blood Cell »)

baserons sur le scénario nommé « WBC » (pour « White Blood Cell », cf. figure 5.1). Il s'agit d'un petit jeu dans lequel nous contrôlons un globule blanc flottant dans la circulation sanguine d'une créature quelconque et notre travail consiste à attraper et éliminer les bactéries. Pour rendre les choses un peu plus intéressantes, il y a des virus également et nous décidons que notre globule blanc est d'une sorte qui ne peut que neutraliser les bactéries, mais pas les virus. Le virus est trop fort pour nous et endommage notre cellule, il doit donc être évité.

Ce chapitre est divisé en trois parties : nous commençons par brièvement examiner et analyser le scénario de départ (paragraphes 5.1 à 5.4) ; nous ajoutons ensuite quelques objets et fonctionnalités supplémentaires, en utilisant des techniques que nous avons déjà rencontrées dans les chapitres précédents (paragraphes 5.5 à 5.10). Dans les derniers paragraphes du chapitre, nous passerons à la découverte de nouvelles structures nous permettant d'ajouter un décompte de score.

5.1 WBC : Le point de départ

Ouvrez le scénario WBC-1 que vous trouverez dans le dossier des scénarios du livre et testez-le.

Exercice 5.1

Ouvrez le scénario WBC-1 et exécutez-le à l'aide de la commande « Run ». Décrivez ce que vous observez.

Exercice 5.2

Pour chacune des classes du scénario, écrire en une ou deux phrases une courte description de ce qu'elles représentent.

Exercice 5.3

Ouvrir le code source de chacune des quatre classes. Etudier ce code et essayer de comprendre son fonctionnement. Mettez en évidence les parties du code qui ne sont pas claires pour vous.

Vous devriez être capable de comprendre l'essentiel de ce qui se produit. Nous allons passer en revue rapidement les éléments intéressants de ces classes.
La classe `Lining` est triviale ; elle ne fait rien. Les objets de cette classe ne sont là que pour décorer ; Ils sont disposés le long des bords supérieur et inférieur de l'écran et n'interagissent avec aucun des autres objets.
Les autres classes méritent un regard plus attentif.

5.2 WhiteCell : mouvement contraint

La classe `WhiteCell` définit le globule blanc – c'est l'objet de notre jeu que nous contrôlons à l'aide du clavier. La structure de son code est plutôt simple : sa méthode `act` n'appelle qu'une méthode pour vérifier si une « entrée de type clavier » correspond à l'une des deux touches « haut » ou « bas », comme on peut le constater dans l'extrait de code ci-dessous :

```
/**
 * Act: move up and down when cursor keys are pressed.
 */
public void act()
{
    checkKeyPress();
}
/**
```

```
 * Check whether a keyboard key has been pressed and react if it has.
 */
private void checkKeyPress()
{
    if (Greenfoot.isKeyDown("up"))
    {
        setLocation(getX(), getY()-4);
    }

    if (Greenfoot.isKeyDown("down"))
    {
        setLocation(getX(), getY()+4);
    }
}
```

Les deux seules lignes intéressantes de cet extrait de code sont celles de la forme suivante :

```
setLocation(getX(), getY() + 4);
```

Leur fonction est d'appliquer un déplacement à notre objet, vers le haut ou vers le bas, en fonction du signe. Dans ces deux lignes, nous utilisons trois méthodes de la classe `Actor` que nous n'avons encore jamais vues : `setLocation(x, y)`, `getX()`, et `getY()`.

Exercice 5.4

Chercher ces trois méthodes dans la documentation de la classe `Actor`. Ecrire leur signature.

Les méthodes `getX()` et `getY()` renvoient les coordonnées x et y courantes, respectivement, sous forme de valeurs entières.
La méthode `setLocation` attend deux paramètres, une coordonnée x et une coordonnée y, et place l'acteur à l'endroit repéré par le point $(x; y)$ à l'écran. Sa signature est

```
void setLocation(int x, int y)
```

Donc, par exemple, l'appel à méthode suivant :

```
setLocation(120, 200)
```

ferait se téléporter notre globule blanc à l'endroit de l'écran repéré par $(120; 200)$. Mais nous ne sommes pas obligés de donner directement des nombres entiers en paramètre à la méthode `setLocation`. En effet, nous pouvons lui passer en paramètre les valeurs renvoyées par les méthodes `getX()` et `getY()` :

```
setLocation(getX(), getY())
```

Lorsqu'à la place d'un paramètre « codé en dur » on place un appel de méthode (ici, `getX()` ou `getY()`), la méthode interne est appelée en premier et le résultat qu'elle renvoie est utilisé comme paramètre de la méthode externe (ici `setLocation`).
Ecrite de cette façon, l'appel à cette méthode ferait calculer les deux coordonnées de notre position courante, et nous donnerait la position correspondant à ces deux coordonnées.

Cette instruction n'aurait donc aucun effet visible : elle nous positionnerait à l'endroit où nous nous trouvons déjà.
Il suffit toutefois d'une très légère modification pour créer le mouvement :

```
setLocation(getX(), getY() + 4);
```

Ici, avant de passer la valeur renvoyée par la méthode `getY()` comme second paramètre à la méthode `setLocation`, nous lui ajoutons 4. Par contre, nous ne modifions pas la valeur renvoyée par `getX()`. Ce qui nous place 4 cellules plus bas que là où nous nous trouvions avant l'appel à la méthode `setLocation`.
Nous avons choisi cette méthode pour générer le mouvement, à la place d'utiliser la méthode `move()` comme plus haut, parce que, de cette façon, le mouvement est indépendant de la rotation de notre globule. La méthode `move()` déplace l'acteur dans la direction à laquelle il fait face, alors que nous voulons déplacer le globule uniquement vers le haut ou le bas, sans devoir effectuer de rotation.

Note
Vous avez peut-être constaté que nous avons utilisé le mot `private` dans la déclaration de la méthode `checkKeyPress` :

```
private void checkKeyPress()
```

Cela s'appelle une *méthode privée*. Jusque là, nous avions déclaré toutes nos méthodes *publiques*.
Les mots-clés `private` et `public` sont appelés des *modificateurs d'accès*, et ils déterminent qui peut voir et appeler une méthode. Lorsqu'une méthode est publique, elle peut être appelée depuis les autres classes de notre programme. Lorsqu'une méthode est privée, elle ne peut être appelée que depuis les méthodes qui se trouvent dans la même classe. Elle ne sera pas visible depuis l'extérieur.
Par exemple, lorsque vous faites un clic droit sur l'objet `WhiteCell` de notre scénario courant, vous verrez la méthode `act` dans le menu popup, car elle est publique, mais pas la méthode `checkKeyPress` qui est privée.
Le but des méthodes publiques est d'offrir des fonctionnalités à d'autres parties du système, de manière à ce que notre objet puisse être appelé à faire quelque chose. Le but des méthodes privées est d'améliorer la structure de notre code en décomposant les tâches à effectuer en sous-tâches plus petites. Il n'est pas prévu qu'elles soient appelées depuis l'extérieur.
A partir de maintenant, nous déclarerons nos méthodes « privées » si elles ne doivent être appelées qu'à l'intérieur de la classe dans laquelle elles sont codées. Nous les déclarerons « publiques » si elles peuvent être appelées depuis d'autres classes.
Les variables, par contre, seront toujours déclarées « privées ».
On peut déclarer des variables publiques en Java, mais c'est de très mauvais goût et très mal vu par la communauté des programmeurs de ce langage.

