2001-2002

UFR Mathématiques de la Décision

Première Année de DEUG Sciences et Technologie mention MASS

Informatique

Volume III

Fabrice Rossi

Conditions de distribution et de copie

Table des Matières

CHAPITRE 8

Graphisme et méthodes

Sommaire

Introduction

Nous continuons dans ce chapitre la découverte du graphisme entamée au chapitre 6. Nous commencerons par revenir sur certains éléments présentés dans ce chapitre à la lumière des acquis du chapitre 7. Nous utiliserons ensuite le graphisme comme illustration de l'intérêt des méthodes, en montrant en particulier comment on peut travailler dans un repère mathématique classique en introduisant des méthodes de changement de repère. Nous verrons aussi comment obtenir les dimensions effectives de la fenêtre dans laquelle nous dessinons, afin d'adapter le dessin à la place disponible. Nous terminerons le chapitre par des exemples d'application à des objets définis mathématiquement (graphe d'une fonction et courbe paramétrique).

8.1 La méthode `paintComponent`

Nous avons présenté au chapitre 6 (et en particulier dans la section 6.1.1) la forme générale d'un programme graphique, sans donner de précision sur le sens des éléments qui la constituent.

Grâce aux connaissances acquises au chapitre 7, nous pouvons maintenant donner quelques détails. Il est clair en particulier qu'un programme graphique est constitué d'au moins deux méthodes, comme dans l'exemple élémentaire suivant :

Appel

```
1 import javax.swing.*;
2 import java.awt.*;
3 public class Appel extends JPanel {
4   public void paintComponent(Graphics g) {
5     super.paintComponent(g);
6     System.out.println("paintComponent");
7   }
```

```
8   public static void main(String[] args) {
9     JFrame fenêtre=new JFrame();
10    fenêtre.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
11    JPanel dessin=new Appel();
12    dessin.setPreferredSize(new Dimension(250,200));
13    fenêtre.getContentPane().add(dessin,BorderLayout.CENTER);
14    fenêtre.pack();
15    fenêtre.setVisible(true);
16  }
17 }
```

On repère facilement les méthodes `paintComponent` et `main`. Deux remarques s'imposent cependant :

1. On constate que l'en-tête de la méthode `paintComponent` ne comporte pas le mot clé `static`. Comme nous l'avons indiqué brièvement à la section 7.1.1, cela signifie que la méthode définie est une méthode d'*instance* et non pas une méthode de classe. De fait, le graphisme impose la création d'objets, technique que nous aborderons au chapitre 12. Du point de vue pratique cependant, cela ne change pas grand chose : la méthode s'exécute "normalement" une fois qu'elle a été appelée.

2. On remarque que la méthode `paintComponent` n'est pas appelée par la méthode `main`. Pourtant, on sait que l'exécution d'un programme début par celle de la méthode `main` et que le graphisme proprement dit est réalisé par la méthode `paintComponent`. En fait, les nombreuses instructions de la méthode `main` ont pour conséquence la mise en place d'un mécanisme complexe qui rend l'appel à `paintComponent` automatique, comme nous l'expliquons ci-dessous.

Avant d'expliquer en détail le fonctionnement de l'appel à `paintComponent`, il est intéressant d'étudier le comportement de la classe `Appel` proposée en exemple. Quand on l'exécute, elle crée une fenêtre vide. De plus, dès l'apparition de cette fenêtre, le programme affiche le texte `paintComponent`. L'affiche se produit à l'emplacement habituellement réservé aux affichages produit par un appel à la méthode `println` (ce qui est parfaitement normal, puisqu'il est produit par la ligne 6 de la classe `Appel` qui utilise justement la méthode `println`), c'est-à-dire en dehors de la fenêtre de graphisme. Le point intéressant est que lors d'un changement de taille de la fenêtre de graphisme, le message `paintComponent` apparaît de nouveau. De même, si on masque la fenêtre graphique puis qu'on la fait réapparaître, le message `paintComponent` est affiché une fois de plus.

En fait, les instructions de la méthode `main` (en particulier la ligne 13) indiquent au programme que pour obtenir le contenu de la fenêtre graphique crée par les instructions précédentes (ligne 9 en particulier), il doit appeler la méthode `paintComponent`. Le principe fondamental du graphisme est qu'à chaque fois qu'une fenêtre graphique doit être dessinée, le programme appelle automatiquement la méthode `paintComponent` adaptée. Quand la taille de la fenêtre change par exemple, la méthode `paintComponent` est appelée. De même quand on fait apparaître une fenêtre préalablement masquée. Ceci explique le comportement du programme `Appel`.

REMARQUE

Le programme `Appel` proposé dans cette section a pour unique but d'illustrer le mécanisme d'appel automatique à `paintComponent`. Il est en général **vivement déconseillé** d'appeler une méthode d'affichage comme `print` ou `println` dans une méthode `paintComponent`.

8.2 Changement de repère

8.2.1 Principe

Le repère utilisé par Java (et en général par tous les langages de programmation) n'est pas un repère usuel, en particulier à cause de son orientation (cf la section 6.2.1). On souhaite souvent décrire des objets mathématiques (par exemple le graphe d'une fonction) dans un repère usuel, mais donner tout même une représentation graphique de ces objets. Pour ce faire, on doit réaliser un changement de repère.

8.2.2 Théorie

On considère un repère orthonormé mathématique usuel (axe des abscisses orienté vers la droite, axe des ordonnées orienté vers le haut). L'écran de l'ordinateur peut représenter une certaine portion rectangulaire du plan mathématique. On décide de travailler sur une zone mathématique décrite par les points $(x_{\min}, y_{\min})$ et $(x_{\max}, y_{\max})$, respectivement coin inférieur gauche et coin supérieur droit du rectangle. Cette zone est représentée par un rectangle de l'écran de largeur `width` et de hauteur `height`. Pour ce faire, on doit convertir les coordonnées depuis le repère mathématique vers le repère de l'écran, en supposant que le rectangle de l'écran est situé en haut à gauche (voir la figure 8.1).

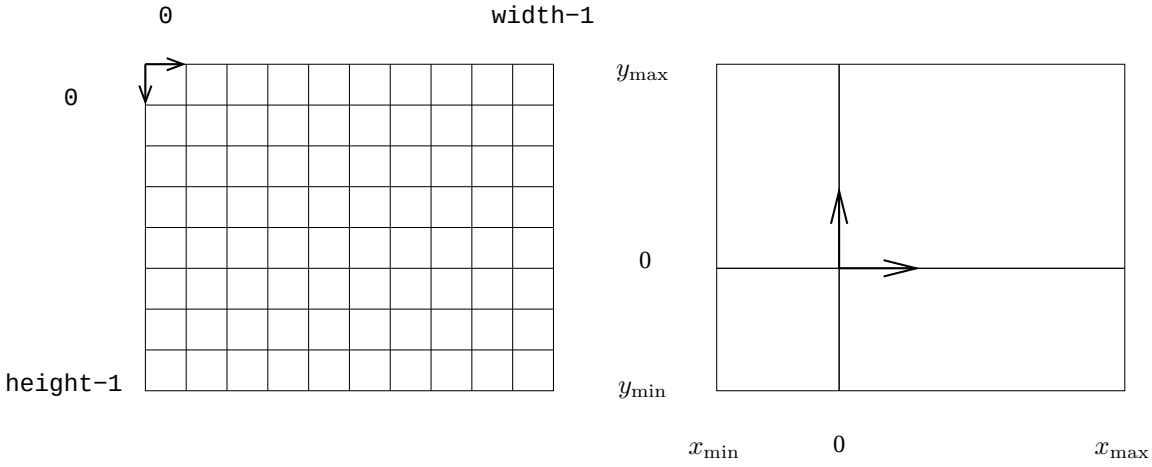

FIG. 8.1 – Repères écran et mathématique

Le changement de repère est une opération affine, qui peut donc se décomposer en une translation suivie d'une opération linéaire (le changement de base qui suit le changement de centre du repère). Cherchons d'abord à identifier la translation.

Le repère écran est centré sur le coin supérieur gauche de la fenêtre. De ce fait, le point de coordonnées écran `(0,0)` est en fait le point de coordonnées mathématiques $(x_{\min}, y_{\max})$ (le coin supérieur gauche de la zone à représenter). La translation de changement de repère doit donc envoyer $(x_{\min}, y_{\max})$ vers `(0,0)`. Il faut donc ajouter $(-x_{\min}, -y_{\max})$ au vecteur coordonnées d'un point exprimé dans le repère de départ.

La transformation linéaire consiste ensuite à passer de la base mathématique vers la base écran. Or, il est clair qu'une abscisse mathématique devient une abscisse écran, sans intervention de son ordonnée mathématique : en d'autres termes, les transformations se font axe par axe.

Considérons donc l'axe des abscisses. Le vecteur qui relie le coin supérieur gauche de la fenêtre à son coin supérieur droit a une coordonnée en x égale à $x_{\max} - x_{\min}$ dans la base mathématique et à

`width-1` dans la base écran. La transformation linéaire de l'axe des abscisses doit donc transformer $x_{\max} - x_{\min}$ en `width-1`. Or, une transformation linéaire de $\mathbb{R}$ dans $\mathbb{R}$ se réduit à une simple multiplication. De ce fait, c'est ici la fonction $f(u) = \frac{\texttt{width-1}}{x_{\max}-x_{\min}}u$.

De la même façon, on montre que la transformation sur l'axe des ordonnées est donnée par la fonction $g(v) = \frac{\texttt{height-1}}{y_{\min}-y_{\max}}v$.

En composant les deux transformations, on obtient donc les coordonnées écran en fonction des coordonnées mathématiques par :

$$\begin{aligned} \texttt{X} &= \frac{\texttt{width-1}}{x_{\max} - x_{\min}}(x - x_{\min}) \\ \texttt{Y} &= \frac{\texttt{height-1}}{y_{\min} - y_{\max}}(y - y_{\max}) \end{aligned}$$

Il est très simple d'inverser les formules pour passer des coordonnées écran aux coordonnées mathématiques :

$$\begin{aligned} x &= x_{\min} + \frac{x_{\max} - x_{\min}}{\texttt{width-1}}\texttt{X} \\ y &= y_{\max} + \frac{y_{\min} - y_{\max}}{\texttt{height-1}}\texttt{Y} \end{aligned}$$

REMARQUE

Dans la pratique, il faut être attentif au fait que les coordonnées écran sont des entiers, ce qui impose une conversion quand on les calcule à partir des coordonnées mathématiques.

8.2.3 Mise en œuvre simple

Dans la pratique, il est intéressant d'appliquer les acquis du chapitre 7 pour mettre en œuvre la technique proposée. En effet, il peut être utile de définir les valeurs $x_{\min}$, $x_{\max}$, etc. par des constantes de classe. De la même façon, il est utile de programmer les calculs de conversion sous forme de méthodes, afin d'éviter d'avoir à les écrire plusieurs fois dans un même programme. On peut ainsi définir une classe de changement de repère, comme l'illustre l'exemple suivant :

Exemple 8.1 :

On propose la classe suivante :

ChangementDeRepere

```
1  public class ChangementDeRepere {
2    public static final double X_MIN=-1.0;
3    public static final double X_MAX=1.0;
4    public static final double Y_MIN=-1.0;
5    public static final double Y_MAX=1.0;
6    public static final int width=200;
7    public static final int height=200;
8    public static int toScreenX(double x) {
9      return (int)Math.round((width-1)*(x-X_MIN)/(X_MAX-X_MIN));
10   }
11   public static int toScreenY(double y) {
12     return (int)Math.round((height-1)*(y-Y_MAX)/(Y_MIN-Y_MAX));
13   }
14   public static double fromScreenX(int X) {
```

```
15      return X_MIN+(X_MAX-X_MIN)*X/(width-1);
16    }
17    public static double fromScreenY(int Y) {
18      return Y_MAX+(Y_MIN-Y_MAX)*Y/(height-1);
19    }
20  }
```

Les méthodes proposées permettent de passer du repère mathématique classique dans le pavé $[-1,1]\times[-1,1]$ vers le repère écran pour un carré de 200 pixels de côté. L'utilisation de diverses constantes permet de modifier simplement le programme pour changer la zone représentée ou le rectangle de l'écran qui lui correspond. L'utilisation de `Math.round` permet d'obtenir un arrondi au pixel le plus proche, alors que `(int)` seulement fait un arrondi au pixel en haut à gauche.

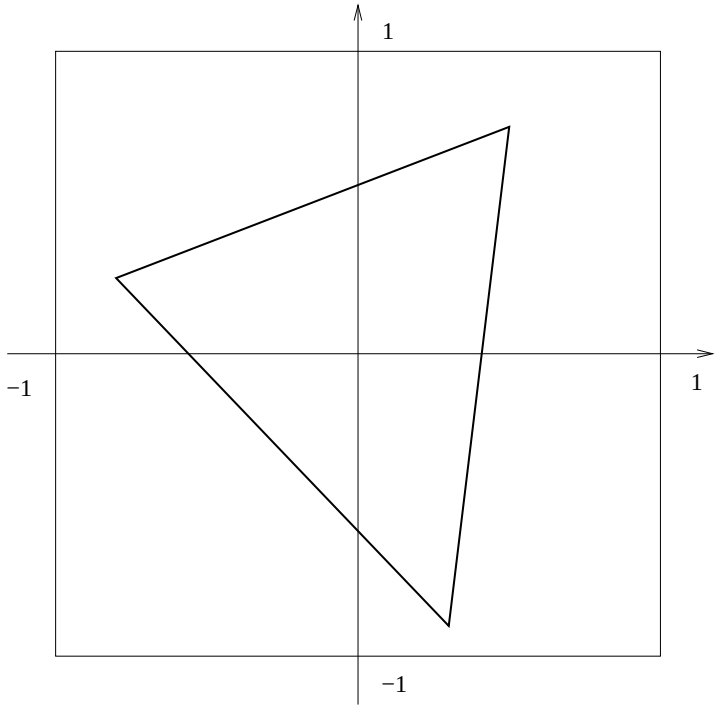

FIG. 8.2 – Un triangle en coordonnées mathématiques

Le programme suivant utilise la classe de changement de repère pour représenter le triangle de sommet $a=(0.5,0.75)$, $b=(-0.8,0.25)$ et $c=(0.3,-0.9)$ (représenté sur la figure 8.2) :

TriangleMath

```
1   import javax.swing.*;
2   import java.awt.*;
3   public class TriangleMath extends JPanel {
4     public void paintComponent(Graphics g) {
5       super.paintComponent(g);
6       int ax=ChangementDeRepere.toScreenX(0.5),
7         ay=ChangementDeRepere.toScreenY(0.75),
8         bx=ChangementDeRepere.toScreenX(-0.8),
9         by=ChangementDeRepere.toScreenY(0.25),
10        cx=ChangementDeRepere.toScreenX(0.3),
```

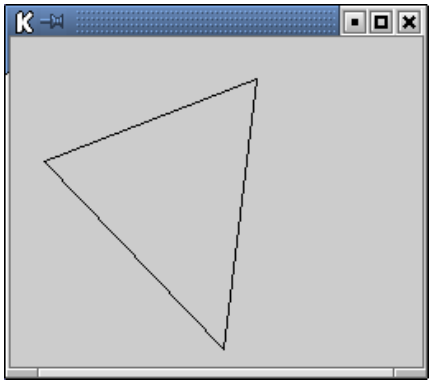

FIG. 8.3 – Affichage produit par le programme `TriangleMath`

```
11        cy=ChangementDeRepere.toScreenY(-0.9);
12      g.drawLine(ax,ay,bx,by);
13      g.drawLine(bx,by,cx,cy);
14      g.drawLine(cx,cy,ax,ay);
15    }
16    public static void main(String[] args) {
17      JFrame fenêtre=new JFrame();
18      fenêtre.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
19      JPanel dessin=new TriangleMath();
20      dessin.setPreferredSize(new Dimension(250,200));
21      fenêtre.getContentPane().add(dessin,BorderLayout.CENTER);
22      fenêtre.pack();
23      fenêtre.setVisible(true);
24    }
25  }
```

Quand on exécute le programme, il produit l'affichage donné par la figure 8.3, ce qui est bien une représentation du triangle de la figure 8.2.