5.3 Bactéries : se faire disparaître

Vous avez pu constater que les objets de type `Bacteria` flottent de la droite vers la gauche en tournant lentement sur eux-mêmes. Comme on peut le constater dans l'extrait de code ci-dessous, c'est la méthode `act` de la classe `Bacteria` qui se charge de cela :

```
/**
 * Float along the bloodstream, slowly rotating.
 */
public void act()
{
    setLocation(getX() - 2, getY());
    turn(1);

    if (getX() == 0)
    {
        getWorld().removeObject(this);
    }
}
```

La première ligne de cette méthode met en oeuvre la même technique que celle vue dans la classe `WhiteCell` pour générer le mouvement : `setLocation(int x, int y)` employée conjointement avec les méthodes `getX()` et `getY()`. Cette fois, nous voulons nous déplacer vers la gauche en soustrayant une certaine quantité à la valeur courante de la coordonnée x. A nouveau, nous ne pouvons pas utiliser la méthode `move()`, car nous désirons un mouvement dont la direction est indépendante de notre rotation.
La deuxième ligne est simple : nous faisons appel à la méthode `turn` pour effectuer la rotation lente.
Après cela suit une instruction `if` qui sert à enlever l'objet du monde lorsqu'il parvient au bord gauche de l'écran de jeu. Nous pouvons savoir si nous avons atteint le bord gauche en examinant si la valeur de notre coordonnée x est 0, et, si c'est le cas, quitter le monde en utilisant la ligne de code suivante

```
getWorld.removeObject(this)
```

Nous utilisons ici la méthode `removeObject` de la classe `World` dont la signature est :

```
void removeObject(Actor object)
```

Nous pouvons appeler cette méthode pour enlever un objet du monde. Nous devons prendre garde aux deux points suivants :

- Nous devons passer un objet en paramètre. Il s'agit de l'objet qui doit être enlevé du monde.
- Cette méthode est définie dans la classe `World`. Elle doit être appelée depuis un objet de type `World`.

Le mot-clé this

Pour passer un objet en paramètre à la méthode `removeObject`, nous utilisons le mot-clé Java `this`. Ce mot-clé se réfère à l'objet courant, celui dans lequel on se trouve au moment de l'exécution du programme principal. C'est ce qui fait que l'utilisation du mot-clé `this` à cet endroit du programme permet à l'objet de type `Bacteria` de se retirer lui-même du monde.

Appels de méthode « en chaîne »

La méthode `removeObject` est définie dans la classe `World`, mais nous sommes en train d'écrire du code dans la classe `Bacteria`. Nous ne pouvons donc pas écrire un simple appel de méthode du type

```
removeObject(this);
```

Cela reviendrait à essayer d'appeler la méthode depuis l'objet de type `Bacteria`, mais cet objet ne dispose pas d'une telle méthode.
Pour invoquer une méthode d'un autre objet, il nous faut en premier lieu spécifier cet objet, ensuite utiliser un point, pour enfin terminer avec l'appel de méthode :

```
my-world-object.removeObject(this);
```

Nous devons par conséquent trouver un moyen d'accéder à l'objet de type `World` qui a été créé au début du programme, c'est-à-dire le monde dans lequel nous nous trouvons. Par chance, la classe `Actor` dispose justement d'une méthode qui permet précisément de faire cela. C'est la méthode `getWorld()` qui renvoie une référence à l'objet de type `World` courant. Nous pouvons alors utiliser cet appel de méthode à la place de « l'objet monde » :

```
getWorld().removeObject(this);
```

Il s'agit d'une « chaîne d'appels de méthodes » comptant deux méthodes : `getWorld()` et `removeObject(..)`. Tout d'abord, `getWorld()` est appelée et nous renvoie « l'objet monde ». C'est ensuite la méthode `removeObject(..)` qui est appelée depuis l'objet monde qui vient de nous être fourni. Nous sommes donc en train de dire à l'objet monde de nous sortir du monde.

5.4 Circulation du sang : créer de nouveaux objets

La dernière classe qu'il nous faut examiner est la sous-classe de notre monde, nommée `Bloodstream`. Elle dispose du constructeur habituel, contenant un appel à la méthode `prepare()` qui permet de placer les objets de départ (le globule blanc et les bordures supérieure et inférieure). La partie intéressante de cette classe est sa méthode `act` :

```
/**
 * Create new floating objects at irregular intervals.
 */
public void act()
{
    if (Greenfoot.getRandomNumber(100) < 3)
    {
        addObject(new Bacteria(), 779, Greenfoot.getRandomNumber(360));
    }
}
```

Cette méthode va placer de nouveaux objets de type `Bacteria` dans le monde à intervalle irrégulier, choisi aléatoirement, au début de chaque cycle d'exécution du scénario. Chaque fois qu'un nombre choisi aléatoirement parmi les 100 premiers nombres entiers (comptés

à partir de 0) est strictement inférieur à 3 – c'est-à-dire en moyenne 3 fois sur 100 cycles de la méthode `act` – une nouvelle bactérie sera créée.
La valeur de la coordonnée x de ce nouvel objet est systématiquement fixée à 779 – soit le bord droit de l'écran de jeu. Notons que le monde est formé de 780 par 360 cellules, et donc 779 est la plus grande valeur possible pour une coordonnée x.
La coordonnée y est un nombre aléatoire choisi entre 0 et 359 y compris, ce qui correspond à toutes les valeurs possibles pour un monde de hauteur égale à 360. La bactérie peut donc prendre verticalement une position quelconque. Nous avons déjà rencontré cette manière de coder ; elle est ici utilisée dans un autre contexte.

5.5 Mouvement de défilement horizontal

Vous devriez maintenant être capables de comprendre le code dans son état actuel et être prêts à ajouter des objets et des fonctionnalités pour rendre ce projet plus intéressant. Nous allons faire cela au travers d'une série d'exercices appliquant certaines techniques de programmation que nous connaissons déjà.
Tout d'abord, nous désirons créer l'impression que notre globule blanc se déplace en permanence vers la droite dans le flux sanguin. Nous ne voulons toutefois pas réellement déplacer le globule, dans la mesure où il doit rester à l'écran en permanence. Nous ferons plutôt se déplacer l'arrière plan – les objets de type `Lining`, soit les bordures – vers la gauche, dans le but de créer une impression de mouvement. C'est une technique souvent utilisée pour réaliser des jeux vidéos à défilement horizontal.
Pour parvenir à notre fin, nous devons réaliser trois choses :

- Déplacer les objets de type `Lining` qui forment la bordure lentement vers la gauche.
- Enlever ces objets lorsqu'ils atteignent le bord gauche.
- Créer de nouveaux objets de type `Lining` sur la droite, sitôt que nécessaire.

Exercice 5.5

Faire en sorte que les objets de type `Lining` se déplacent continûment vers la gauche. Ils doivent se déplacer d'une cellule par cycle de la méthode `act`. Vous pouvez le faire en copiant la première ligne de la méthode `act` de la classe `Bacteria` et en modifiant l'appel de méthode qui permet le déplacement vers la gauche.

Exercice 5.6

Faire en sorte que les objets de type `Lining` disparaissent lorsqu'ils atteignent le bord gauche. On pourra à nouveau s'inspirer du code que l'on trouve dans la classe `Bacteria`.

Exercice 5.7

Ecrire un commentaire adéquat pour la méthode `act` que vous avez codée en réalisant les deux exercices qui précèdent.

Exercice 5.8

Faire apparaître de nouveaux objets de type `Lining` à la droite de l'écran. Modifier pour cela

la méthode `act` de la classe `Bloodstream`. Les nouveaux objets devraient apparaître avec une probabilité de 1 pour cent, soit en moyenne une fois sur 100 cycles de la méthode `act`.

Le dernier exercice peut être réalisé en copiant deux fois l'instruction `if` de la méthode `act` de la classe `Bloodstream` : une fois pour les objets de type `Lining` du haut de l'écran de jeu et une fois pour ceux du bas. Après quoi il faut remplacer le nombre 3 par le nombre 1 dans cette instruction pour réduire la fréquence d'apparition. Il reste alors à changer la classe de l'objet à créer et à fixer la coordonnée y plutôt que de la générer aléatoirement : $y = 0$ pour le haut de l'écran et $y = 359$ pour le bas. Par exemple, l'instruction

```
addObject(new Lining(), 779, 359);
```

permet d'ajouter un objet de type `Lining` en bas à droite de l'écran de jeu.