8.2.4 Le cas des longueurs

Quand on travaille avec un repère mathématique, il faut bien veiller à ne pas confondre points et longueurs. En effet, on pourrait être tenté d'appliquer les formules de changement de repère à tous les arguments d'une méthode. Considérons par exemple le rectangle définit par les deux sommets $a = (x, y)$ et $b = (u, v)$ (en coordonnées mathématiques). On peut calculer les coordonnées de ces sommets dans le repère de l'écran, soit $A = (X, Y)$ et $B = (U, V)$. Si on suppose par exemple que $X < U$ et $Y < V$, on trace le rectangle en effectuant l'appel `g.drawRect(X,Y,U-X+1,V-Y+1)`.

Les problèmes commencent quand on souhaite travailler de façon similaire à celle retenue par `Java`, c'est-à-dire en donnant un coin du rectangle associé à sa largeur et sa hauteur. Transformer les coordonnées du coin ne pose pas de problème. Par contre, si on applique le changement de repère à la largeur ou à la hauteur, on obtient un résultat inutilisable. En effet, le changement de repère comprend à la fois une translation et une mise à l'échelle. Or, la translation ne s'applique pas à une longueur. De ce fait, pour obtenir les largeur et hauteur écran d'un rectangle, il faut seulement appliquer le changement d'échelle, comme l'illustre le programme suivant.

Exemple 8.2 :

On commence par ajouter à la classe `ChangementDeRepere` déjà proposée dans l'exemple 8.1 les méthodes suivantes :

ChangementDeRepere

```
1  public static int toScreenWidth(double w) {
2    return (int)Math.round((width-1)*w/(X_MAX-X_MIN));
3  }
4  public static int toScreenHeight(double h) {
5    return (int)Math.round((height-1)*h/(Y_MAX-Y_MIN));
6  }
```

Pour tracer le rectangle de coin supérieur gauche $(-0.7, 0.5)$, de largeur 1.2 et de hauteur 1.3, on propose l'exemple suivant :

RectangleMath

```
1  import javax.swing.*;
2  import java.awt.*;
3  public class RectangleMath extends JPanel {
4    public void paintComponent(Graphics g) {
5      super.paintComponent(g);
6      int x=ChangementDeRepere.toScreenX(-0.7),
7        y=ChangementDeRepere.toScreenY(0.5),
8        w=ChangementDeRepere.toScreenWidth(1.2),
9        h=ChangementDeRepere.toScreenHeight(1.3);
10     g.drawRect(x,y,w,h);
11   }
12   public static void main(String[] args) {
13     JFrame fenêtre=new JFrame();
14     fenêtre.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
15     JPanel dessin=new RectangleMath();
16     dessin.setPreferredSize(new Dimension(250,200));
17     fenêtre.getContentPane().add(dessin,BorderLayout.CENTER);
18     fenêtre.pack();
19     fenêtre.setVisible(true);
20   }
21 }
```

Comme prévu, on obtient l'affichage illustré par la figure 8.4.

8.3 Adaptation dynamique à la taille de la fenêtre

8.3.1 Principe

Dans la pratique, il est parfois utile de tenir compte de la taille effective de la fenêtre au moment du dessin. En effet, l'utilisateur d'un programme a presque toujours la possibilité de changer la taille d'une fenêtre, soit pour qu'elle encombre moins son bureau (s'il utilise plusieurs programmes, et donc plusieurs fenêtres, en même temps), soit au contraire pour mieux étudier ce qu'elle contient.

Dans les programmes que nous avons réalisés jusqu'à présent (au chapitre 6 ainsi que dans le présent chapitre), nous nous sommes contentés de fixer la taille initiale de la fenêtre de dessin (cf la section 6.1.1). Or, il est possible de connaître lors de l'exécution de la méthode `paintComponent` la

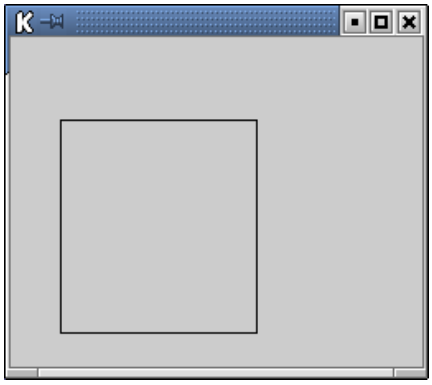

FIG. 8.4 – Affichage produit par le programme `RectangleMath`

taille actuelle de la fenêtre, en utilisant deux méthodes, `getWidth` et `getHeight`. Pour des raisons complexes que nous ne pouvons détailler ici, ces méthodes s'appellent comme des méthodes locales, c'est-à-dire comme si nous les avions définies dans la classe que nous écrivons[1].

Voici la documentation des méthodes :

`int getWidth()`

Renvoie la largeur de la fenêtre au moment de l'appel. Les pixels utilisables ont une abscisse comprise au sens large entre `0` et `getWidth()-1`.

`int getHeight()`

Renvoie la hauteur de la fenêtre au moment de l'appel. Les pixels utilisables ont une ordonnée comprise au sens large entre `0` et `getHeight()-1`.

8.3.2 Un cadre

L'exemple 6.4 du chapitre 6 propose le tracé d'un rectangle avec un bord épais, un cadre. En utilisant les méthodes `getWidth` et `getHeight`, on peut modifier le programme de cet exemple afin d'obtenir un cadre qui s'adapte automatiquement à la taille effective de la fenêtre, comme l'illustre l'exemple suivant :

Exemple 8.3 :

On propose la solution suivante :

CadreAdaptatif

```
1  import javax.swing.*;
2  import java.awt.*;
3  public class CadreAdaptatif extends JPanel {
4    public void paintComponent(Graphics g) {
5      super.paintComponent(g);
6      int w=getWidth();
7      int h=getHeight();
8      int d=20,l=10;
9      g.setColor(new Color(255,255,255));
10     g.fillRect(0,0,w-1,h-1);
11     g.setColor(new Color(255,0,0));
```

[1] Pour simplifier, on peut dire que le compilateur ajoute les méthodes en question à la classe que nous écrivons, ce qui nous permet de faire un appel sans utiliser un nom complet. La situation réelle est un peu plus complexe, mais cela n'empêche pas d'utiliser les méthodes en question.

```
12      g.fillRect(d,d,w-1-2*d,h-1-2*d);
13      g.setColor(new Color(255,255,255));
14      g.fillRect(d+l,d+l,w-1-2*(l+d),h-1-2*(l+d));
15    }
16    public static void main(String[] args) {
17      JFrame fenêtre=new JFrame();
18      fenêtre.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
19      JPanel dessin=new CadreAdaptatif();
20      dessin.setPreferredSize(new Dimension(250,200));
21      fenêtre.getContentPane().add(dessin,BorderLayout.CENTER);
22      fenêtre.pack();
23      fenêtre.setVisible(true);
24    }
25  }
```

Les caractéristiques du tracé sont déterminées essentiellement grâce aux méthodes `getWidth` et `getHeight`. Le code utilise aussi une variable `d` qui indique la distance entre le bord du cadre et le bord de la fenêtre, ainsi qu'une variable `l` qui donne l'épaisseur du cadre.

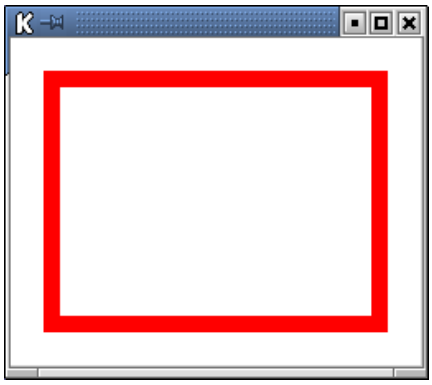

FIG. 8.5 – Affichage initial produit par le programme `CadreAdaptatif`

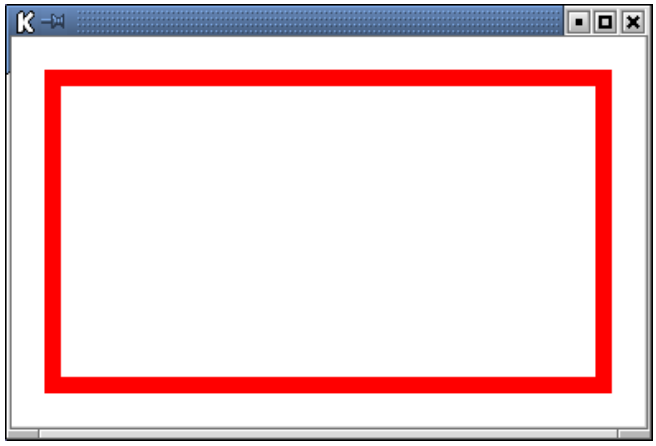

FIG. 8.6 – Affichage produit par le programme `CadreAdaptatif` après changement des dimensions de la fenêtre

Quand on lance le programme, on obtient l'affichage reproduit par la figure 8.5. Si on change la taille de la fenêtre, le cadre s'adapte, c'est-à-dire que lors de l'appel automatique à `paintComponent`, l'utilisation des méthodes `getWidth` et `getHeight` permet d'obtenir des coordonnées de tracé qui produisent le résultat de la figure 8.6.

8.3.3 Changement de repère adaptatif

Il est utile d'appliquer la technique que nous venons de voir dans la section précédente à la représentation d'objets mathématiques. L'idée est de fixer la zone mathématique représentée et d'utiliser toute la place disponible dans la fenêtre graphique. Pour ce faire, il faut **impérativement** que le changement de repère soit programmé dans la classe graphique, en programmant de plus une méthode **d'instance** à la place d'une méthode de classe. Nous expliquerons les points techniques au chapitre 12. A ce niveau du cours, nous pouvons nous contenter d'un modèle qu'il sera facile d'adapter à d'autres cas concrets.

Exemple 8.4 :

Nous reprenons l'exemple 8.1 en représentant le pavé $[-1,1] \times [-1,1]$ grâce à l'intégralité de la fenêtre graphique. Nous obtenons le programme suivant :

TriangleAdaptatif

```
1  import javax.swing.*;
2  import java.awt.*;
3  public class TriangleAdaptatif extends JPanel {
4    public static final double X_MIN=-1.0;
5    public static final double X_MAX=1.0;
6    public static final double Y_MIN=-1.0;
7    public static final double Y_MAX=1.0;
8    public int toScreenX(double x) {
9      return (int)Math.round((getWidth()-1)*(x-X_MIN)/(X_MAX-X_MIN));
10   }
11   public int toScreenY(double y) {
12     return (int)Math.round((getHeight()-1)*(y-Y_MAX)/(Y_MIN-Y_MAX));
13   }
14   public void paintComponent(Graphics g) {
15     super.paintComponent(g);
16     int ax=toScreenX(0.5),ay=toScreenY(0.75),
17       bx=toScreenX(-0.8),by=toScreenY(0.25),
18       cx=toScreenX(0.3),cy=toScreenY(-0.9);
19     g.drawLine(ax,ay,bx,by);
20     g.drawLine(bx,by,cx,cy);
21     g.drawLine(cx,cy,ax,ay);
22   }
23   public static void main(String[] args) {
24     JFrame fenêtre=new JFrame();
25     fenêtre.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
26     JPanel dessin=new TriangleAdaptatif();
27     dessin.setPreferredSize(new Dimension(250,200));
28     fenêtre.getContentPane().add(dessin,BorderLayout.CENTER);
29     fenêtre.pack();
30     fenêtre.setVisible(true);
```

```
31    }
32  }
```

Il est très important de noter deux points :

1. les méthodes de changement de repère doivent impérativement être dans la classe qui définit la méthode `paintComponent` ;
2. comme la méthode `paintComponent`, les méthodes de changement de repère ne comportent pas le mot clé `static` (ce sont des méthodes d'instance).

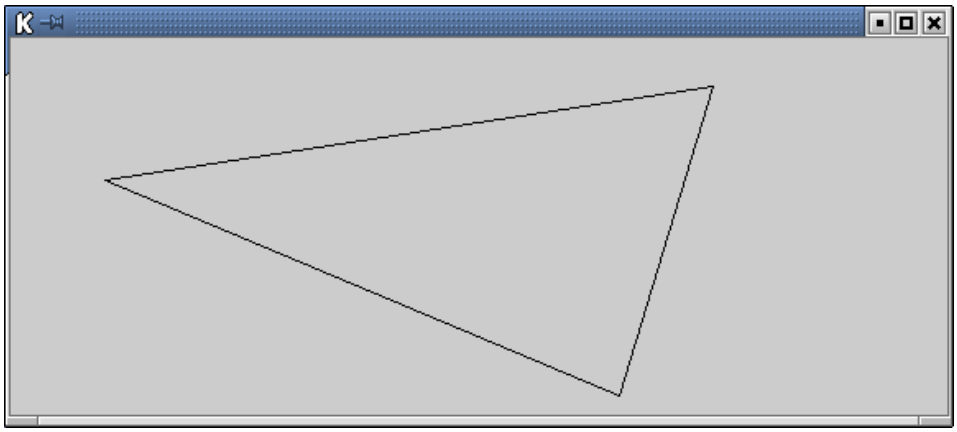

FIG. 8.7 – Affichage produit par le programme `TriangleAdaptatif` après changement des dimensions de la fenêtre

L'affichage produit par le programme à son démarrage est assez semblable à celui obtenu pour le programme d'origine (à savoir `TriangleMath`, dans l'exemple 8.1). Par contre, quand on change le taille de la fenêtre, l'affichage se modifie car la zone mathématique choisie (ici $[-1,1] \times [-1,1]$) remplit toujours toute la fenêtre. On obtient par exemple le résultat de la figure 8.7.

8.4 Représentation du graphe d'une fonction

8.4.1 Le problème

Le graphe d'une fonction f de $[a,b] \subset \mathbb{R}$ dans $\mathbb{R}$ est l'ensemble des points $(x, f(x))$ pour $x \in [a,b]$. Il est bien entendu impossible de représenter exactement cet ensemble qui est infini. La représentation informatique passe donc par une approximation. Pour ce faire, on divise l'intervalle $[a,b]$ en choisissant n points régulièrement espacés (par exemple), avec $x_0 = a$ et $x_n = b$. Ensuite, on trace les segments reliant $(x_i, f(x_i))$ à $(x_{i+1}, f(x_{i+1}))$ pour tout $i < n$. La ligne brisée obtenue est une approximation du graphe de f.

8.4.2 Solution simple

Pour obtenir une solution simple, on fixe la valeur de n. Considérons par exemple le tracé de la fonction $\cos(x^2)$ sur l'intervalle $[0, \pi]$. Voici une classe proposant le tracé de cette fonction :

DessinCos

```
1  import java.awt.*;
2  import javax.swing.*;
```

```
3  public class DessinCos extends JPanel {
4    // la fonction à tracer (ici, cos(x^2))
5    public static double f(double x) {
6      return Math.cos(x*x);
7    }
8    public static final double X_MIN=0;
9    public static final double X_MAX=Math.PI;
10   public static final double Y_MIN=-1.0;
11   public static final double Y_MAX=1.0;
12   public int toScreenX(double x) {
13     return (int)Math.round((getWidth()-1)*(x-X_MIN)/(X_MAX-X_MIN));
14   }
15   public int toScreenY(double y) {
16     return (int)Math.round((getHeight()-1)*(y-Y_MAX)/(Y_MIN-Y_MAX));
17   }
18   public void paintComponent(Graphics g) {
19     super.paintComponent(g);
20     int n=10;
21     double x=X_MIN;
22     double f=f(x);
23     // n points, soit n-1 intervalles
24     double h=(X_MAX-X_MIN)/(n-1);
25     for(int i=1;i<n;i++) {
26       double xi=X_MIN+i*h;
27       double fi=f(xi);
28       g.drawLine(toScreenX(x),toScreenY(f),toScreenX(xi),toScreenY(fi));
29       x=xi;
30       f=fi;
31     }
32   }
33   public static void main(String[] args) {
34     JFrame fenêtre=new JFrame();
35     fenêtre.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
36     JPanel dessin=new DessinCos();
37     dessin.setPreferredSize(new Dimension(250,200));
38     fenêtre.getContentPane().add(dessin,BorderLayout.CENTER);
39     fenêtre.pack();
40     fenêtre.setVisible(true);
41   }
42 }
```

Cette classe est relativement simple, la boucle de la méthode `paintComponent` se chargeant du tracé. Comme dans la section précédente, on utilise une représentation qui s'adapte automatiquement aux dimensions de la fenêtre.

Le résultat graphique obtenu est assez mauvais, comme le montre la figure 8.8. Le problème est simplement que le nombre de points utilisés est trop faible pour donner une représentation correcte. Si on passe à `n=30`, on obtient une représentation plus conforme à la réalité (voir la figure 8.9), mais le tracé devient un peu plus lent (la différence n'est pas notable sur un ordinateur récent car la fonction à calculer est simple).

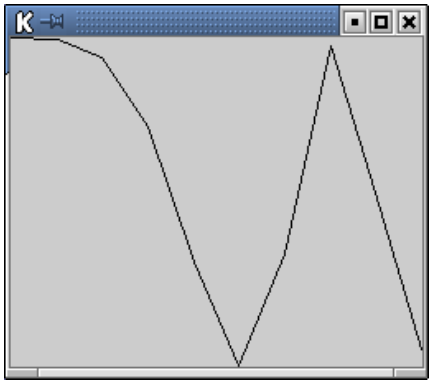

FIG. 8.8 – Affichage produit par `DessinCos` (`n=10`)

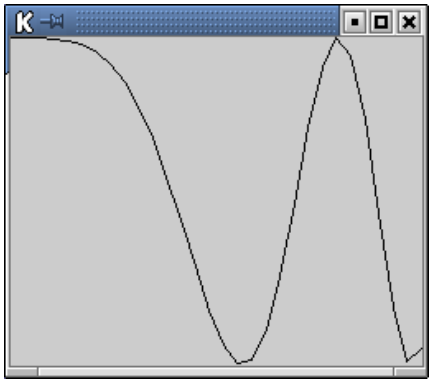

FIG. 8.9 – Affichage produit par `DessinCos` (`n=30`)

8.4.3 Solution optimale

La solution proposée par `DessinCos` est intéressante, mais pas optimale. En effet, on doit faire plusieurs essais avant de déterminer la bonne valeur pour `n`, c'est-à-dire le nombre de segments à utiliser. Or, dans la pratique, il existe un nombre de segments optimal qui dépend des dimensions de la fenêtre. En effet, comme nous l'avons déjà indiqué à la section 6.2.1, l'ordinateur ne travaille pas sur des points sans dimension, mais sur des pixels. De ce fait, quand la fenêtre dans laquelle on dessine comporte par exemple `w` colonnes de pixels, il est inutile de découper l'intervalle $[a, b]$ en plus de `w` segments. Par contre, si on utilise moins de `w` segments, on ne profite pas de toute la précision offerte par l'écran. De ce fait, le nombre de segments optimal est `w` (en fait, il s'agit d'un bon choix, la valeur optimale dépendant de la fonction à représenter).

La méthode la plus simple permettant d'utiliser `w` segments consiste à remplacer les utilisations de `n` par `getWidth()` dans le code de `DessinCos`. On peut aussi remarquer que le passage par les coordonnées mathématiques ne se fait pas de façon très logique dans le code actuel. En effet, il serait plus naturel de travailler directement en coordonnées écran pour les abscisses. Il faut alors transformer une abscisse écran en abscisse mathématique, grâce aux formules de la section 8.2.2. On obtient le code suivant :

DessinCosOptimal

```
1  import java.awt.*;
2  import javax.swing.*;
3  public class DessinCosOptimal extends JPanel {
4    // la fonction à tracer (ici, cos(x^2))
5    public static double f(double x) {
6      return Math.cos(x*x);
7    }
8    public static final double X_MIN=0;
9    public static final double X_MAX=Math.PI;
10   public static final double Y_MIN=-1.0;
11   public static final double Y_MAX=1.0;
12   public double fromScreenX(int X) {
13     return X_MIN+(X_MAX-X_MIN)*X/(getWidth()-1);
14   }
15   public int toScreenY(double y) {
16     return (int)Math.round((getHeight()-1)*(y-Y_MAX)/(Y_MIN-Y_MAX));
17   }
18   public void paintComponent(Graphics g) {
19     super.paintComponent(g);
20     double f=f(X_MIN);
21     for(int i=0;i<getWidth();i++) {
22       double fi=f(fromScreenX(i+1));
23       g.drawLine(i,toScreenY(f),i+1,toScreenY(fi));
24       f=fi;
25     }
26   }
27   public static void main(String[] args) {
28     JFrame fenêtre=new JFrame();
29     fenêtre.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
30     JPanel dessin=new DessinCosOptimal();
31     dessin.setPreferredSize(new Dimension(250,200));
32     fenêtre.getContentPane().add(dessin,BorderLayout.CENTER);
```

```
33      fenêtre.pack();
34      fenêtre.setVisible(true);
35    }
36  }
```

Comme le montre la figure 8.10, le tracé obtenu est beaucoup plus satisfaisant (plus "lisse"). De plus, le tracé reste "parfait" quelles que soient les dimensions effective de la fenêtre, comme l'illustre la figure 8.11.

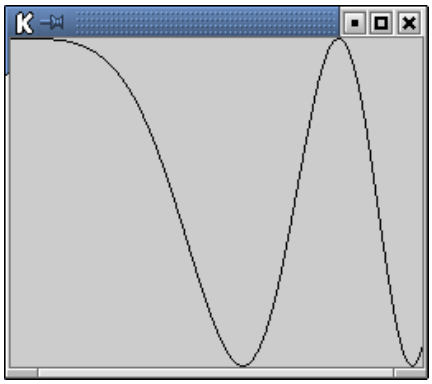

FIG. 8.10 – Affichage produit par `DessinCosOptimal`

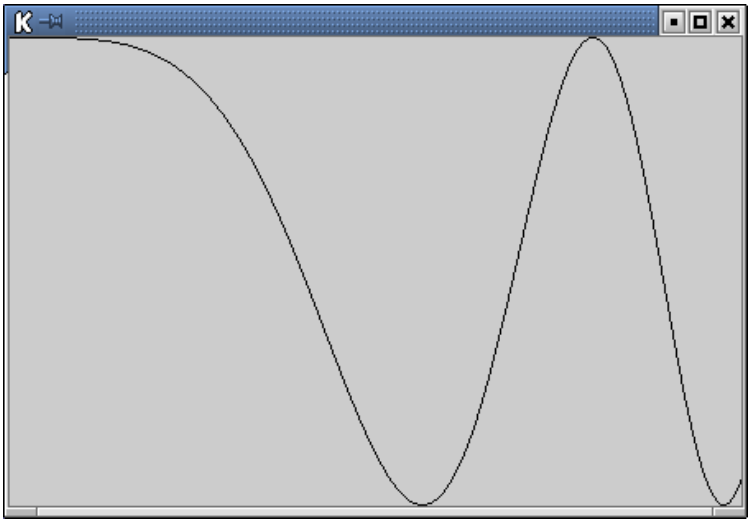

FIG. 8.11 – Affichage produit par `DessinCosOptimal` après changement de taille de la fenêtre

8.4.4 Courbe paramétrique

Une courbe paramétrique est donnée par deux fonctions, $x(t)$ et $y(t)$ qui indiquent la position d'un mobile sur la courbe à l'instant t. Pour tracer une telle courbe, on procède *a priori* comme pour le graphe d'une fonction, c'est-à-dire en divisant l'intervalle dans lequel t varie en un nombre fini de segments et en reliant les points du plan correspondant aux extrémités des segments. Commençons par un exemple de mise en œuvre.

Exemple 8.5 :

On se propose de dessiner la courbe paramétrique donnée par :

$$\begin{cases} x(t) = \sin t \cos t \\ y(t) = \cos 3t \end{cases}$$

Le paramètre t sera élément de l'intervalle $[0, 2\pi]$. La solution proposée est la suivante :

Parametrique

```
1  import java.awt.*;
2  import javax.swing.*;
3  public class Parametrique extends JPanel {
4    public static double x(double t) {
5      return Math.cos(t)*Math.sin(t);
6    }
7    public static double y(double t) {
8      return Math.cos(3*t);
9    }
10   public static final double X_MIN=-0.5;
11   public static final double X_MAX=0.5;
12   public static final double Y_MIN=-1.0;
13   public static final double Y_MAX=1.0;
14   public static final double T_MIN=0;
15   public static final double T_MAX=2*Math.PI;
16   public int toScreenX(double x) {
17     return (int)Math.round((getWidth()-1)*(x-X_MIN)/(X_MAX-X_MIN));
18   }
19   public int toScreenY(double y) {
20     return (int)Math.round((getHeight()-1)*(y-Y_MAX)/(Y_MIN-Y_MAX));
21   }
22   public void paintComponent(Graphics g) {
23     super.paintComponent(g);
24     int n=40;
25     double t=T_MIN;
26     double x=x(t);
27     double y=y(t);
28     // n points, soit n-1 intervalles
29     double h=(T_MAX-T_MIN)/(n-1);
30     for(int i=1;i<n;i++) {
31       double ti=T_MIN+i*h;
32       double xi=x(ti);
33       double yi=y(ti);
34       g.drawLine(toScreenX(x),toScreenY(y),toScreenX(xi),toScreenY(yi));
35       x=xi;
36       y=yi;
37     }
38   }
39   public static void main(String[] args) {
40     JFrame fenêtre=new JFrame();
41     fenêtre.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
42     JPanel dessin=new Parametrique();
```

```
43      dessin.setPreferredSize(new Dimension(250,200));
44      fenêtre.getContentPane().add(dessin,BorderLayout.CENTER);
45      fenêtre.pack();
46      fenêtre.setVisible(true);
47    }
48  }
```

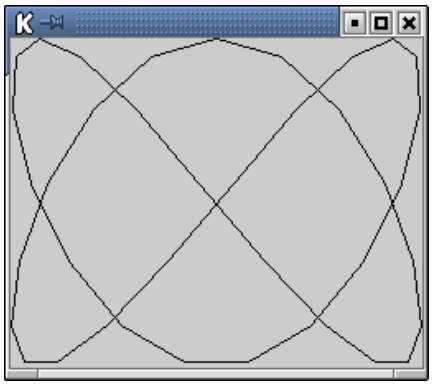

FIG. 8.12 – Affichage produit par `Parametrique`

La solution proposée est très proche de celle retenue pour le tracé du graphe d'une fonction dans la section 8.4.2. Le résultat obtenu est correct, mais assez peu satisfaisant (trop "anguleux"), comme le montre la figure 8.12. On pourrait bien sûr améliorer le résultat en augmentant le nombre de points tracés, au détriment de l'efficacité du programme.

Il est tentant de rechercher une solution semblable à celle proposée dans la section 8.4.3. La situation est cependant un peu plus complexe que pour le graphe d'une fonction. En effet, il n'y a pas de relation simple entre la valeur de h et la distance entre les points $P(t) = (x(t), y(t))$ et $P(t+h) = (x(t+h), y(t+h))$. On pourrait bien sûr utiliser les dérivées des fonctions x et y (à condition qu'elles existent), mais cela change les données du problème : il nous faudrait en effet quatre fonctions plutôt que deux. Nous allons proposer un algorithme simple qui fonctionne correctement en pratique avec deux fonctions.