En cas de problème, vous trouverez toutes les solutions de ces exercices codées dans le scénario WBC-2.

5.6 Ajouter des virus

FIGURE 5.2 – Un virus

Dans ce jeu, notre tâche est d'attraper des bactéries qui flottent dans la circulation du sang. Pour rendre cette tâche moins facile, nous allons ajouter un facteur de danger sous la forme de virus. Les virus flottent eux aussi dans le flux sanguin, mais notre globule blanc doit les éviter.

Exercice 5.9

En créant une sous-classe de la classe `Actor`, ajouter au scénario une nouvelle classe pour les virus. Son nom doit être `Virus`. Une image de virus est déjà prête à l'utilisation dans le scénario fourni.

Exercice 5.10

Augmenter la méthode `act` de la classe `Bloodstream` de sorte à faire apparaître des virus sur la droite de l'écran de jeu, au fur et à mesure du déroulement de la partie en cours. Faire en sorte que le choix de la coordonnée y soit aléatoire. La probabilité d'apparition de ces nouveaux objets doit être de un pour cent ; ils doivent apparaître en moyenne une fois sur cent cycles de la méthode `act`. Tester votre code en l'état : des virus devraient apparaître de temps en temps sur le bord droit, sans se déplacer, pour l'instant.

Exercice 5.11

Programmer le déplacement et la rotation des objets de type `Virus` : Tout comme les bactéries, ils se déplacent vers la gauche et tournent sur eux-même. Ils doivent par contre bouger de quatre cellules à chaque cycle de la méthode `act` et tourner dans le sens anti-horaire.

Exercice 5.12

Compléter les commentaires de la classe `Virus` : le commentaire du haut de la classe et celui de la méthode `act`.

5.7 Collision : faire disparaître les bactéries

Vérifions maintenant si nous sommes en contact avec une bactérie (et dans ce cas nous l'éliminons), ou avec un virus (dans quel cas nous perdons et le jeu se termine). Cela ressemble de très près avec ce que nous avons fait avec les crabes, vers de sable et autres homards.

Exercice 5.13

Créer une nouvelle méthode privée dans la classe `WhiteCell`, nommée `checkCollision`. Le corps de cette méthode peut être vide au départ. Appeler cette méthode depuis la méthode `act`.

Exercice 5.14

Dans la méthode `checkCollision`, ajouter le code qui élimine toute bactérie que nous serions en train de toucher. Utiliser les méthodes `isTouching()` et `removeTouching()` comme pour le scénario du crabe.

Exercice 5.15

Faire jouer un son lors de l'élimination d'une bactérie. Le son « `slurp.wav` » est à nouveau inclus dans le scénario – vous pouvez l'employer.

Exercice 5.16

Utiliser plusieurs fois le même son est un signe indéniable de paresse. Fabriquer et utiliser un autre son pour accompagner l'élimination des bactéries.

Exercice 5.17

Ajouter encore à la méthode `checkCollision()` une instruction `if` pour tester si nous sommes en contact avec un virus. Si c'est le cas, faire jouer un son (le son « `game-over.wav` » est inclus dans le scénario à cet effet) et stopper l'exécution de Greenfoot.

Une solution à tous les exercices énoncés jusqu'ici figure dans le scénario WBC-2.

Note

Après quelques parties de notre jeu, on ne peut manquer d'observer que la partie s'arrête et qu'elle est par conséquent perdue alors que le globule blanc se trouve proche d'un virus mais

qu'il n'est pas en train de le toucher vraiment. Il y a encore de la distance entre les deux. Ce comportement est causé par la façon dont sont stockées les images dans les ordinateurs. Nous remettons la discussion de ce problème à un chapitre ultérieur.

5.8 Faire varier la vitesse

Pour créer un peu de diversité dans la dynamique du jeu et le rendre un peu plus intéressant, nous voulons faire varier la vitesse de déplacement d'une bactérie. Pour l'instant, cette vitesse est la même pour tous les objets de ce type, qui, rappelons-le, se déplacent vers la gauche de deux cellules à chaque cycle de la méthode `act`. Nous voulons changer cela de manière à ce que la vitesse de déplacement d'une bactérie soit un nombre choisi aléatoirement entre 1 et 3.
Pour réaliser cela, nous créons une variable qui contiendra la vitesse, lui donnons une valeur initiale appropriée et l'utilisons dans l'instruction qui fait bouger la bactérie.

Exercice 5.18

Dans la classe `Bacteria`, ajouter une variable d'instance de type `int`, nommée `speed`.

Exercice 5.19

Dans le constructeur de la classe `Bacteria`, attribuer une valeur aléatoire comprise entre 1 et 3 à la variable `speed`.

Peut-être vous demandez-vous comment générer une valeur comprise entre 1 et 3, alors que nous avons appris plus haut que les nombres aléatoires provenant de la méthode `getRandomNumber` de Greenfoot sont compris entre 0 et une borne supérieure. La réponse est simple : Il suffit d'ajouter 1 à un nombre aléatoire compris entre 0 et 2 (obtenu à l'aide de la méthode ci-dessus).

```
speed = Greenfoot.getRandomNumber(3) + 1;
```

Il est important d'ajouter 1, car nous ne voulons pas d'une vitesse nulle – la bactérie ne se déplacerait jamais.

Exercice 5.20

Modifier l'instruction qui génère le mouvement dans la méthode `act` ; de la coordonnée x, retrancher la vitesse, soit le contenu de la variable `speed`, plutôt que le nombre 2.

Il n'y a rien de plus à faire. Chaque bactérie se voit assigner une vitesse au moment de sa création, comprise entre 1 et 3, et se déplacera à cette vitesse tout au long de sa vie dans le jeu. Si vous y prêtez attention, vous verrez certaines bactéries se déplacer plus vite que d'autres.

5.9 Globules rouges

Ajoutons quelques globules rouges à l'ensemble. C'est purement cosmétique : les objets déjà présents n'interagiront pas avec les globules rouges et ceux-ci n'influenceront en rien le déroulement du jeu. Mais ils sont jolis !

Exercice 5.21

Ajouter à notre scénario une classe nommée `RedCell`. Vous trouverez l'image correspondante dans le scénario en question.

Exercice 5.22

Faire se déplacer les globules rouges comme les bactéries : ils se déplacent de la droite vers la gauche à vitesse variable, en tournant lentement sur eux-même. Il y a une petite différence à respecter : Les globules rouges sont plus lents que les bactéries ; leur vitesse varie entre 1 et 2, plutôt qu'entre 1 et 3.

Exercice 5.23

Ajouter à la méthode `act` de la classe `Bloodstream` le code qu'il faut pour générer les globules rouges. Ce code est très similaire à celui permettant de créer les bactéries ou les virus, mais les globules rouges apparaissent plus souvent : il faut leur donner une probabilité d'apparition de 6 pour cent ; ils seront donc générés 6 fois en moyenne sur 100 cycles de la méthode `act`.

Notre jeu a meilleure allure ! Il commence a y avoir de nombreux objets dans notre circulation sanguine et c'est une bonne chose.
Avant de passer à la tâche suivante, nous pouvons faire encore une amélioration d'ordre cosmétique. Les globules rouges sont tous créés dans la même position, en ce qui concerne la rotation sur eux-même (la valeur de leur « angle de rotation » est fixée à 0 à la création d'un globule rouge). Il en résulte que tous les globules rouges créés presque simultanément tournent de façon synchronisée. Cela manque d'un peu de caractère aléatoire.
Pour remédier à cela, nous voulons que nos globules rouges soient créés avec une « valeur de rotation » aléatoire. Il suffit d'ajouter l'instruction suivante au constructeur de la classe `RedCell` :

```
setRotation(Greenfoot.getRandomNumber(360));
```

Exercice 5.24

Donner une « valeur de rotation » aléatoire au globule rouge, au moment de sa création, en utilisant l'instruction ci-dessus.