Dans la méthode de tracé proposée par l'exemple précédent, on fixe une grandeur h et on calcule les points correspondants à t, $t+h$, etc. Le principal problème de cette approche est que h est fixé. Dans la pratique, les points $P(t)$ et $P(t+h)$ peuvent être relativement éloignés, même pour une valeur faible de h, ou au contraire très proches, même pour une valeur élevée de h. Le principe de l'algorithme proposé est simplement d'adapter h dynamiquement pendant le tracé, ce qui reste tout de même délicat techniquement. En fait, à chaque tentative de tracé, on va déterminer h tel que la distance entre $P(t)$ et $P(t+h)$ soit inférieure à un seuil préalablement choisi, comme par exemple 2 pixels. Voici une solution possible :

Exemple 8.6 :

On propose la solution suivante :

ParametriqueAdaptatif

```
1  import java.awt.*;
2  import javax.swing.*;
3  public class ParametriqueAdaptatif extends JPanel {
4    public static double x(double t) {
5      return Math.cos(t)*Math.sin(t);
```

```
6   }
7   public static double y(double t) {
8     return Math.cos(3*t);
9   }
10  public static final double X_MIN=-0.5;
11  public static final double X_MAX=0.5;
12  public static final double Y_MIN=-1.0;
13  public static final double Y_MAX=1.0;
14  public static final double T_MIN=0;
15  public static final double T_MAX=2*Math.PI;
16  public int toScreenX(double x) {
17    return (int)Math.round((getWidth()-1)*(x-X_MIN)/(X_MAX-X_MIN));
18  }
19  public int toScreenY(double y) {
20    return (int)Math.round((getHeight()-1)*(y-Y_MAX)/(Y_MIN-Y_MAX));
21  }
22  public static int distance(int a,int b,int c,int d) {
23    return Math.max(Math.abs(a-c),Math.abs(b-d));
24  }
25  public void paintComponent(Graphics g) {
26    super.paintComponent(g);
27    int n=40;
28    double t=T_MIN;
29    int x=toScreenX(x(t));
30    int y=toScreenY(y(t));
31    // valeur de départ pour h
32    double h=(T_MAX-T_MIN)/(n-1);
33    int dmax=2;
34    while(t<T_MAX) {
35      double h1=h;
36      int x1=toScreenX(x(t+h));
37      int y1=toScreenY(y(t+h));
38      int d1=distance(x,y,x1,y1);
39      while (d1==0) {
40        h=h1;
41        h1=h1*2;
42        x1=toScreenX(x(t+h1));
43        y1=toScreenY(y(t+h1));
44        d1=distance(x,y,x1,y1);
45      }
46      if(h==h1) {
47        h=0;
48      }
49      while(d1>dmax ) {
50        double m=(h+h1)/2;
51        int x2=toScreenX(x(t+m));
52        int y2=toScreenY(y(t+m));
53        int d2=distance(x,y,x2,y2);
54        if(d2==0) {
```

```
55            h=m;
56          } else {
57            d1=d2;
58            x1=x2;
59            y1=y2;
60            h1=m;
61          }
62        }
63        g.drawLine(x,y,x1,y1);
64        x=x1;
65        y=y1;
66        t=t+h1;
67        h=h1;
68      }
69    }
70    public static void main(String[] args) {
71      JFrame fenêtre=new JFrame();
72      fenêtre.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
73      JPanel dessin=new ParametriqueAdaptatif();
74      dessin.setPreferredSize(new Dimension(250,200));
75      fenêtre.getContentPane().add(dessin,BorderLayout.CENTER);
76      fenêtre.pack();
77      fenêtre.setVisible(true);
78    }
79  }
```

Certains points méritent quelques explications :

- comme son nom l'indique, la méthode `distance` calcule la distance entre deux points de l'écran. La distance choisie compte le nombre de pixels qui séparent les deux points dans la direction de plus grand éloignement (horizontalement ou verticalement) ;
- la partie complexe se situe entre les lignes 35 et 62. Ces lignes cherchent une valeur de h telle que $P(t)$ et $P(t+h)$ soient proches au sens de la distance en pixel (c'est pourquoi on travaille avec les coordonnées écran). L'idée de l'algorithme est basé sur la dichotomie :

  1. Dans un premier temps (boucle des lignes 39 à 45), on cherche une valeur h_1 telle que la distance entre $P(t)$ et $P(t+h_1)$ soit supérieure ou égal à un pixel. En effet, comme les pixels ont une surface, deux points mathématiques distincts peuvent correspondre au même pixel.

  2. Dans un deuxième temps (boucle des lignes 49 à 62), on applique l'algorithme inspiré de la dichotomie : on sait que $P(t+h)$ est identique (au sens des pixels) à $P(t)$ et que $P(t+h)$ et $P(t+h_1)$ sont distants d'au moins un pixel. On tente de réduire la distance entre $P(t+h)$ et $P(t+h_1)$ au minimum acceptable en essayant le point $P(t+m)$, avec $m = \frac{h+h_1}{2}$. Si ce point est trop près ou trop loin de $P(t+h)$, on remplace h ou h_1 par m (comme dans la dichotomie) et on recommence.

Le résultat produit par le programme est illustré par la figure 8.13 et est très satisfaisant. De plus, quand on change la taille de la fenêtre, le dessin s'adapte, tout en restant précis et sans point "anguleux" (voir la figure 8.14).

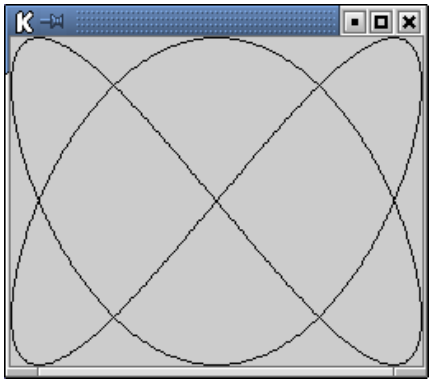

FIG. 8.13 – Affichage produit par `ParametriqueAdaptatif`

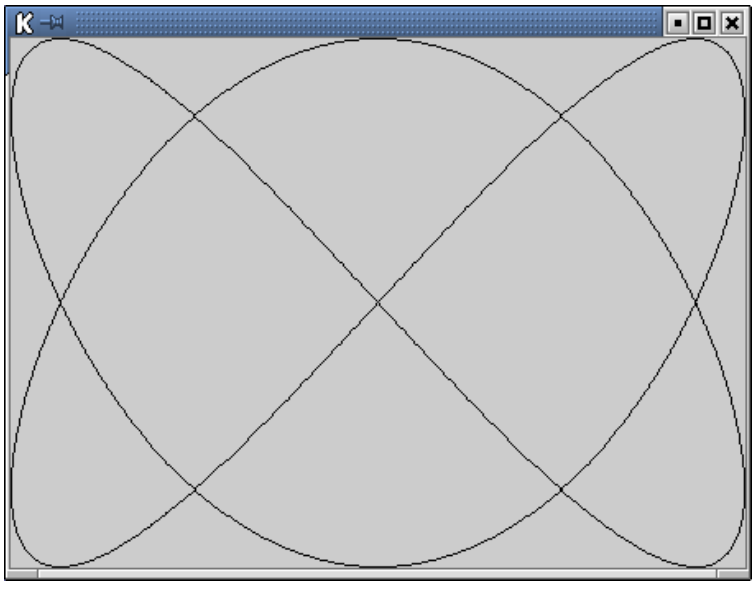

FIG. 8.14 – Affichage produit par `ParametriqueAdaptatif` après changement de taille de la fenêtre

8.5 Conclusion

Nous avons vu dans ce chapitre comment utiliser les méthodes pour produire des résultats graphiques difficiles à obtenir autrement. Certains programmes proposés utilisent par exemple trois fois chaque méthode de changement de repère. Sans méthode, il aurait fallu recopier trois fois le code en question, multipliant ainsi les risques d'erreur. De plus, nous avons appris à utiliser de nouvelles méthodes graphiques qui permettent de connaître la taille effectivement de la fenêtre au moment de l'exécution de la méthode `paintComponent`.

Dans le présent chapitre, et comme pour le chapitre 6, notre but est simplement d'illustrer les constructions `Java` et algorithmiques étudiées jusqu'à présent avec des exemples plus agréables et ludiques que le calcul de suites ou la résolution de système d'équations. En ce sens, nous conseillons vivement au lecteur de mettre en œuvre les algorithmes et les solutions proposés, pour acquérir une connaissance pratique réelle des méthodes et des concepts abordés antérieurement.

CHAPITRE 9

Première approche des objets

Sommaire

Introduction

Dans les chapitres précédents, nous avons appris à utiliser les types fondamentaux en tant qu'outils de représentation des informations à faire manipuler par un programme. Nous connaissons les outils algorithmiques nécessaires à l'écriture de programmes évolués (les sélections et les boucles), et nous savons grâce aux méthodes et aux classes comment organiser un programme complexe. Il nous reste maintenant à améliorer la représentation de l'information dans un programme pour dépasser les applications simplistes qui manipulent quelques nombres entiers ou réels.

Comment faire par exemple pour manipuler un texte ? Comment stocker dans la mémoire de l'ordinateur les notes d'un élève (par exemple) si le nombre de notes peut être saisi par l'utilisateur *au moment de l'exécution du programme* ? De façon plus générale, comment obtenir une représentation simple des informations à faire traiter par un programme ?

Le but de ce chapitre est d'introduire la notion **d'objet** qui généralise la notion de valeur et qui permet la représentation simple et efficace d'informations complexes. Nous commencerons par étudier le **type objet** `String` qui permet la manipulation de chaînes de caractères, c'est-à-dire de suite quelconque de caractères. Cela nous permettra d'avoir un premier aperçu de la notion d'objet.

Nous introduirons ensuite les **méthodes d'instance** qui permettent de manipuler les objets et jouent en fait le rôle joué par les opérateurs pour les types fondamentaux. Nous étudierons ensuite les problèmes que posent la gestion mémoire des objets en parlant de la **manipulation par référence** et de ces conséquences complexes.

Motivés par un souci d'efficacité, nous présenterons un exemple d'objets modifiables, les `StringBuffer`s qui proposent une autre représentation des chaînes de caractères. Nous utiliserons cet exemple important pour illustrer les conséquences les plus subtiles de la manipulation par référence à savoir les **effets de bord**.

9.1 Les chaînes de caractères : le type `String`

9.1.1 Le type `String` et les valeurs littérales associées

Principe de base

Commençons par une définition :

Définition 9.1 *Une **chaîne de caractères** est une suite finie de caractères.*

Nous avons vu qu'il existe un type fondamental `char` qui permet de manipuler **un** caractère. Le problème est donc de manipuler **plusieurs** caractères en même temps.

Un des moyens pour manipuler une chaîne de caractères en `Java` est de passer par le type `String` (nous verrons qu'on peut aussi utiliser le type `StringBuffer`, cf la section 9.4). Ce nouveau type peut s'utiliser exactement comme n'importe quel autre type (pour l'instant, nous ne connaissons que les types fondamentaux, présentés à la section 2.1.2) :

- on peut déclarer des variables de type `String` ;
- une méthode peut demander un ou plusieurs paramètres de type `String` (voir l'exemple 9.5) ;
- une méthode peut déclarer `String` comme type pour son résultat (voir les exemples 9.21 et 9.26).

Valeur littérale

Nous avons déjà rencontré des *valeurs littérales* de type `String`. En effet, quand on veut afficher un texte à l'écran, on écrit `System.out.println("Bonjour")`, ce qui provoque l'affichage du texte compris entre les guillemets (ici, `Bonjour`). Ce texte est une chaîne de caractères et le compilateur `Java` lui associe le type `String`. Les valeurs littérales de type `String` sont donc les suites de caractères quelconques, comprises entre des guillemets.

Exemple 9.1 :

Chaque ligne suivante comporte une valeur littérale de type `String` :

```
"Bonjour"
"x+2"
"static void main("
"Voilà un texte..."
""
```

Notons que la dernière valeur littérale désigne la chaîne de caractères vide, c'est-à-dire ne contenant aucun caractère. Cette construction n'est pas possible avec le type `char` pour lequel les valeurs littérales doivent toujours désigner exactement un caractère. La construction ■ est de ce fait refusée par le compilateur.

Les caractères spéciaux

On peut éventuellement vouloir insérer un guillemet dans une chaîne de caractères. Si on écrit `"exemple " "`, on obtient pas une chaîne de caractères. Pour pouvoir mettre un guillemet dans un chaîne de caractères, il faut le faire précéder d'un *antislash*[1], c'est à dire l'écrire `\"`. Notre exemple précédent s'écrit donc `"exemple \" "`. Il faut être attentif à bien terminer les chaînes de caractères et à ne pas mettre de guillemet directement dans une chaîne car le compilateur a des difficultés à comprendre un programme qui comporte des erreurs liées aux chaînes, comme l'illustre l'exemple suivant :

[1] un *backslash* en anglais.

Exemple 9.2 :

On considère le programme suivant :

MauvaiseValeur

```
1 public class MauvaiseValeur {
2   public static void main(String[] args) {
3     String s="ABCD " EFGH ";
4   }
5 }
```

Le compilateur refuse le programme et donne des messages d'erreur qui montrent que la présence du guillemet sans antislash perturbe son analyse du programme :

Erreur de compilation

```
MauvaiseValeur.java:3: ';' expected
    String s="ABCD " EFGH ";
                     ^
MauvaiseValeur.java:3: unclosed string literal
    String s="ABCD " EFGH ";
                          ^
2 errors
```

Notons d'autre part que certains autres "caractères" ne peuvent pas être insérés directement dans une valeur littérale de type `String`. On ne peut pas par exemple passer à la ligne à l'intérieur d'une chaîne, comme l'indique l'exemple suivant :

Exemple 9.3 :

On considère le programme suivant :

PasDeSautDeLigne

```
1 public class PasDeSautDeLigne {
2   public static void main(String[] args) {
3     String tentative="un passage à
4 la ligne";
5     System.out.println(tentative);
6   }
7 }
```

Le programme est refusé par le compilateur qui donne le message d'erreur suivant :

Erreur de compilation

```
PasDeSautDeLigne.java:3: unclosed string literal
    String tentative="un passage à
                     ^
PasDeSautDeLigne.java:4: unclosed string literal
la ligne";
       ^
2 errors
```

La tabulation (qui permet d'aligner proprement des textes) n'est pas non plus directement utilisable en `Java`. Pour obtenir ces "caractères" spéciaux, on utilise de nouveau le symbole *antislash*, selon la correspondance dans le tableau qui suit.

Séquence	“caractère” correspondant
\n	passage à la ligne
\t	tabulation
\\	le caractère \ lui-même
\’	le caractère ’ (type char)

Voici un exemple simple d’application :

Exemple 9.4 :

Nous allons écrire un programme qui affiche le tableau proposé, en le plaçant d’abord dans une `String`. Pour que le programme soit lisible, on utilise par anticipation l’opérateur `+` (cf la section 9.1.2), qui colle bout à bout deux `String`s :

Echappement

```
1  public class Echappement {
2    public static void main(String[] args) {
3      String tableau="Séquence\t‘‘caractère’’ correspondant\n";
4      tableau+="\\n\t\tpassage à la ligne\n";
5      tableau+="\\t\t\ttabulation\n";
6      tableau+="\\\\\t\tle caractère \\ lui-même\n";
7      tableau+="\\’\t\tle caractère ’ (type char)";
8      System.out.println(tableau);
9      char c='\’';
10     System.out.println(c);
11   }
12 }
```

Les deux dernières lignes du programme réalisent une démonstration de la séquence `\’`, inutile pour les `String`s, mais indispensable pour le type fondamental `char`. L’affichage produit par le programme est le suivant :

Affichage

```
Séquence        ‘‘caractère’’ correspondant
\n              passage à la ligne
\t              tabulation
\\              le caractère \ lui-même
\’              le caractère ’ (type char)
```

L’utilisation de la tabulation permet un alignement correct des colonnes. Chaque colonne peut contenir au plus huit caractères, d’où l’utilisation de deux tabulations dans l’exemple proposé.

REMARQUE

Notons au passage que chaque caractère spécial est bien un unique caractère. La séquence `\n` comporte deux caractères dans un programme, mais une chaîne construite à partir de celle-ci ne comportera qu’un seul caractère. Par exemple, la chaîne `"\n\n"` est de longueur 2 et correspond à deux passages à la ligne.

Applications

Voici un exemple simple d'application des `String`s :

Exemple 9.5 :

Ce programme calcule la moyenne de deux notes :

Moyenne

```
1  import dauphine.util.*;
2  public class Moyenne {
3    public static int lireNote(String message) {
4      int result;
5      System.out.print(message);
6      result=Console.readInt();
7      while (result<0||result>20) {
8        System.out.println("La valeur doit être comprise entre 0 et 20");
9        System.out.print(message);
10       result=Console.readInt();
11     }
12     return result;
13   }
14   public static void main(String[] args) {
15     Console.start();
16     int noteDeMath,noteDEco;
17     noteDeMath=lireNote("Note de math=");
18     noteDEco=lireNote("Note d'économie=");
19     System.out.println("Moyenne= "+((double)noteDeMath+noteDEco)/2);
20   }
21 }
```

On remarque que la méthode `lireNote` prend pour paramètre une chaîne de caractères, qui est ensuite transmise à `System.out.print` pour être affichée. Ceci permet d'utiliser la méthode `lireNote` pour lire diverses notes en indiquant comme paramètre effectif le message à afficher pour demander la note. Une utilisation de ce programme provoque typiquement ce genre d'affichage :

Affichage

```
Note de math=34
La valeur doit être comprise entre 0 et 20
Note de math=15
Note d'économie=-5
La valeur doit être comprise entre 0 et 20
Note d'économie=13
Moyenne= 14.0
```

L'utilisation d'un paramètre de type `String` permet donc de proposer une méthode de lecture plus pratique à utiliser.

Nous pouvons aussi reprendre l'exemple de la classe `Couleur` proposée dans la section 7.7.3 :

Exemple 9.6 :

Nous avons vu comment utiliser des constantes de classe pour représenter simplement les couleurs des cartes à jouer. Nous pouvons maintenant compléter la classe `Couleur` afin de permettre la transformation d'une valeur numérique en une `String`, par exemple en vue d'un affichage (comme dans la classe `Carte` que nous avions proposer avec la classe `Couleur` dans la section 7.7.3. Voici la nouvelle version de `Couleur` :

Couleur

```
1  public class Couleur {
2    public static final int CARREAU=0;
3    public static final int COEUR=1;
4    public static final int PIQUE=2;
5    public static final int TREFLE=3;
6    public static String toString(int couleur) {
7      switch(couleur) {
8      case CARREAU:
9        return "Carreau";
10     case COEUR:
11       return "Coeur";
12     case PIQUE:
13       return "Pique";
14     case TREFLE:
15       return "Trèfle";
16     }
17     return "Erreur";
18   }
19 }
```

Voici un exemple très basique d'utilisation de la classe :

DemoCouleur

```
1  public class DemoCouleur {
2    public static void main(String[] args) {
3      int carte=Couleur.CARREAU;
4      System.out.println(Couleur.toString(carte));
5    }
6  }
```

En conjonction avec les possibilités des `String`, la méthode `toString` de la classe `Couleur` sera beaucoup plus intéressante en pratique que la méthode `écrireCouleur` de la classe `Carte` proposée à la section 7.7.3. Nous aurons en effet avec la nouvelle méthode strictement plus de possibilités pratiques.

9.1.2 Concaténation

Dans le chapitre 2, nous avons appris à utiliser les différents types fondamentaux en étudiant les opérateurs applicables aux valeurs de ces différents types. Le seul opérateur utilisable avec les `String`s est le symbole de l'addition, qui réalise une opération de concaténation. On peut écrire par exemple `String s="Bon"+"jour" ;`. La variable `s` contiendra alors la chaîne de caractères `"Bonjour"`. Dans cette concaténation, on peut bien sûr faire apparaître des variables de type `String`.

On peut de plus ajouter plusieurs chaînes les unes aux autres dans une même expression, comme par exemple `String s="Bonjour"+" à "+"tous";`.

L'intérêt principal de la concaténation est de simplifier l'utilisation des méthodes d'affichage, comme nous l'avons déjà vu à la section 3.5.3. En fait, nous manipulons depuis ce chapitre des `String`s, mais sans le savoir !

De plus, comme nous l'avons justement vu lors de notre apprentissage de l'affichage, il est en fait possible de concaténer une valeur d'un type fondamental quelconque à une `String`. Le processeur va d'abord convertir la valeur en une `String` (cf la section 9.1.4) puis concaténer les deux chaînes. La chaîne résultat de `12+"ABC"` est donc `"12ABC"`.

La concaténation nous permet d'améliorer le programme de l'exemple 9.5 :

Exemple 9.7 :

On propose le programme suivant :

MoyenneConcat

```
1  import dauphine.util.*;
2  public class MoyenneConcat {
3    public static int lireValeur(String message,int min,int max) {
4      int result;
5      System.out.print(message);
6      result=Console.readInt();
7      while (result<min||result>max) {
8        System.out.println("La valeur doit être comprise entre "+min
9                           +" et "+max);
10       System.out.print(message);
11       result=Console.readInt();
12     }
13     return result;
14   }
15   public static void main(String[] args) {
16     Console.start();
17     int noteDeMath,noteDEco;
18     noteDeMath=lireValeur("Note de math=",0,20);
19     noteDEco=lireValeur("Note d'économie=",0,20);
20     System.out.println("Moyenne= "+((double)noteDeMath+noteDEco)/2);
21   }
22 }
```

Dans cette nouvelle version, la méthode `lireValeur` est très générale : elle correspond à une saisie dans laquelle on impose un intervalle pour la valeur entière proposée par l'utilisateur.

REMARQUE

L'opération de concaténation permet de construire progressivement, par ajout successif, un résultat complexe, en particulier au moyen d'une boucle. Nous illustrerons ce type d'utilisation dans l'exemple 9.21.

9.1.3 Saisie

On peut demander à l'utilisateur de saisir une chaîne de caractères sous la forme d'une `String`, en utilisant la méthode `readString` de la classe `Console`.

Voici un exemple très élémentaire d'utilisation de cette méthode :

Exemple 9.8 :

Un programme qui s'essaye à la politesse :

Bonjour

```
1  import dauphine.util.*;
2  public class Bonjour {
3    public static void main(String[] args) {
4      Console.start();
5      System.out.print("Donnez votre nom : ");
6      String nom=Console.readString();
7      System.out.println("Bonjour, "+nom);
8    }
9  }
```

Ce programme donnera par exemple l'affichage suivant :

```
Donnez votre nom : Toto
Bonjour, Toto
```

9.1.4 Conversion

D'un type quelconque vers une `String`

On a vu dans les exemples précédents, et, plus généralement, depuis le début du présent ouvrage, que l'ordinateur est capable de transformer automatiquement une valeur numérique en une chaîne de caractères lors d'une concaténation. De façon générale, on peut se demander comment transformer une valeur d'un type fondamental en une `String` de façon directe, sans passer par une concaténation.

On remarque tout d'abord qu'une approche naïve n'est pas valide, comme le montre l'exemple suivant :

Exemple 9.9 :

Considérons le programme suivant :

MauvaisType

```
1  public class MauvaisType {
2    public static void main(String[] args) {
3      double x=2;
4      String s=x;
5      String t=2;
6    }
7  }
```

Le compilateur refuse le programme et affiche le messages d'erreur suivants :

Erreur de compilation

```
MauvaisType.java:4: incompatible types
found   : double
required: java.lang.String
    String s=x;
             ^
MauvaisType.java:5: incompatible types
found   : int
```

```
required: java.lang.String
    String t=2;
             ^
2 errors
```

On voit donc que le compilateur refuse de convertir un entier et un réel en une `String`.

REMARQUE

Le type `String` se comporte donc comme les autres types, c'est-à-dire que seules certaines affectations sont considérées comme valides, celles qui respectent les types. Dans le cas de `String`, on peut seulement placer dans une variable de type `String` des éléments du même type.

Comme la solution naïve n'est pas utilisable, on utilise des méthodes de conversion. Il faut savoir que le type `String` est associé à une classe du même nom. Cette classe propose des méthodes de classe `valueOf` qui transforment leur paramètre en une chaîne de caractères (de type `String`).

Exemple 9.10 :

Voici comment le programme de l'exemple 9.9 peut être modifié pour devenir correct :

TypeCorrect

```
1  public class TypeCorrect {
2    public static void main(String[] args) {
3      double x=2;
4      String s=String.valueOf(x);
5      String t=String.valueOf(2);
6    }
7  }
```

Comme par exemple pour certaines méthodes de la classe `Math` (voir la section 3.3.2), il existe en fait une version de `valueOf` pour chaque type fondamental, ce qui permet de faire toutes les conversions souhaitées.

Contrairement à ce qu'on pourrait croire, le compilateur fait une différence entre un caractère seul et une chaîne de caractères (représentée par une `String`). Il est obligatoire de passer par la méthode `valueOf` adaptée pour pouvoir transformer un `char` en une `String`, comme l'illustre l'exemple suivant :

Exemple 9.11 :

On considère le programme suivant :

CharEtStringFaux

```
1  public class CharEtStringFaux {
2    public static void main(String[] args) {
3      char c='a';
4      String s="b";
5      s=c;
6      c=s;
7      s='u';
8      c="v";
9    }
10 }
```

Le compilateur refuse le programme et donne le message d'erreur suivant :

Erreur de compilation

```
CharEtStringFaux.java:5: incompatible types
found   : char
required: java.lang.String
    s=c;
      ^
CharEtStringFaux.java:6: incompatible types
found   : java.lang.String
required: char
    c=s;
      ^
CharEtStringFaux.java:7: incompatible types
found   : char
required: java.lang.String
    s='u';
      ^
CharEtStringFaux.java:8: incompatible types
found   : java.lang.String
required: char
    c="v";
      ^
4 errors
```

Les lignes 3 et 4 sont naturellement acceptées. Ce n'est pas le cas pour les lignes suivantes, car on tente soit de placer une `String` dans une variable de type `char` (lignes 6 et 8), soit le contraire (lignes 5 et 7). Grâce à la méthode `valueOf` de la classe `String`, il est possible de corriger les lignes 5 et 7, qu'on peut remplacer par :

```
s=String.valueOf(c);
s=String.valueOf('u');
```

Pour les lignes 6 et 8, il faut passer par une *méthode d'instance* qu'on étudiera à la section 9.2.3.

REMARQUE

Il est impossible de réaliser une conversion par une "mise entre guillemets" aux effets magiques. Supposons qu'on souhaite par exemple convertir en `String` le contenu de la variable `x` (de type `double` par exemple). Certains programmeurs débutants pensent qu'il suffit d'écrire `"x"` et que, *de façon exceptionnelle*, le compilateur comprendra qu'il faut en fait remplacer cette valeur littérale de type `String` par `String.valueOf(x)`. Cette supposition est incorrecte : **le compilateur n'interprète jamais le contenu d'une valeur littérale de type `String`**. La chaîne `"x"` reste immuablement la chaîne `"x"`, même s'il existe une variable appelée `x`.

D'une `String` vers un type fondamental

Que penser de la chaîne de caractères `"1234"` ? On peut se demander si le processeur est capable de comprendre que cette chaîne correspond à un entier. La réponse est à la fois négative et positive. Du côté négatif, comme nous venons de le rappeler dans la remarque précédente, le compilateur n'interprète jamais le contenu d'une chaîne de caractères, ce qui est illustré par l'exemple suivant :

Exemple 9.12 :

Le programme suivant tente naïvement de placer une chaîne de caractères contenant l'écriture décimale d'un entier dans une variable de type `int` :

StringNoInt

```
1 public class StringNoInt {
2   public static void main(String[] args) {
3     String s="1234";
4     int x=s;
5   }
6 }
```

Le compilateur refuse ce programme et donne le message d'erreur suivant :

Erreur de compilation

```
StringNoInt.java:4: incompatible types
found   : java.lang.String
required: int
    int x=s;
          ^
1 error
```

Par contre, il existe des méthodes de conversion. En d'autres termes, on peut dans un programme demander au processeur d'analyser le contenu d'une `String` afin d'en tirer éventuellement une valeur numérique.

Chaque type fondamental est associé à une classe portant presque le même. Nous avons d'ailleurs étudié certains éléments de ces classes au chapitre 3 (par exemple aux sections 3.4.3 et 3.6.3). Chacune de ces classes définie une méthode `parseXxx` dont le principe est de transformer la `String` paramètre en une valeur du type associé à la classe, quand cela est possible. En cas de problème, la méthode provoque l'arrêt du programme. Voici la liste des méthodes utilisables :

- la classe `Byte` définit la méthode :

  `byte parseByte(String s)`

  Renvoie le contenu de la chaîne `s` sous forme d'un `byte`, si cela est possible. Produit une erreur sinon.

- la classe `Double` définit la méthode :

  `double parseDouble(String s)`

  Renvoie le contenu de la chaîne `s` sous forme d'un `double`, si cela est possible. Produit une erreur sinon.

- la classe `Float` définit la méthode :

  `float parseFloat(String s)`

  Renvoie le contenu de la chaîne `s` sous forme d'un `float`, si cela est possible. Produit une erreur sinon.

- la classe `Integer` définit la méthode :

  `int parseInt(String s)`

  Renvoie le contenu de la chaîne `s` sous forme d'un `int`, si cela est possible. Produit une erreur sinon.

- la classe `Long` définit la méthode :

`long parseLong(String s)`

Renvoie le contenu de la chaîne `s` sous forme d'un `long`, si cela est possible. Produit une erreur sinon.

– la classe `Short` définit la méthode :

`short parseShort(String s)`

Renvoie le contenu de la chaîne `s` sous forme d'un `short`, si cela est possible. Produit une erreur sinon.