5.10 Ajouter des bordures

Nous procéderons à une dernière amélioration d'ordre esthétique avant de passer à des choses plus sérieuses : ajouter des bordures sur la gauche et la droite de l'écran de jeu.
La raison principale pour laquelle nous faisons cet ajout est qu'il faut tenir compte de l'effet suivant : dans Greenfoot, la position d'un objet est fixée relativement aux coordonnées de son centre. Et donc, lorsque nous plaçons un nouvel objet sur le bord droit de l'écran, la moitié de son image apparaît brusquement d'un seul coup, plutôt que lentement et progressivement depuis la droite. Le même problème se produit du côté gauche lorsqu'un objet disparaît. L'objet est supprimé alors que la moitié de son image est encore visible à l'écran. Cela rend saccadé le mouvement de nos objets à certains moments de leur « vie à l'écran » et n'est pas très joli. Pour pouvoir éliminer ce défaut, nous utilisons un truc simple : nous plaçons des objets de bordure sur la gauche et la droite de notre

écran de jeu pour masquer l'endroit qui pose problème. Ces bordures ont été dessinées pour donner l'impression que l'on regarde la circulation du sang à travers un microscope (voir la figure 5.1). Les objets seront maintenant cachés par les bordures tant au moment où ils apparaissent qu'au moment où ils disparaissent ; on aura l'impression de les voir apparaître progressivement depuis la droite et disparaître de la même façon au moment d'atteindre le bord gauche[1].

Exercice 5.25

Créer une nouvelle sous-classe de la classe `Actor`, nommée `Border`. L'image est à disposition dans le scénario.

Nous pourrions maintenant ajouter un objet de bordure à gauche et à droite de l'écran à la main et utiliser la fonction Save the World du menu Controls de Greenfoot pour les ajouter de manière permanente à notre scénario. Nous préférons plutôt les placer à des endroits très précis, et cela est difficile à faire à la main. Il nous est donc plus facile d'écrire un peu de code pour positionner ces objets.
Dans la méthode `prepare()` de la classe `Bloodstream`, il nous faut ajouter les instructions suivantes :

```
Border border = new Border();
addObject(border, 0, 180);
Border border2 = new Border();
addObject(border2, 770, 180);
```

Exercice 5.26

Ajouter le code ci-dessus à la méthode `prepare()` de la classe `Bloodstream`. Tester le scénario. Quels sont les problèmes que vous pouvez observer ?

Nous devons régler deux problèmes mineurs : primo, notre globule blanc est partiellement caché par la bordure de gauche et secundo, nos objets apparaissent en dessus de la bordure plutôt qu'en dessous. Remédions à tout cela.

Exercice 5.27

Dans la méthode `prepare()` de la classe `Bloodstream`, trouver l'endroit où la coordonnée x du globule blanc est définie à la création du jeu. Augmenter la valeur de cette coordonnée, de façon à ce que le globule apparaisse suffisamment à droite.

Exercice 5.28

Dans le constructeur de la classe `Bloodstream`, ajouter l'instruction suivante :

```
setPaintOrder(Border.class);
```

Cela fera apparaître les objets de type `Border` en dessus de tous les autres.

1. Il y a une autre façon de régler le problème : on pourrait créer un *monde sans limites*, ce qui nous permettrait de placer des acteurs hors des limites et de les faire entrer lentement dans le périmètre visible. Cela peut être fait en invoquant un autre constructeur de la super-classe `World`, qui est décrit dans la documentation de cette classe. C'est un peu plus compliqué à réaliser et nous ne le ferons pas ici.

La méthode `setPaintOrder` nous permet de spécifier quels objets devraient être dessinés par dessus les autres objets. La méthode admet une liste de paramètres de longueur variable et nous pouvons spécifier autant de classes que nous le désirons. Par exemple :

```
setPaintOrder(Class1.class, Class2.class, Class3.class);
```

Dans l'exemple ci-dessus, les objets de la classe `Class1` seraient dessinés tout dessus ; juste en dessous viendraient les objets de la classe `Class2`, suivis par ceux de la classe `Class3` et, ensuite seulement, par tous les objets dont la classe n'est pas mentionnée dans la liste des paramètres, dans un ordre non spécifié.
En écrivant

```
setPaintOrder(Border.class);
```

nous spécifions que les objets de type `Border` doivent être tout dessus, et nous ne nous préoccupons pas de l'ordre dans lequel les autres seront dessinés.
Toutes les solutions aux exercices dont nous avons discuté jusque là figurent dans le scénario WBC-3 et si vous avez rencontré des problèmes, vous pouvez comparer votre scénario à celui-là.

Exercice 5.29

Faire en sorte que le globule blanc puisse non seulement se déplacer verticalement, mais également horizontalement, tant vers la gauche que vers la droite.

5.11 Finalement : ajouter le score

Nous allons maintenant mettre en place les éléments nécessaires au décompte du score. Il nous faudra tout d'abord déterminer quels sont ces éléments et, ensuite, écrire le code correspondant. La première idée qui vient à l'esprit est que le joueur doit obtenir des points chaque fois qu'il neutralise une bactérie. Mettons que le joueur obtienne 20 points chaque fois qu'il attrape une bactérie.
Nous allons, pour commencer, ajouter du code à la classe `WhiteCell`. Il nous faut une variable entière pour contenir notre score, nous devons modifier le score chaque fois que nous détruisons une bactérie et afficher le résultat à l'écran.

Exercice 5.30

Ajouter une variable d'instance de type `int` nommée `score` à votre classe `WhiteCell`.

Exercice 5.31

Ajouter une instruction qui permet d'ajouter 20 à la variable définie dans l'exercice précédent chaque fois que le joueur attrape une bactérie.

Nous avons vu dans le chapitre précédent comment ajouter une variable d'instance. Nous écrivons :

```
private int score;
```

au tout début de la classe, avant la première méthode.

Nous pouvons ensuite ajouter 20 points au score en écrivant l'instruction suivante directement après avoir enlevé l'objet `Bacteria` :

```
score = score + 20;
```

Il nous reste à afficher le score à l'écran. Pour le faire, nous utilisons une méthode appelée `showText(..)`. Cette méthode est définie dans la classe `World` comme on peut le constater dans la documentation de cette classe.

Exercice 5.32

Rechercher la méthode `showText` de la classe `World`. De combien de paramètres dispose-t-elle ? Quels sont ces paramètres ?

A nouveau, il nous faut appeler une méthode de la classe `World`, comme nous l'avons déjà fait plus haut pour supprimer un objet du monde. Nous utilisons la même technique qu'avant : nous appelons `getWorld()` pour obtenir l'accès à l'objet monde de notre scénario, et nous « chaînons » ensuite notre appel à la méthode `showText` à la suite :

```
getWorld().showText(..)
```

Il nous reste à déterminer quels paramètres nous allons passer à la méthode `showText`. Son en-tête est :

```
void showText(String text, int x, int y)
```

Les deux derniers paramètres sont simplement la coordonnée x et la coordonnée y de l'emplacement où nous voulons faire apparaître notre texte. Le premier paramètre est le texte que nous voulons montrer, et il est d'un type que nous avons déjà rencontré mais dont nous n'avons pas parlé formellement : `String`.
Les variables de type `String` peuvent stocker du texte, soit des caractères, des mots ou des phrases. On appelle de telles variables des *chaînes de caractères* et on les écrit entre guillemets lorsqu'on les affecte à une variable. Par exemple :

```
String name = "Fred";
String message = "Game over";
```

Nous avons vu une chaîne de caractères plus haut lorsque nous avons fait jouer un son :

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

La méthode `playSound` prend en paramètre une chaîne de caractères, et la valeur `"slurp.wav‘‘` en est une que nous lui passons. Nous pouvons faire de même avec la méthode `showText`. Par exemple,

```
getWorld().showText("Hello", 80, 25);
```

fera afficher à l'écran le mot « Hello » à l'emplacement spécifié.

Exercice 5.33

Ajouter l'instruction suivante au code de la classe `WhiteCell` de sorte que le mot « Hello » s'affiche à l'écran lorsque le joueur attrape une bactérie.

Exercice 5.34

Changer l'instruction pour que ce soit votre nom qui s'affiche à la place.

Exercice 5.35

Changer à nouveau le texte pour faire apparaître le mot « Score : », y compris les deux points.

Nous avons vu comment afficher du texte ; il nous reste encore à trouver comment afficher notre score qui est contenu dans une variable de type `int`. Nous allons utiliser la *concaténation des chaînes de caractères*.
On écrit la concaténation des chaînes de caractères à l'aide du signe « + » qui permet de joindre deux chaînes :

```
"abc" + "def"
```

devient

```
"abcdef"
```

et

```
"Wolfgang" + "Amadeus" + "Mozart"
```

devient

```
"WolfgangAmadeusMozart"
```

Notons que l'opérateur de concaténation n'ajoute pas automatiquement des espaces. Si vous désirez un espace entre deux parties, vous devrez l'écrire vous-même.
La même manière d'écrire peut être utilisée pour insérer un entier dans une chaîne de caractères. Si nous « additionnons » une chaîne et un entier, la valeur entière est convertie en une chaîne de caractères et concaténée ensuite.

```
"Score: " + 20
```

devient

```
"Score: 20 "
```

Nous pouvons également utiliser le nom d'une variable à la place de la valeur entière. Au moment d'évaluer l'expression

```
"Score: " + score
```

la valeur stockée dans notre variable `score` sera convertie en une chaîne de caractères, et ensuite ajoutée à la chaîne `"Score: "`. Nous disposons maintenant de tout ce qu'il nous faut pour faire afficher notre score à l'écran.

Exercice 5.36

Faire afficher le score dans votre jeu.

Exercice 5.37

Le jeu tel qu'il est maintenant semble un peu trop facile. Accélérez-le ! Vous pouvez tester les valeurs suivantes : Les bactéries ont une vitesse aléatoire donnée en cellules par cycle d'action dont la valeur est comprise entre 1 et 5. Le globule blanc se déplace latéralement à une vitesse de 4 et, verticalement, à une vitesse de 8. Les virus se déplacent avec une vitesse de 8.

Vous pouvez également accélérer le scénario en utilisant le curseur de vitesse situé en bas de la fenêtre principale : ajustez-le à un peu plus de 50 pour cents. Testez avec vos propres vitesses pour rendre le jeu difficile mais sans le rendre impossible.

La solution des exercices précédents figure dans l'extrait de code ci-dessous :

```
public class WhiteCell extends Actor
{
    private int score;

    // Other methods omitted

    private void checkCollision() {
        if (isTouching(Bacteria.class)) {
            Greenfoot.playSound("slurp.wav");
            removeTouching(Bacteria.class);
            score = score + 20;
            getWorld().showText("Score: " + score, 80, 25);
        }
        if (isTouching(Virus.class)) {
            Greenfoot.playSound("game-over.wav");
            Greenfoot.stop();
        }
    }
}
```

5.12 Gérer le score depuis la classe monde

Jusqu'ici, ajouter l'affichage du score a été fait de manière directe et plutôt simple. Il y a toutefois un désavantage à gérer le score et son affichage depuis la classe `WhiteCell` : cela rend difficile l'ajout des scores liés à d'autres événements.
Supposons maintenant que nous voulions que nos règles de décompte des points soient les suivantes :

- On obtient 20 points pour chaque bactérie neutralisée.
- On perd 15 points par bactérie qui atteint le bord gauche de l'écran de jeu.
- Le fait de se faire heurter par un virus ne stoppe pas immédiatement le jeu ; par contre, cela fait perdre 100 points.
- Si le nombre total de points passe en dessous de zéro, le joueur a perdu et le jeu s'arrête.

Cette façon de déterminer le score est plus intéressante, mais elle pose un problème. Le deuxième événement, soit une bactérie qui s'échappe, sera visible dans l'objet de type `Bacteria`. Depuis cet endroit, nous n'avons pas accès à notre variable `score`, qui est définie dans l'objet de type `WhiteCell`. Nous pourrions tenter d'accéder à l'objet globule blanc pour lui faire mettre à jour le score. Il est toutefois plus facile de déplacer la variable `score` dans notre classe `Bloodstream` qui est une sous-classe de la classe `World`, dans la

mesure où tous les objets du scénario peuvent facilement accéder à cet objet en utilisant la méthode `getWorld()`.
Après déplacement de cette variable, l'objet de type `Bloodstream` contient le score courant, et les objets `WhiteCell` et `Bacteria` pourront appeler l'objet `Bloodstream` pour lui faire mettre à jour le score lorsque c'est nécessaire.

Exercice 5.38

Déplacer la variable `score` de la classe `WhiteCell` à la classe `Bloodstream`.

Exercice 5.39

Ajouter une ligne au constructeur de la classe `Bloodstream` pour initialiser le décompte du score à zéro. Cela n'est pas absolument nécessaire, car la valeur par défaut pour une variable d'instance entière est zéro, mais c'est une bonne chose à faire tout de même.

Exercice 5.40

Ajouter à la classe `Bloodstream` une méthode publique nommée `addScore()`. Déplacer le code qui permet l'incrémentation et l'affichage du score dans cette méthode. Notons qu'il n'est plus nécessaire d'appeler la méthode `getWorld()` avant la méthode `showText()` vu que nous sommes à présent dans une classe de type `World`.

Exercice 5.41

Ecrire un commentaire qui s'applique à la nouvelle méthode.

On donne ci-dessous le code de la méthode `addScore()` :

```
public void addScore()
{
    score = score + 20;
    showText("Score: " + score, 80, 25);
}
```

Finalement, nous devons faire en sorte que l'objet `WhiteCell` envoye un message à l'objet monde lorsqu'il désire qu'un score soit modifié. On pourrait essayer d'écrire l'instruction suivante dans la classe `WhiteCell` :

```
getWorld().addScore();
```

C'est une idée raisonnable, mais elle ne fonctionnera pas à cause d'un problème subtil : la méthode `getWorld()`, telle que définie dans la classe `World` de Greenfoot, nous renvoie un objet de type `World`. Et la classe `World` ne dispose pas d'une méthode nommée `addScore()`. Par contre, *notre* objet monde est en fait de type `Bloodstream` et dispose en conséquence de cette méthode.
Nous devons utiliser une « construction » du langage Java, qui s'appelle un *cast* pour dire au compilateur de Greenfoot que notre monde est de type `Bloodstream` et le stocker ensuite dans une variable de ce type. Après avoir fait cela, nous pouvons appeler la méthode `addScore`.

L'opération de cast s'écrit en indiquant le type entre parenthèses avant l'appel de méthode, comme montré ci-dessous :

```
Bloodstream bloodstream = (Bloodstream)getWorld();
```

On peut ensuite appeler la méthode `addScore()` depuis cet objet :

```
bloodstream.addScore();
```

Le « casting » est un concept difficile à comprendre au premier abord, et il vous faudra probablement un moment pour le saisir dans son ensemble. Nous reprendrons plus en détail l'étude de cette façon de programmer plus loin, lorsque nous en saurons plus sur les types et leurs subordonnés.

Exercice 5.42

Ajouter au code de la classe `WhiteCell` l'appel à la méthode `addScore` de la classe `Bloodstream`. Tester le scénario. Votre jeu devrait maintenant fonctionner normalement, et attraper des bactéries devrait rapporter des points.

Vu de l'extérieur, nous en sommes de nouveau au point que nous avions déjà atteint : attraper des bactéries nous rapporte des points, et tout changement de score est affiché à l'écran. Déplacer le code de gestion du score de la classe `WhiteCell` à la classe `Bloodstream` n'a pas eu d'effet visible à ce stade. Mais nous avons maintenant un gros avantage par rapport à la situation précédente : nous pouvons modifier notre score en fonction d'autres événements.