Voici un exemple d'utilisation de certaines des méthodes :

Exemple 9.13 :

DemoParse

```
1  public class DemoParse {
2    public static void main(String[] args) {
3      String s="1234";
4      int x=Integer.parseInt(s);
5      System.out.println(x);
6      System.out.println(2*x);
7      s=String.valueOf(x/3.0);
8      double y=Double.parseDouble(s);
9      float z=Float.parseFloat(s);
10     System.out.println(y);
11     System.out.println(z);
12     System.out.println(s);
13   }
14 }
```

Ce programme produit l'affichage suivant :

AFFICHAGE

```
1234
2468
411.3333333333333
411.33334
411.3333333333333
```

Voici maintenant un exemple d'utilisation problématique :

Exemple 9.14 :

On considère le programme suivant :

DemoParseErreur

```
1  import dauphine.util.*;
2  public class DemoParseErreur {
3    public static void main(String[] args) {
4      Console.start();
5      String s=Console.readString();
6      int x=Integer.parseInt(s);
7      System.out.println(2*x);
8    }
9  }
```

Si l'utilisateur propose valeur entière, le programme fonctionne parfaitement et donne par exemple l'affichage suivant :

AFFICHAGE

```
56765
113530
```

Par contre, si l'utilisateur ne donne pas un entier, le programme plante et donne le message suivant :

ERREUR D'EXÉCUTION

```
2.5
Exception in thread "main" java.lang.NumberFormatException: 2.5
        at java.lang.Integer.parseInt(Integer.java:414)
        at java.lang.Integer.parseInt(Integer.java:454)
        at DemoParseErreur.main(DemoParseErreur.java:6)
```

Il est donc souhaitable d'utiliser les méthodes `readXxxx` adaptées plutôt que de tenter une conversion[2].

9.1.5 La notion d'objet

Quand on manipule les types fondamentaux, on travaille avec des valeurs. Tous les autres types sont des **types objet**. On ne parle plus alors de valeur du type considéré, mais d'**objet**. Par exemple, la chaîne de caractères `"exemple"` est un **objet** de type `String`.

Les objets sont une généralisation de la notion de valeur :

- chaque objet possède un type qui est nécessairement associé à une classe (voir la section 9.2.4), comme par exemple le type `String` qui correspond à une classe de même nom. On dit qu'**un objet est instance de la classe** correspondant à son type. On parlera donc d'objet instance de `String`. On dit aussi qu'un tel objet est de classe `String` ;
- pour chaque type objet, il existe un moyen de "fabriquer" un objet qui correspond aux valeurs littérales pour les types fondamentaux. Pour les `String`s, on dispose de valeurs littérales, mais c'est un cas exceptionnel. Pour les autres types objet, on utilisera un **constructeur** (cf la section 9.4.3) ;
- pour chaque type objet, il existe l'équivalent des opérateurs numériques et logiques : ce sont les **méthodes d'instance** (cf la section 9.2). Ces méthodes permettent de réaliser des opérations sur les objets, d'une façon analogue aux combinaisons de valeurs réalisées par les opérateurs pour les types fondamentaux. La concaténation des `String`s est un cas exceptionnel, les autres types objets n'utilisant pas en général d'opérateurs ;
- les objets sont gérés en mémoire d'une façon très différente de celle utilisée pour les types fondamentaux, ce qui permet un comportement beaucoup plus riche, mais souvent plus délicat à analyser (cf la section 9.3).

Dans la suite de ce chapitre, nous allons étudier les objets et les mécanismes spécifiques qui permettent leur manipulation, en nous basant au départ sur l'exemple des `String`s.

[2] Les méthodes de la classe `Console` sont basées sur les méthodes que nous venons de présenter. La principale différence est que les méthodes `readXxxx` gèrent les erreurs.

9.2 Les méthodes d'instance

9.2.1 Introduction

Les objets sont utilisables à travers de *méthodes* qui permettent des manipulations évoluées. De la même façon que les types fondamentaux peuvent être utilisés avec des opérations qui leur sont propres, les **méthodes d'instance** permettent de définir des opérations propres à un type objet donné. Les méthodes d'instance (qu'on peut aussi appeler **méthodes d'objet**) ne s'utilisent pas de la même façon que les méthodes de classe que nous avons étudiées aux chapitres 3 et 7. Comme les méthodes de classe, les méthodes d'instance sont associées à une classe, celle du type objet auquel elles sont associées.

Dans cette section, nous allons étudier comment manipuler plus complètement les `String`s et apprendre par ce biais à utiliser des méthodes d'instance.

9.2.2 Principe

Un exemple

Avant d'étudier précisément les méthodes d'instance, commençons par observer un exemple simple :

Exemple 9.15 :

On considère le programme suivant :

DemoLength

```
1 public class DemoLength {
2   public static void main(String[] args) {
3     String s="ABCD";
4     System.out.println(s.length());
5     System.out.println("UVW".length());
6   }
7 }
```

Ce programme affiche :

AFFICHAGE

```
4
3
```

On constate donc que `s.length()` a pour valeur le nombre de caractères de la chaîne représentée par `s`. `length` est une méthode d'instance de la classe `String` et s'applique donc aux objets de type `String`.

Une méthode d'instance

La définition d'une **méthode d'instance** est strictement identique à celle d'une méthode de classe (cf la section 3.1.1) : c'est une suite d'instructions, désignée par un identificateur et élément d'une classe. Elle s'appelle avec des paramètres et produit éventuellement un résultat.

La seule différence entre les deux catégories de méthodes est qu'**une méthode d'instance possède obligatoirement un paramètre**. Ce paramètre est **l'objet appelant** et est obligatoirement du type[3] de la classe qui définit la méthode.

[3] En fait, son type doit être un sous-type de celui de la classe qui définit la méthode (cf [10]), mais ce problème dépasse le cadre de cet ouvrage.

De plus, l'objet appelant n'est pas utilisé comme un paramètre classique : un appel de méthode d'instance se réalise en donnant un objet, suivi d'un point, suivi de l'identificateur de la méthode appelée (avec éventuellement des paramètres). Si on reprend l'exemple, l'appel `s.length()` obéit à ces nouvelles règles : `s` est une variable de type `String` et contient[4] donc un objet de type `String`. Cet objet est suivi d'un point et de l'identificateur de la méthode appelée.

Une impression de déjà-vu

Nous avons déjà vu des exemples de méthodes d'instance : les méthodes `print` et `println` (cf la section 3.5.2). En effet, `System.out` est une constante de la classe `System`. Cette constante "contient" un objet de type `PrintStream` qui définit des méthodes d'instance `print` et `println`. L'écriture `System.out.println` est donc tout simplement un appel de méthode d'instance.

Nous avons aussi utilisé les méthodes d'instance associées au type objet `Graphics` dans les chapitres 6 et 8. Cela explique pourquoi nous avons toujours utilisé une construction de la forme `g.drawLine(1,2,3,4)`, car `g` correspondait à un objet de type `Graphics`.

Compilation et exécution de l'appel

L'appel d'une méthode d'instance est traité exactement comme celui d'une méthode de classe. La seule nouveauté est l'objet appelant. Comme nous l'avons dit dans les paragraphes précédents, cet objet doit être du type de la classe qui définit la méthode d'instance. Cette règle est vérifiée à la compilation, comme le montre l'exemple suivant :

Exemple 9.16 :

Dans le programme suivant, le programmeur tente d'appeler la méthode `length` de différentes façons erronées :

BadLength

```
1 public class BadLength {
2   public static void main(String[] args) {
3     char c='A';
4     System.out.println(c.length());
5     System.out.println(String.length("ABCD"));
6     System.out.println(String.length());
7     System.out.println(12.length());
8   }
9 }
```

Le compilateur refuse toutes les utilisations envisagées et donne les messages d'erreur suivants :

Erreur de compilation

```
BadLength.java:7: ')' expected
    System.out.println(12.length());
                           ^

BadLength.java:4: char cannot be dereferenced
    System.out.println(c.length());
                       ^

BadLength.java:5: length() in java.lang.String cannot be applied to
(java.lang.String)
```

[4] Nous employons le mot contient pour simplifier la présentation. La section 9.3 montre que la situation est plus complexe.

```
    System.out.println(String.length("ABCD"));
                              ^
BadLength.java:6: non-static method length() cannot be referenced from a
static context
    System.out.println(String.length());
                              ^
4 errors
```

On remarque que la ligne 7 pose un gros problème au compilateur qui ne comprend vraiment pas ce que le programmeur tente de faire (appliquer une méthode `length` à un entier). Les autres messages d'erreur sont assez délicats à interpréter :

- le message "*char cannot be dereferenced*" est l'expression qu'emploie le compilateur pour indiquer que `char` n'est pas un type objet (c'est un type fondamental) ;
- le message correspondant à la ligne 5 indique qu'il n'existe pas de méthode[5] appelée `length` et prenant comme paramètre un objet de type `String` ;
- enfin, le message correspondant à la ligne 6 indique que la méthode `length` est une méthode d'instance (c'est la signification de "*non-static method length()*") et qu'elle ne peut donc pas être appelée comme une méthode de classe (c'est la signification de "*referenced from a static context*").

Les règles concernant les signatures des méthodes (cf la section 3.3) s'appliquent pleinement. La méthode d'instance `length` (de la classe `String`) a comme signature `length()` : elle doit donc être appelée sans paramètre (excepté bien sûr l'objet appelant). L'exemple suivant montre qu'il n'est pas possible de faire autrement :

Exemple 9.17 :

On considère le programme :

BadLength2

```
1 public class BadLength2 {
2   public static void main(String[] args) {
3     System.out.println("ABCD".length(2));
4     System.out.println("ABCD".length("UV"));
5   }
6 }
```

Le compilateur refuse le programme et donne les messages d'erreur suivant :

Erreur de compilation

```
BadLength2.java:3: length() in java.lang.String cannot be applied to (int)
    System.out.println("ABCD".length(2));
                       ^
BadLength2.java:4: length() in java.lang.String cannot be applied to
(java.lang.String)
    System.out.println("ABCD".length("UV"));
                       ^
2 errors
```

Comme pour un appel de méthode de classe, le compilateur indique donc que la méthode `length` ne peut pas être utilisée avec des paramètres.

[5] Le compilateur ne fait pas ici de distinction entre les méthodes de classe et les méthodes d'instance. Il indique simplement que la seule méthode `length` de la classe `String` ne peut pas prendre comme paramètre un objet `String`.

L'exécution d'un appel de méthode d'instance est complètement identique à celle d'une méthode de classe. Nous reviendrons en détail sur ce point au chapitre 12, quand nous apprendrons à créer nos propres types objet et donc nos propres méthodes d'instance.

Pourquoi des méthodes d'instance ?

Il n'existe pas vraiment de justification technique à l'utilisation de méthodes d'instance[6]. D'un point de vue pratique, on réduit simplement le texte à taper. Si, pour obtenir le nombre de caractères d'une chaîne, on devait utiliser une méthode de classe, on écrirait quelque chose comme `String.length(s)`, où `s` désigne un objet `String`. Il est plus rapide d'écrire `s.length()`.

REMARQUE

Notons que l'appel `String.length(s)` **n'est pas possible**, comme le montre l'exemple 9.16.

Notation pour les méthodes d'instance

Pour documenter des méthodes d'instance, nous utiliserons les conventions de présentation proposées à la section 3.4.1. Pour la méthode `length`, nous obtenons la description qui suit.

La classe `String` définit la méthode d'instance suivante :

`int length()`

Cette méthode (sans paramètre) renvoie la **longueur** (le nombre de caractères) de l'objet appelant (qui représente une chaîne de caractères).

9.2.3 Manipulations au niveau du caractère

Motivation

Nous avons vu jusqu'à présent des applications essentiellement mathématiques et/ou graphiques de la programmation. Dans la pratique, on utilise un ordinateur pour bien d'autres applications, notamment le traitement de textes (au sens large du terme). Le prototype d'une opération de traitement de texte est par exemple le calcul de statistiques sur un texte : combien des lettres dans le texte, combien de mots, etc. Pour pouvoir travailler sur un texte, il faut impérativement accéder aux caractères qui composent ce texte.

Deux méthodes d'instance

Pour connaître le nombre de caractères d'une chaîne, on utilise la méthode d'instance `length` présentée dans la section précédente. Pour accéder à un caractère donné, on utilise la méthode d'instance de la classe `String` qui suit :

`char charAt(int index)`

Cette méthode renvoie le caractère situé à la position `index` dans la chaîne de caractères appelante. On numérote les caractères de gauche à droite, en donnant au premier le numéro 0.

Commençons par un exemple simple d'utilisation des deux méthodes.

[6] Le langage ADA 95 manipule comme `Java` des objets et n'utilise pas la notion de méthode d'instance.

Exemple 9.18 :

On considère le programme suivant :

AfficheTexte

```
1  import dauphine.util.*;
2  public class AfficheTexte {
3    public static void main(String[] args) {
4      Console.start();
5      System.out.print("Entrez un mot : ");
6      String mot=Console.readString();
7      for(int index=0;index<mot.length();index++) {
8        System.out.println(mot.charAt(index));
9      }
10   }
11 }
```

Ce programme affiche chaque caractère du texte saisi sur une ligne distincte. Le point important à noter est que la boucle `for` va de 0 à `mot.length()-1` (inclus). En effet, comme les caractères sont numérotés à partir de 0, le dernier a pour numéro $n-1$ si la chaîne est longueur n. Voici un exemple d'affichage produit par le programme :

AFFICHAGE

```
Entrez un mot : bonjour
b
o
n
j
o
u
r
```

Problèmes d'accès

Si on tente d'accéder à un élément qui n'existe pas, le processeur produit une **exception**, qui provoque l'arrêt du programme. Avant de terminer son exécution, le programme affiche un message assez complexe qui comporte le texte `String index out of range`: suivi de la valeur numérique correspondant au caractère non existant. Si on tente `t.charAt(4)`, où `t` fait référence à une chaîne de caractères ne comportant pas de 5-ième caractère, on obtiendra donc le message `String index out of range`: 4. L'exemple suivant illustre cette situation.

Exemple 9.19 :

Voici un programme qui tente d'accéder à un caractère inexistant :

OutOfRange

```
1  import dauphine.util.*;
2  public class OutOfRange {
3    public static void main(String[] args) {
4      String toto="abcd";
5      System.out.println(toto.charAt(4));
6    }
7  }
```

Quand on lance ce programme, on obtient l'affichage suivant :

Erreur d'exécution

```
Exception in thread "main" java.lang.StringIndexOutOfBoundsException:
String index out of range: 4
        at java.lang.String.charAt(String.java:507)
        at OutOfRange.main(OutOfRange.java:5)
```

Il faut bien noter qu'il s'agit d'une erreur **d'exécution**, pas de compilation.

Applications simples

Pour illustrer les possibilités des méthodes étudiées, donnons une solution élémentaire à un problème de statistique évoqué en introduction : combien trouve-t-on d'occurrences d'un caractère donné dans un texte ?