5.13 Un peu d'abstraction pour un score plus global

La tâche suivante est de faire respecter la deuxième règle de décompte des points : nous perdons 15 points lorsqu'une bactérie s'échappe par la gauche de l'écran.
L'appel à la méthode `addScore()` de la classe `Bloodstream` depuis la classe `Bacteria` est maintenant chose facile à réaliser en ajoutant à cette classe le même code que nous avons utilisé dans la classe `WhiteCell`. Le problème est que cela nous ajoute 20 points au score au lieu d'en enlever 15 !
Il serait possible de créer dans la classe `Bloodstream` une nouvelle méthode, que nous pourrions appeler `perdrePoints()` par exemple, qui nous permettrait de soustraire 15 points à notre score et faire un appel à cette méthode dans notre code. Bien que cette façon de procéder donne le résultat escompté, nous allons employer une solution plus élégante : nous allons généraliser notre méthode `addScore` pour pouvoir l'employer à la fois pour ajouter et ôter des points.
Nous pouvons faire cela en changeant la méthode `addScore()` de façon à ce qu'elle attende un paramètre entier positif ou négatif, donnant le nombre de points à ajouter ou à enlever, comme on peut le voir dans l'extrait de code ci-dessous :

```
public void addScore(int points)
{
        score = score + points;
        showText("Score: " + score, 80, 25);
}
```

La méthode `addScore` ainsi améliorée reçoit maintenant le nombre de points à ajouter au score en paramètre. Lorsque le nouveau score est construit à partir de l'ancien, nous lui ajoutons le contenu du paramètre `points` au lieu de simplement lui ajouter 20.
Nous pouvons maintenant utiliser cette méthode pour modifier le score de différentes façons : l'instruction

```
bloodstream.addScore(20);
```

ajoute 20 points au score courant, tandis que

```
bloodstream.addScore(100);
```

lui en ajoute 100. Nous pouvons aussi ôter facilement des points au score en utilisant un nombre négatif :

```
bloodstream.addScore(-15);
```

Ce que nous venons de faire ici est un exemple d'*abstraction* – un concept important en programmation. A la place d'écrire une méthode qui peut faire une seule chose très spécifique, soit *ajouter 20 points*, nous écrivons une méthode qui peut faire un grand nombre d'opérations *similaires*, soit *ajouter un nombre quelconque de points* (y compris les nombres de points négatifs) ; cette dernière méthode peut alors être utilisée dans un cadre plus général. Elle permet d'appliquer un schéma de calcul à des situations différentes. C'est souvent une bonne idée que de généraliser une méthode en lui ajoutant des paramètres pour la rendre plus souple d'utilisation.

Exercice 5.43

Changer la méthode `addScore` de façon à ce qu'elle attende un paramètre, comme discuté ci-dessus.

Exercice 5.44

Changer le code de la classe `WhiteCell` en conséquence : passer un paramètre à la méthode `addScore`, de façon à ce que neutraliser une bactérie rapporte toujours 20 points.

Exercice 5.45

Ajouter du code à la classe `Bactérie`, de façon que le joueur perde 15 points lorsqu'une bactérie quitte l'écran.

Lorsque nous ajoutons le code qui permet la gestion du score à la classe `Bacteria`, il est important de ne pas appeler la méthode `getWorld()` *après* avoir retiré l'objet du monde. Si l'objet a été retiré du monde, un appel à la méthode `getWorld` après coup ne fonctionnera pas, car cet objet ne fait plus partie d'un monde. La méthode `getWorld()` renverrait alors l'objet spécial `null`, et essayer d'appeler une méthode de la classe `World` provoquerait une erreur de type `NullPointerException`.

Le code ci-dessous a été écrit pour éviter ce problème :

```
/**
* Float along the bloodstream, slowly rotating.
*/
    public void act()
    {
        setLocation(getX() - speed, getY());
        turn(1);

        if (getX() == 0)
        {
            Bloodstream bloodstream = (Bloodstream)getWorld();
            bloodstream.addScore(-15);
            bloodstream.removeObject(this);
        }
    }
```

Exercice 5.46

Dans la classe `Bloodstream`, créer une nouvelle méthode qui s'appelle `showScore()`. Elle ne prend pas de paramètres et ne renvoie rien. Déplacer dans cette méthode l'instruction `showText` qui fait afficher le score. A l'endroit où se trouvait cette instruction, écrire un appel à la méthode `showScore` nouvellement créée.

Exercice 5.47

Dans le constructeur de la classe `Bloodstream`, ajouter un appel à la méthode `showScore`. Cela fera afficher le score dès le départ.

En plus de faire apparaître le score à la construction de l'objet de type `Bloodstream`, nous avons également créé une méthode spéciale pour l'affichage de ce score. Il est toujours bien de créer une méthode dédiée pour toute tâche nécessitant d'être effectuée plus d'une fois, même si cette tâche est de faible ampleur.
Nous sommes maintenant prêts à programmer nos deux dernières règles de calcul du score.

Exercice 5.48

Changer votre programme de façon à ce que le fait de toucher un virus ne fasse pas se terminer le jeu tout de suite. A la place, le virus est détruit et vous perdez 100 points.
Il est important d'enlever le virus. Sinon, nous le toucherions un grand nombre de fois et nous perdrions 100 points à chaque cycle de la méthode `act`.

Exercice 5.49

Faire jouer un nouveau son lorsque le globule blanc entre en contact avec un virus.

Exercice 5.50

Déplacer la gestion de fin de partie (faire jouer le son de fin de partie et stopper Greenfoot) dans la méthode `addScore()`, de sorte à ce que le jeu se termine dès que le score passe

en dessous de zéro. Il vous faudra utiliser une instruction conditionnelle `if` placée après les instructions qui modifient et font afficher le score.

Si vous rencontrez des problèmes, le scénario `WBC-4` contient tout le code dont il est question ci-dessus (avec également des modifications proposées plus loin). Essayez tout de même de l'écrire entièrement par vous-même. Vous devriez à ce stade disposer de toutes les connaissances nécessaires.

5.14 Ajouter un compte à rebours

La dernière chose qui nous manque maintenant est un moyen de gagner ce jeu.
Nous allons permettre cela en ajoutant un compte à rebours qui montre le temps restant à jouer. Si le score est resté en dessus de zéro durant tout le compte à rebours, la partie est gagnée.
Ajouter le compte à regours utilise des techniques de programmation similaires à celles utilisées pour ajouter le décompte du score ; nous devrions donc être capables de le faire en suivant les instructions données dans la liste d'exercices ci-dessous.

Exercice 5.51

Dans la classe `Bloodstream`, ajouter une variable d'instance de type `int`, appelée `time`.

Exercice 5.52

Dans le constructeur, initialiser la variable `time` à 2000. (Nous allons tenter de survivre durant 2000 cycles de la méthode `act`.)

Exercice 5.53

Définir une nouvelle méthode privée appelée `showTime` qui permet d'afficher en haut à droite de l'écran la valeur de la variable `time`, soit le temps restant à jouer. Appeler cette méthode depuis le constructeur pour montrer la valeur initiale de cette variable.

Exercice 5.54

Définir une nouvelle méthode privée appelée `countTime` qui diminue la valeur de la variable `time` de 1 chaque fois qu'elle est appelée et qui fait afficher ensuite la valeur courante de cette variable en appelant à son tour la méthode `showTime`. Ecrire un appel à cette méthode depuis la méthode `act` de la classe `Bloodstream`.

Exercice 5.55

Dans votre méthode `countTime`, ajouter une instruction `if` qui stoppe l'exécution lorsque le compte à rebours atteint 0.

Exercice 5.56

Ajouter une nouvelle méthode privée dont le nom est `showEndMessage`. Lorsqu'elle est appelée, elle fait afficher un message au centre de l'écran qui nous dit que nous avons gagné et quel est notre score. Par exemple :

```
Le temps à disposition est écoulé -- Vous avez gagné!!!
Votre score final: 1425 points
```

Ajouter un appel à cette méthode au moment où le compte à rebours est terminé et où nous avons gagné.