Voici un exemple de résolution de ce problème :

Exemple 9.20 :

Compte

```
1  import dauphine.util.*;
2  public class Compte {
3    public static int howManyChar(String texte,char lettre) {
4      int result=0;
5      for(int k=0;k<texte.length();k++) {
6        if (texte.charAt(k)==lettre) {
7          result++;
8        }
9      }
10     return result;
11   }
12   public static void main(String[] args) {
13     System.out.println("Dans \"Le soleil brille\", il y a "
14                        +howManyChar("Le soleil brille",'l')
15                        +" fois la lettre l");
16   }
17 }
```

Comment la méthode `howManyChar` fonctionne-t-elle ? Grâce à la boucle `for`, le processeur va étudier successivement toutes les lettres de la chaîne de caractères, dans l'ordre, de la première à la dernière. A chaque fois qu'il rencontre une lettre égale à la lettre recherchée, il incrémente de 1 la valeur de `result` qui contient le nombre de fois où la lettre voulue a été rencontrée. On peut donner l'algorithme de la méthode :

Données :
- `texte`, une chaîne de caractères
- `lettre`, un caractère

Résultat : le nombre d'occurrences de `lettre` dans `mot`.

1. placer 0 dans `result`
2. pour `k` allant de 0 à la longueur de `mot` diminuée de 1

(a) si le caractère numéro `k` de `mot` est égal à `lettre` :
ajouter 1 à `result`

3. Résultat : `result`

Voici maintenant un exemple de construction d'une chaîne résultat par l'intermédiaire d'une suite de concaténation :

Exemple 9.21 :

On souhaite "retourner" une chaîne de caractères, c'est-à-dire produire une nouvelle chaîne dont le contenu est celui de la chaîne de départ retourné. Voici une solution possible, implantée par la méthode `miroir` de la classe suivante :

Miroir

```
1  import dauphine.util.*;
2  public class Miroir {
3    public static String miroir(String s) {
4      String résultat="";
5      for(int i=0;i<s.length();i++) {
6        résultat=s.charAt(i)+résultat;
7      }
8      return résultat;
9    }
10   public static void main(String[] args) {
11     Console.start();
12     System.out.print("Entrez un texte : ");
13     String texte=Console.readString();
14     System.out.println(miroir(texte));
15   }
16 }
```

Voici un exemple d'affichage produit par le programme :

Affichage

```
Entrez un texte : Démonstration
noitartsnoméD
```

La méthode fonctionne très simplement : elle parcourt tous les caractères de la chaîne en les ajoutant à une chaîne résultat initialement vide. Comme l'ajout se fait à gauche, on obtient au final un texte inversé.

9.2.4 Retour sur la notion de classe

Nous savons depuis le chapitre 3 qu'une classe est un groupe d'éléments. Nous avons appris au chapitre 7 à définir des méthodes et des constantes de classe qui sont deux des éléments acceptables dans une classe. Le présent chapitre montre qu'une classe peut aussi contenir les éléments suivants :

– la description d'un type objet :
pour pouvoir créer un objet, il faut définir la façon dont il va représenter ce qu'il est sensé représenter. Pour prendre un exemple simple, nous pouvons considérer `String` et `StringBuffer` (que nous étudierons à la section 9.4). Ces deux types objet représentent des chaînes de caractères. Il y a donc au moins deux façons différentes de représenter informatiquement un même concept. La classe `String` contient la description des éléments qui interviennent

dans la représentation des chaînes de caractères choisie pour le type correspondant. Il en est de même pour la classe `StringBuffer` ;
- la programmation de méthodes d'instance :
  quand une classe définit un type objet, elle lui associe en général un ensemble de méthodes d'instance qui permettent une utilisation simple et efficace du type en question.

Nous étudierons comment définir ces nouveaux éléments au chapitre 12.

9.3 La gestion mémoire des objets

9.3.1 Le problème

Nous avons jusqu'à présent occulté discrètement un problème délicat posé par le type `String`. En effet, nous avons vu des types auxquels on pouvait associer un nombre de cases mémoire élémentaires (les types fondamentaux). Or, ce n'est évidemment pas le cas pour une chaîne de caractères : un texte de 3 lettres occupe moins de place dans la mémoire qu'un texte de 10000 lettres. Donc *a priori*, `String` ne peut pas vraiment être un type *classique*.

En fait, c'est plus précisément la notion de valeur qui pose problème. Une valeur **fondamentale**, c'est-à-dire d'un des types fondamentaux (cf. section 2.1.2, page 18), occupe un nombre de cases fixé (qui dépend de son type). Comme, d'après notre définition, une variable est un groupe de cases *fixé*, on peut utiliser les cases en question pour **écrire** la valeur (en fait sa représentation informatique).

Un objet de type `String` n'occupe pas un nombre de cases dépendant seulement de son type car une chaîne courte occupe moins de place qu'une chaîne longue, sans pour autant que l'une ou l'autre ne soit pas une chaîne de caractères. Un objet est donc représenté en mémoire par un groupe de cases dont le nombre **n'est pas fixé par le type de l'objet**. Comme nous l'avons expliqué à la section 9.2.4, ce type (c'est-à-dire en fait la classe associée) se contente en général de décrire la façon dont l'objet du monde "réel" est représenté en mémoire (voir le chapitre 12 pour des précisions).

Le principal problème est que pour des raisons techniques, une variable doit nécessairement occuper un nombre fixe de cases dans la mémoire. La solution à cette contradiction passe par la notion de **référence** et l'utilisation d'une séparation de la mémoire en deux morceaux : la **pile** et le **tas**.

9.3.2 Les références

Quel est donc le sens de la déclaration d'une variable de type `String` ? En fait, on indique que la variable déclarée est une **référence** sur l'objet qui va être manipulé. Une référence est un moyen informatique de repérer l'emplacement où est rangé un objet dans la mémoire. C'est en quelque sorte *l'adresse* de l'objet dans la mémoire. De plus, une référence prend toujours 4 cases (32 *bits*) de la mémoire[7], quelle que soit la taille occupée par l'objet qu'elle désigne. Le processeur permet d'utiliser une valeur particulière pour une référence : `null`. Cette valeur indique que la variable ne fait référence à aucun objet (cf la section 9.3.6 pour des précisions). Notons que comme pour les variables classiques, les références ne sont pas initialisées.

Comme une référence occupe un nombre de cases fixé, elle peut être rangée dans une variable, mais cela ne règle pas pour autant le problème des objets.

[7] pour les microprocesseurs les plus répandus en 2002 (les Athlons/Durons d'AMD et les Pentiums/Celerons d'Intel). Dans un futur très proche, on passera à 8 cases (64 *bits*) sur tous les microprocesseurs, ce qui permet entre autre de gérer plus de mémoire. Les références à 8 cases sont déjà une réalité pour de nombreux modèles de processeurs, comme les DEC Alpha, les Itaniums d'Intel, etc.

9.3.3 La pile et le tas

En fait, la mémoire de l'ordinateur est découpée en deux parties qui sont utilisées de façon différentes : la **pile** (*stack* en anglais) et le **tas** (*heap* en anglais).

La pile

Jusqu'à présent, nous avons exclusivement travaillé avec la pile. Cette partie de la mémoire possède deux propriétés importantes :

1. elle contient les variables déclarées dans le programme ;
2. le processeur peut créer des zones dans la pile afin de séparer la mémoire d'une méthode de celle d'une autre (voir la section 7.2.6).

Pour des raisons techniques complexes, ces propriétés imposent une contrainte : une partie du contenu de la pile doit nécessairement être déterminée à la compilation. En termes simples, pour que le compilateur soit capable de manipuler les variables grâce à leur nom, il faut impérativement que le contenu de chaque variable occupe un nombre de cases fixé **à la compilation**. C'est pourquoi les variables ne peuvent pas contenir directement les objets et qu'elles doivent être limitées aux types fondamentaux et aux références.

Le tas

Le tas est la deuxième partie importante de la mémoire. Le tas est destiné à contenir les objets. Il n'est pas organisé en zone : le processeur ne peut pas placer de barrière dans le tas. De plus, il est impossible de manipuler **directement** le contenu du tas. Pour travailler avec un objet, il faut nécessairement connaître une référence vers cet objet. **La référence indique en fait l'emplacement de l'objet dans le tas**.

Dans la pile, la position des variables est déterminée à la compilation, ce qui permet aux programmes de les manipuler directement. Dans le tas, la position des objets est déterminée **dynamiquement**, ce qui oblige à utiliser des références. Par contre, les objets peuvent occuper un nombre de cases déterminé lui aussi **dynamiquement** c'est-à-dire à l'exécution du programme. En fait, les objets sont **créés** à l'exécution du programme, contrairement aux valeurs qui sont simplement écrites dans une zone mémoire dont la taille a été déterminée à la compilation.

REMARQUE

La division de la mémoire en pile et tas résulte donc d'un compromis : la pile permet une manipulation simple, mais n'autorise que les éléments dont la taille est déterminée à la compilation. Au contraire, la manipulation du tas est relativement complexe, mais il permet le stockage d'éléments dont la taille est déterminée à l'exécution. La manipulation par référence autorise aussi des manipulations impossibles à réaliser autrement (les effets de bord, cf les sections 9.4.6 et 9.4.7), ainsi qu'une efficacité supérieure dans certains cas (voir la section 9.3.5).

9.3.4 Représentation de la mémoire

La manipulation des références est parfois très délicate, c'est pourquoi il est utile de recourir à une représentation graphique de la mémoire. Nous avons déjà vu une telle représentation à la section 2.1.4 et nous l'avons complétée à la section 7.2.6. Nous devons maintenant introduire la distinction pile *versus* tas et proposer une représentation des références.

Étudions pour commencer le sens d'une déclaration et d'une affectation simple :

```
String texte;
texte="Bonjour";
```

La première ligne crée une variable de type `String` qui ne contient au départ aucune référence (et non pas la référence `null`). On ne peut donc pas afficher le contenu de `texte` qui n'est pas encore défini (aucune différence avec une variable d'un type fondamental).

La seconde ligne demande au processeur de réaliser deux opérations :

1. créer un objet de type `String` et le placer dans la mémoire. Cet objet représentera le texte `Bonjour` ;
2. placer dans la variable `texte` l'adresse de (ou la référence vers) l'objet qui vient d'être créé.

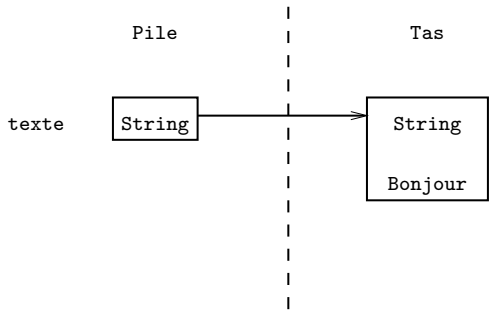

FIG. 9.1 – Création d'un objet et manipulation par référence

La figure 9.1 donne une représentation de l'état de la mémoire après l'exécution de l'affectation. Détaillons cette représentation :

- la mémoire est séparée en pile et tas, avec une identification claire de chaque partie ;
- les variables (ici la variable `texte` seule) sont représentées comme dans les sections 2.1.4 et 7.2.6, par une case précédée du nom de la variable et contenant son type ;
- dans le cas des types fondamentaux (non représenté dans cet exemple), la case contient la valeur éventuelle de la variable ;
- dans le cas des objets, la case de la variable est le point de départ d'une flèche qui représente la référence que la variable contient ;
- le point d'arrivée de la flèche est l'objet désigné par la référence : on représente un objet par une case, contenant le type de l'objet et sa "valeur".

REMARQUE

Il ne s'agit en aucun cas de donner un ensemble de règles formelles pour représenter la mémoire (taille des cases, longueur des flèches, etc.) mais bien de permettre une représentation simplifiée pour analyser plus efficacement des programmes complexes. Nous respecterons donc les principes de représentation illustrés par l'exemple qui précède, en nous attachant avant tout à l'aspect pratique : le but est d'illustrer l'évolution de la mémoire, pas d'ajouter à la difficulté de compréhension une difficulté de représentation !

9.3.5 Retour sur l'affectation

L'affectation proprement dite

Dès le chapitre 2, nous avons étudié une des instructions les plus utilisées, l'affectation (voir la section 2.2). Rappelons l'interprétation par le processeur d'une affectation :

1. le processeur évalue l'expression située à droite du symbole d'affectation (`=`) ;

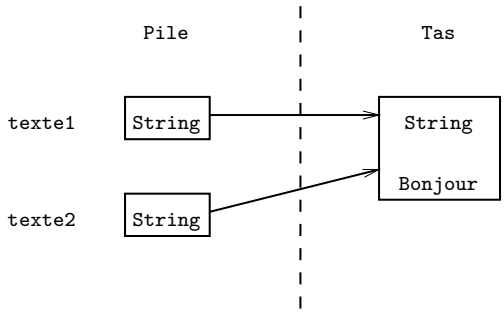

FIG. 9.2 – Deux variables font référence au même objet

2. le processeur place la valeur obtenue dans la variable située à gauche du symbole d'affectation.

Cette interprétation (la sémantique de l'affectation) n'est absolument pas remise en question par l'utilisation des références. Il faut cependant comprendre que les variables contiennent des références, pas des objets. Quand on place le contenu d'une variable dans une autre, on se contente donc de **recopier les références, pas les objets**.

Pour bien comprendre le fonctionnement de l'affectation, étudions un exemple très simple :

```
String texte1,texte2;
texte1="Bonjour";
texte2=texte1;
```

Voici comment interpréter ce programme :

1. la première ligne se contente de créer deux variables (dans la pile). Les variables ne contiennent pour l'instant aucune valeur ;
2. la deuxième ligne a les effets suivants :
   (a) le processeur crée un objet de type `String` (dans le tas) de valeur le texte `Bonjour` ;
   (b) le processeur place la référence vers cet objet dans la variable `texte1` ;
3. la troisième ligne est interprétée de la façon suivante :
   – le processeur évalue l'expression `texte1`. Il obtient comme valeur la référence vers l'objet représentant le texte `Bonjour`, créé à l'instruction précédente ;
   – le processeur place la référence obtenue dans la variable `texte2`.

Après l'exécution de la troisième ligne, les deux variables **font donc référence au même objet**. La figure 9.2 représente la mémoire après l'exécution de l'affectation.

REMARQUE

La représentation proposée par la figure 9.2 utilise bien entendu deux flèches distinctes. La flèche n'est donc pas une représentation du *contenu* de la variable (les contenus des variables `texte1` et `texte2` sont strictement identiques), mais permet *d'illustrer le lien* entre la variable et l'objet : la variable *désigne* l'objet.