5.15 Résumé des techniques de programmation

Dans ce chapitre, nous avons partiqué de manière plus approfondie des concepts importants qu'il nous est vraiment nécessaire de connaître : les variables et les méthodes. Plus précisément : la déclaration et l'affectation de variables et la définition et l'appel de méthodes.
Nous avons également vu un premier exemple d'interaction entre objets : notre objet acteur a été amené à appeler l'objet de type `World` pour lui demander de réaliser quelque chose (changer le score ou enlever un objet).
Nous avons découvert le type `String` et appris à concaténer deux chaînes de caractères.
Il est important de noter que nous avons vu notre premier exemple de « généralisation du code par l'abstraction » : nous avons ajouté un paramètre à une méthode pour la rendre utilisable dans un contexte plus étendu, c'est à dire de façon à pouvoir l'appeler depuis différents endroits de notre code. Cela permet de réaliser des actions pas tout à fait identiques mais liées par un but commun. C'est mieux que d'écrire plusieurs méthodes permettant de traiter individuellement chaque cas.
Nous travaillerons l'abstraction plus encore dans les chapitres à venir.

5.16 Concepts du chapitre

- La méthode `setLocation` permet de définir la position d'un acteur à l'aide de deux coordonnées : x et y.
- Les *modificateurs d'accès* (`private` ou `public`) permettent de déterminer qui peut appeler une méthode.
- Les *méthodes privées* ne sont visibles que depuis la classe dans laquelle elles sont définies. Elles sont utilisées pour améliorer la structure du code.
- Le mot-clé `this` peut être utilisé pour se référer à l'objet courant.
- La méthode `getWorld` nous donne accès au « monde » depuis un objet acteur.
- Le type `String` est utilisé pour représenter du texte, tel que des mots ou des phrases. En français, on parle de chaînes de caractères. On écrit une chaîne de caractères entre guillemets.
- Les variables de type `String` permettent de stocker des objets « chaînes de caractères ». Ce sont des *références* à ces objets.
- La *concaténation* permet de « fusionner » deux chaînes en une. L'opérateur de concaténation s'écrit à l'aide d'un signe +.
- La *concaténation* peut également être utilisée pour combiner une chaîne avec un entier.
- *Ajouter des paramètres* à des méthodes peut les rendre plus souples et plus utiles.

5.17 Un peu de pratique

Exercice 5.57

Faire des tests en modifiant les paramètres du jeu pour le rendre plus amusant et plus intéressant. Vous pouvez modifier :

a) la vitesse à laquelle chacun des acteurs du jeu se déplace ;

b) le nombre de points que vous gagnez et perdez ;

c) la quantité de temps à disposition ;

d) les effets sonores ;

e) la fréquence à laquelle de nouveaux acteurs apparaissent ;

f) la vitesse d'exécution du scénario.

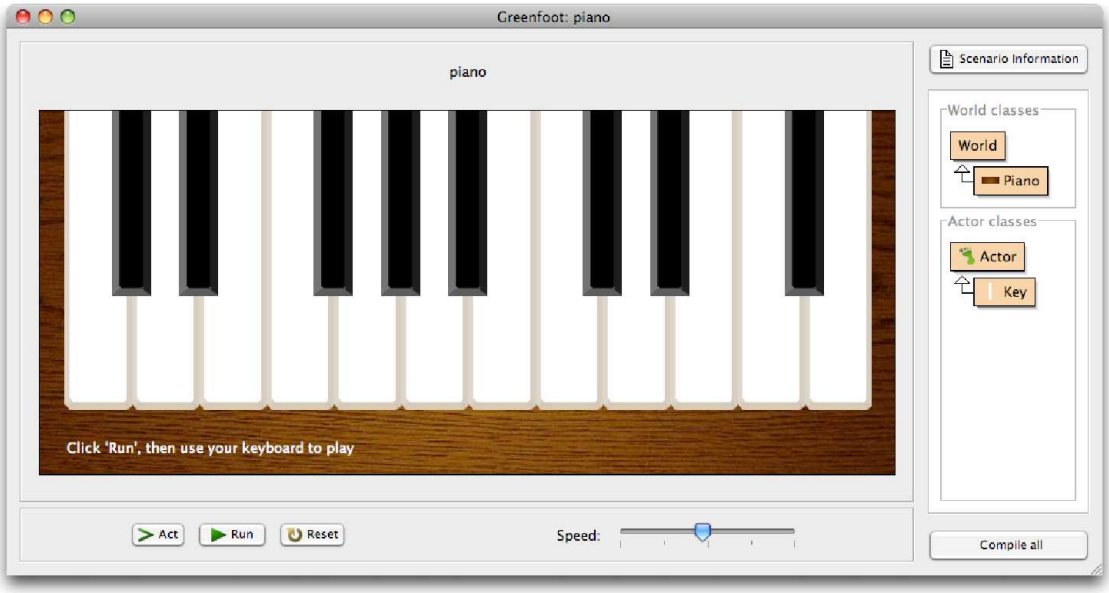
Exercice 5.58

Changez les images du scénario pour le modifier complètement. Par exemple, on pourrait imaginer un vaisseau se déplaçant dans l'espace, récupérant des astronautes et évitant des astéroïdes ; ou encore un lapin courant dans un champ, attrapant des carottes et évitant des chiens.
Tout ce que vous voudrez. Inventez quelque chose ; soyez créatifs.
Lorsque vous avez modifié les images et que cela vous plaît, il faudrait changer les noms des classes, également.
Modifier l'aspect visuel de votre jeu pourrait vous donner des idées de nouvelles fonctionnalités.

Chapitre 6

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 6.1

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 6.2

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

6.1 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 6.3

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 6.4

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

Exercice 6.5

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.

6.2 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(''3a.wav'');
```

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 6.6

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

Exercice 6.7

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 6.8

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 6.9

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.

6.3 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 6.10

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 paramètres pour initialiser les champs.

Exercice 6.11

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.

6.4 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 6.12

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

Exercice 6.13

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 6.14

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 6.15

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.

6.5 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 déclarées 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 6.16

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 6.17

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 6.18

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 63 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 6.19

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*.

6.6 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";
```

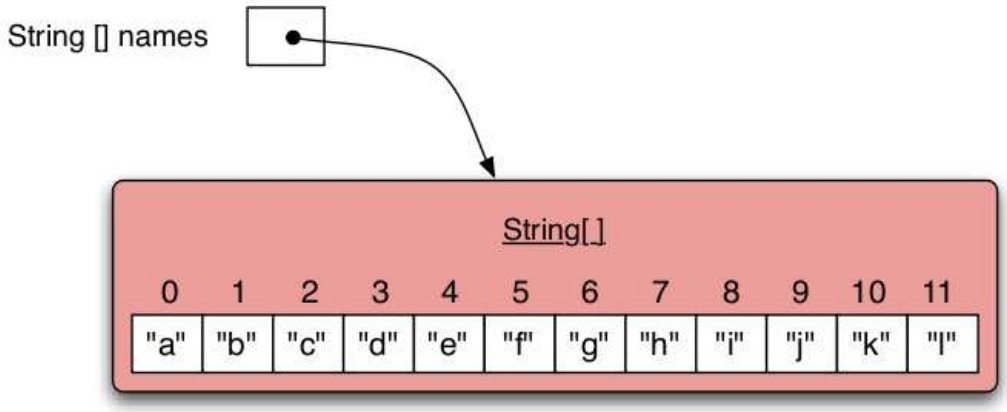
Ce cas de figure est très simple, la variable est une boîte qui peut contenir une seule valeur. La valeur est stockée dans la variable.
Dans le cas d'un tableau, nous créons un objet séparé qui peut contenir plusieurs variables. Nous pouvons alors stocker une référence à cet objet tableau dans une variable 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* et 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 6.20

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.

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 6.21

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 6.22

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

Exercice 6.23

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

Exercice 6.24

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 6.25

L'implémentation 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.

6.7 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.

6.8 Concepts du chapitre

- *Les opérateurs logiques*, comme `&&` (et) et `!` (non) peuvent être utilisés pour combiner plusieurs expressions booléennes en une seule expression booléenne.
- En programmation, l'*abstraction* se présente sous de nombreuses formes différentes. L'une d'entre elles est la technique qui consiste à écrire du code qui permet de résoudre toute une classe de problèmes, plutôt qu'un seul problème spécifique.
- Une *boucle* est une instruction d'un langage de programmation qui permet de faire exécuter du code plusieurs fois.
- Une *variable de boucle* est une variable locale qui est utilisée pour compter le nombre d'itérations dans une boucle. Pour ce qui est de la boucle `while`, cette variable doit être déclarée immédiatement avant la boucle.
- Un *tableau* est un objet qui contient plusieurs variables du même type. On peut accéder à chacune de ces variables en utilisant un *indice*.
- Un *élément* d'un tableau est référencé à l'aide des crochets (`[]`) et d'un *indice* qui permet de spécifier la position de cet élément dans le tableau.
- Le type `String` est défini par une classe standard. Cette classe dispose de nombreuses méthodes utiles, que nous pouvons consulter dans la *bibliothèque de la documentation Java*.

6.9 Un peu de pratique

Cette fois, nous allons mettre en pratique la technique de programmation que nous venons d'apprendre : l'utilisation de la boucle `while`. Nous le ferons avec un scénario différent appelé bubbles (voir figure 6.1). Ouvrez le scénario, et faites les exercices ci-dessous.

Exercice 6.26

Ouvrez le scénario bubble. Vous verrez que le monde est vide. Placez quelques objets de type `Bubble` dans le monde en utilisant le *constructeur par défaut*. (Le constructeur par défaut eest celui qui ne prend aucun paramètre.) souvenez-vous : vous pouvez également faire cela à l'aide d'un clic droit dans le monde. Que pouvez-vous observer après avoir créé plusieurs objets ?

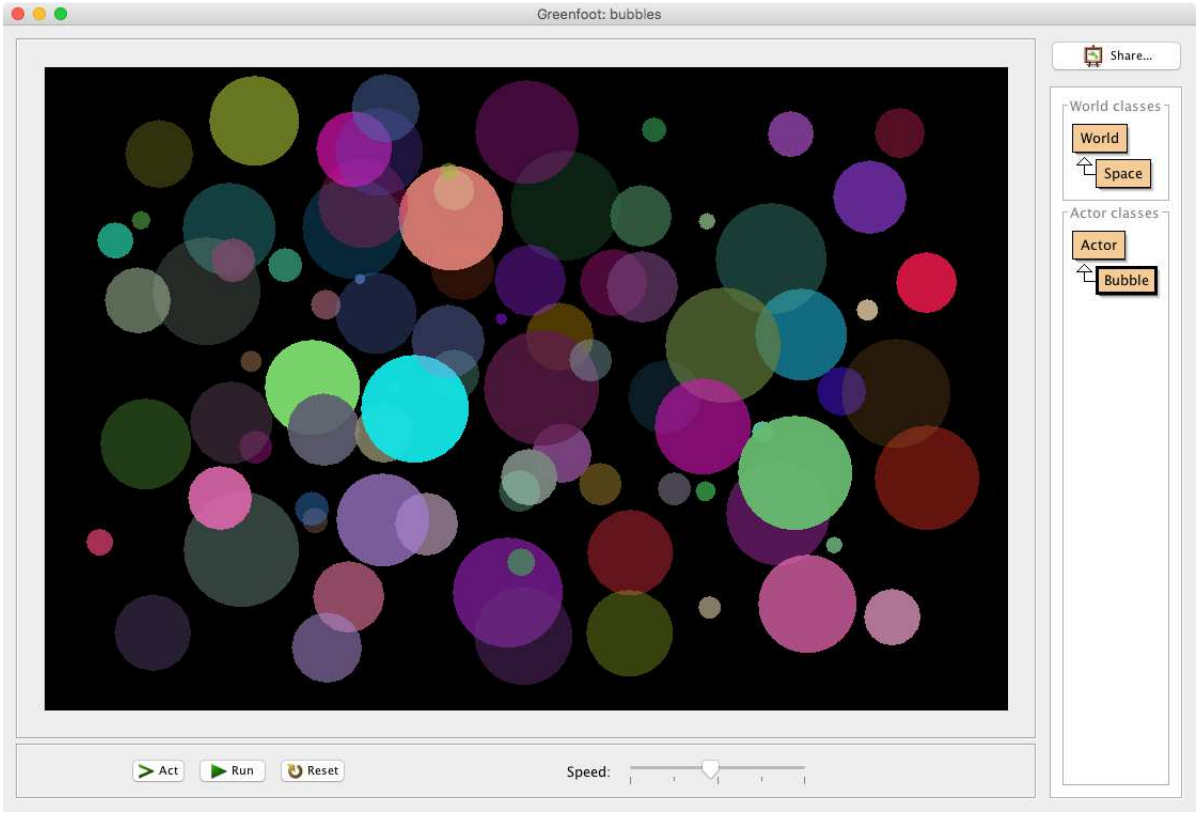


FIGURE 6.1 – Des bulles flottantes

Exercice 6.27

Dans `Space`, la sous-classe de la classe monde, créer une nouvelle méthode privée, nommée `setup()`. Appeler cette méthode depuis le constructeur. Dans cette méthode, créer une nouvelle bulle, en utilisant le constructeur par défaut, et la placer au centre du monde. Cliquer sur Reset un certain nombre de fois pour tester.

Exercice 6.28

Modifier votre méthode `setup()` de manière à ce qu'elle provoque la création de 21 bulles, en utilisant une boucle `while`. Toutes les bulles sont placées au centre de l'écran. Faites tourner le scénario pour tester.

Exercice 6.29

Faire en sorte que les 21 bulles se placent aléatoirement dans le monde.

Exercice 6.30

Placer les bulles sur une droite oblique de sorte que la position du centre de la première soit $(0, 0)$, de la seconde soit $(30, 30)$, de la troisième soit $(60, 60)$ etc. Les coordonnées du centre de la dernière des 21 bulles seront $(600, 600)$.

Exercice 6.31

Placer cette fois les centres des bulles sur la diagonale descendante du rectangle noir qui forme le fond de notre monde. La première bulle doit être positionnée au coin en haut à gauche du rectangle et la dernière en bas à droite. Les coordonnées du centre de la dernière bulle sont $(900, 600)$.

Exercice 6.32

Ajouter une deuxième boucle `while` à la méthode `setup()` pour pouvoir ajouter d'autres bulles. Cette boucle place 10 bulles sur une ligne horizontale, débutant au point $(x, y) = (300, 100)$ et augmentant x de 40 à chaque itération ; y reste constant. Les coordonnées des centres seront donc $(300, 100)$, $(340, 100)$, $(380, 100)$ et ainsi de suite. La taille de la bulle doit également augmenter, commençant à 10 pour la première bulle puis 20 pour la seconde, 30 pour la troisième, etc. Utiliser le deuxième constructeur de la classe `Bulle` pour réaliser cela (le constructeur qui prend un seul paramètre).

Exercice 6.33

Enlever les boucles existantes de votre méthode `setup()`. Ecrire une nouvelle boucle `while` qui permet de créer 18 bulles. Ces bulles seront toutes placées au centre du monde et leur taille commence à 190 et décroit de 10 à chaque itération de la boucle. La dernière bulle est de taille 10. Faire en sorte de créer la plus grande en premier et la plus petite en dernier, de sorte à obtenir des bulles de taille 190, 180, 170, ..., toutes les unes sur les autres.

Utiliser le troisième constructeur de la classe `Bubble`, celui qui prend deux paramètres. Il permet de spécifier non seulement la taille, mais également la direction initiale du mouvement. Fixer la direction comme suit : la première bulle a la direction 0, la deuxième la direction 20, la suivante 40, etc. C'est la dire que la direction entre deux bulles consécutives augmente de 20 degrés à chaque fois.

Tester le résultat.