Il très important de comprendre que l'affectation ne recopie **jamais** l'objet référencé par l'opération. C'est un avantage et un inconvénient de la manipulation par référence :

– cela permet de conserver à l'affectation toute son **efficacité** : on ne recopie qu'une référence (quatre cases dans la mémoire), même si l'objet référencé est une chaîne très longue. L'opération est donc rapide et elle évite de plus le gaspillage de mémoire ;
– deux variables *distinctes* peuvent être utilisées pour manipuler un objet *unique*. Cela aura des conséquences difficiles à maîtriser (voir la section 9.4.6) ;

– lors d'un passage de paramètre (ou lors de la définition d'un résultat), seule la référence sera recopiée, pas l'objet : la méthode appelée et la méthode appelante pourront donc manipuler le *même* objet, ce qui aura des avantages et des inconvénients (voir la suite de la présente section ainsi que la section 9.4.7).

Dans les méthodes

Les mécanismes de passage de paramètres (section 7.3.3) et de définition d'un résultat (section 7.4.4) sont basés sur la copie des valeurs transmises. Nous venons de voir que la copie de valeur est toujours de mise pour les objets, mais qu'en fait seule les références sont copiées lors d'une affectation. C'est exactement la même chose lors d'une transmission d'information entre deux méthodes : **quand une méthode transmet un objet à une autre méthode, seule la référence est transmise : l'objet n'est pas copié**.

Pour bien comprendre les mécanismes, étudions un exemple volontairement complexe :

Exemple 9.22 :

On considère le programme suivant :

PassageReference

```
1  public class PassageReference {
2    public static String manipulation(String a,String b) {
3      a=b;
4      return b;
5    }
6    public static void main(String[] args) {
7      String u="TOTO",v="ABCD";
8      String w=manipulation(u,v);
9      System.out.println(u);
10     System.out.println(v);
11     System.out.println(w);
12   }
13 }
```

L'affichage produit par le programme est le suivant :

Affichage

```
TOTO
ABCD
ABCD
```

On voit que `u` n'est pas modifié par l'appel, ce qui est parfaitement normal : le contenu d'une variable d'une méthode ne peut pas être modifié par les instructions d'une autre méthode. De façon générale, l'affichage produit semble parfaitement normal. La grosse différence entre les types fondamentaux et les types objet réside dans l'interprétation de l'effet du programme sur la mémoire. Lors de l'appel de la méthode, par exemple, les `String`s ne sont pas recopiées : `a` désigne donc la même chaîne que `u`, par exemple. La figure 9.3 représente l'état de la mémoire juste avant l'exécution de la ligne 3, c'est-à-dire après le passage de paramètres.

Quand la ligne 3 est exécutée, on modifie simplement le contenu de la variable locale `a` (le paramètre formel), qui désigne ensuite le même objet que la variable `b` (la représentation de la chaîne `"ABCD"`). Le résultat de cette opération est illustrée par la figure 9.4.

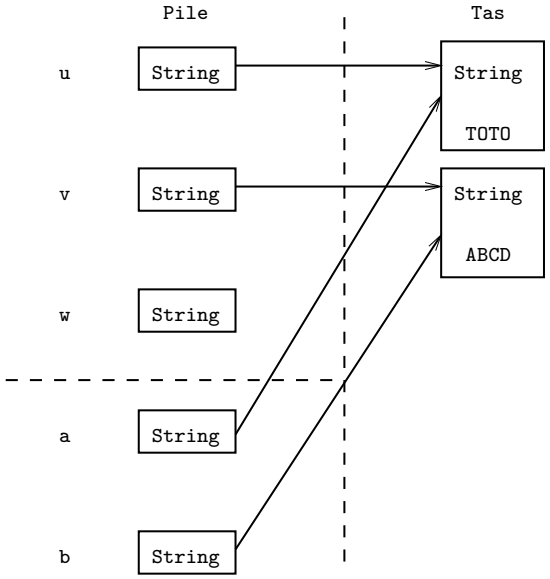

FIG. 9.3 – Passage de paramètres et références

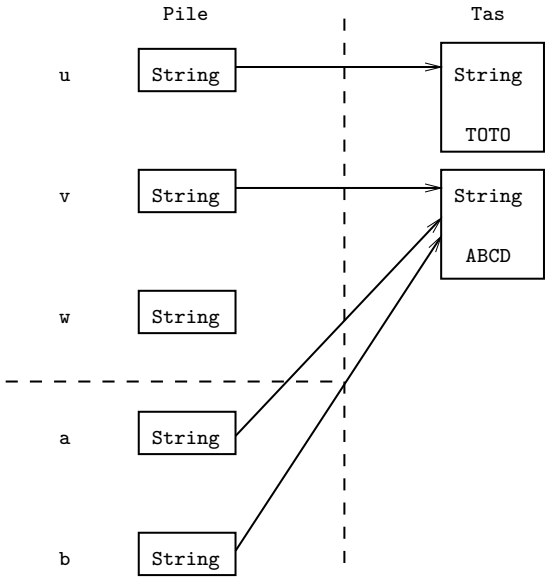

FIG. 9.4 – Copie d'une référence dans une variable locale

Quand la ligne 4 est exécutée, le processeur définit la valeur de retour de la méthode qui est la référence contenue dans `b`. La figure 9.5 représente l'état de la mémoire au retour dans la méthode `main`. Il est important de comprendre la copie des références s'applique à la fois lors de l'entrée dans la méthode (passage des paramètres) et lors la sortie de la méthode (résultat).

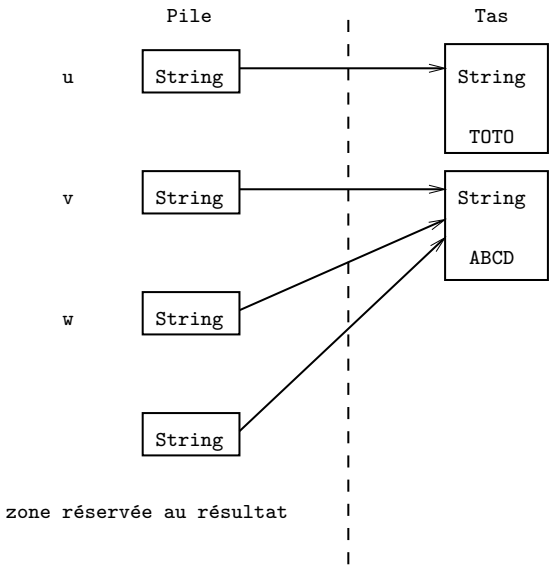

FIG. 9.5 – Référence comme résultat

Nous étudierons d'autres conséquences du mécanisme de passage de paramètres et de définition du résultat à la section 9.4.7.

9.3.6 La référence `null`

Nous savons depuis la section 2.2.5 qu'il est indispensable de donner une valeur à une variable avant de pouvoir l'utiliser. Cette règle s'applique bien entendu aux variables d'un type objet quelconque, comme l'illustre l'exemple suivant :

Exemple 9.23 :

Dans le programme suivant, on essaie d'utiliser une variable de type `String` sans lui avoir donné de contenu :

PasDeValeur

```
1 public class PasDeValeur {
2   public static void main(String[] args) {
3     String s;
4     System.out.println(s);
5   }
6 }
```

Le compilateur refuse le programme et donne le message d'erreur suivant :

ERREUR DE COMPILATION

```
PasDeValeur.java:4: variable s might not have been initialized
    System.out.println(s);
```

```
                              ^
1 error
```

Le point nouveau pour les types objet est l'existence d'une référence particulière, la référence `null`. Cette valeur ne possède pas de type et elle peut donc être placée dans toute variable possédant un type objet (elle reste impossible à placer dans une variable d'un type fondamental). Sa sémantique est simple : elle indique que la variable qui la contient le désigne aucun objet. Considérons d'abord un exemple trompeur :

Exemple 9.24 :

Le programme suivant est parfaitement correct :

Null

```
1 public class Null {
2   public static void main(String[] args) {
3     String s=null;
4     System.out.println(s);
5   }
6 }
```

Il produit l'affichage suivant :

Affichage

```
null
```

A la lumière de cet exemple, on pourrait donc croire que la variable `s` contient une référence vers la chaîne de caractères `"null"`. Il n'en est rien. La méthode `println` est tout simplement capable de faire la différence entre une référence vers une `String` (elle affiche alors la chaîne correspondante) et la référence `null` qui ne correspond à aucun objet (la méthode affiche alors le texte "null").

Pour bien comprendre l'interprétation de `null`, étudions l'exemple suivant :

Exemple 9.25 :

On considère le programme suivant :

PasDObjet

```
1 public class PasDObjet {
2   public static void main(String[] args) {
3     String s=null;
4     System.out.println(s.length());
5   }
6 }
```

Ce programme compile sans problème. Par contre, son exécution est interrompue par une erreur :

Erreur d'exécution

```
Exception in thread "main" java.lang.NullPointerException
        at PasDObjet.main(PasDObjet.java:4)
```

En effet, la référence `null` (*null pointer* en anglais) **ne correspond à aucun objet**. Donc, la tentative d'appel de la méthode d'instance `length` ne peut pas réussir, car une méthode d'instance nécessite un objet appelant. La figure 9.6 représente l'état de la mémoire juste avant

l'exécution de la ligne 4 du programme. Cette illustration précise que la variable `s` contient la valeur `null` et ne correspond donc à aucun objet (d'où l'absence de flèche partant de `s`).

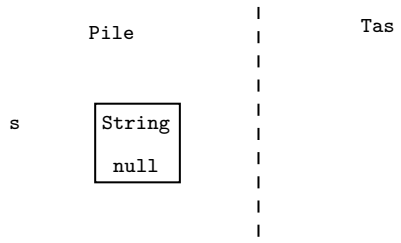

FIG. 9.6 – La variable `s` contient la référence `null`

On peut se demander pourquoi le compilateur accepte la construction et que l'erreur n'est détectée qu'à l'exécution du programme. La réponse est simple : les objets sont créés dynamiquement (en général par l'intermédiaire de constructeur, comme expliqué à la section 9.4.3). Il est donc en général impossible de savoir à la compilation (statiquement) si une variable va bien contenir une référence vers un objet (et non pas `null`) lors de l'exécution (dynamiquement). Le compilateur ne peut donc pas, en général, détecter ce genre de problème.

L'exemple suivant donne une des applications possibles de `null` et illustre l'impossibilité pour le compilateur de vérifier qu'une variable désigne bien un objet.

Exemple 9.26 :

On souhaite réaliser une méthode qui fabrique une sous-chaîne à partir d'une chaîne de caractères donnée. La sous-chaîne est constituée des caractères de la chaîne d'origine depuis le caractère de position `départ` (inclus) jusqu'à celui de position `fin` (non inclus). Si les paramètres `départ` et `fin` sont mal choisis, il est impossible de produire un résultat pertinent. C'est le cas par exemple si `départ` ne désigne pas une position correcte, ou encore si `fin` est plus petit que départ. La méthode `subString` qui fabrique la sous-chaîne commence par éliminer certains cas (lignes 4, 5 et 6). Pour ce faire, elle renvoie `null` au lieu d'une chaîne quelconque. De ce fait, la méthode renvoie un résultat qui correspond réellement à un objet `String` seulement dans le cas où ce résultat a un sens.

Voici le programme proposé :

SousChaine

```
1  import dauphine.util.*;
2  public class SousChaine {
3    public static String subString(String s,int départ,int fin) {
4      if( fin<départ || départ>=s.length() || départ<0) {
5        return null;
6      }
7      String résultat="";
8      fin=Math.min(fin,s.length());
9      for(int i=départ;i<fin;i++) {
10       résultat+=s.charAt(i);
11     }
12     return résultat;
13   }
14   public static void main(String[] args) {
15     Console.start();
16     System.out.print("Entrez un texte : ");
```

```
17      String texte=Console.readString();
18      System.out.print("Début du sous-texte : ");
19      int d=Console.readInt();
20      System.out.print("Fin du sous-texte : ");
21      int f=Console.readInt();
22      String sousTexte=subString(texte,d,f);
23      System.out.println("Résultat : "+sousTexte);
24    }
25  }
```

On remarque que la méthode `main` effectue des saisies. De ce fait, le compilateur ne peut pas savoir (au moment de la compilation du programme) ce que l'utilisateur va saisir à l'exécution. Il ne peut donc pas savoir si la méthode pourra faire un calcul pertinent et renvoyer une référence vers une `String`, ou si au contraire, le calcul sera impossible et que la méthode renverra la référence `null`. Le compilateur doit donc accepter le programme, le processeur se chargeant à l'exécution de découvrir d'éventuels problèmes.

Voici deux exemples d'interaction avec le programme :

Affichage

```
Entrez un texte : Le soleil brille
Début du sous-texte : 3
Fin du sous-texte : 9
Résultat : soleil
```

Affichage

```
Entrez un texte : Le soleil brille toujours
Début du sous-texte : 3
Fin du sous-texte : 2
Résultat : null
```

9.3.7 Les comparaisons

Une première application importante des chaînes de caractères est l'interaction avec l'utilisateur. Jusqu'à présent, nous ne savions pas comment faire saisir à l'utilisateur autre chose qu'une valeur numérique. Grâce aux chaînes de caractères, il est possible de saisir des textes. Une première application simple est de vérifier si l'utilisateur a bien saisi une réponse autorisée (par exemple "oui" ou "non"). Pour ce faire, il faut pouvoir comparer entre elles deux chaînes de caractères. Or, l'utilisation de l'opérateur `==` réserve quelques surprises, comme l'illustre l'exemple suivant :

Exemple 9.27 :

On considère le programme suivante qui simule (de façon très naïve) une identification par mot de passe :

MotDePasseIncorrect

```
1   import dauphine.util.*;
2   public class MotDePasseIncorrect {
3     public static void main(String[] args) {
4       Console.start();
5       String motDePasse="zqdf!!";
6       System.out.print("Donnez le mot de passe : ");
```

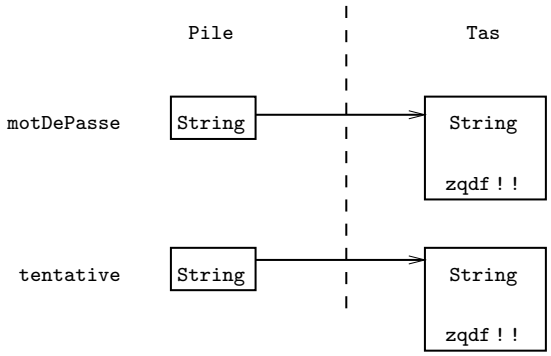

FIG. 9.7 – Deux objets distincts d'apparences identiques

```
7       String tentative=Console.readString();
8       if (tentative==motDePasse) {
9         System.out.println("Accès autorisé");
10      } else {
11        System.out.println("Accès refusé");
12      }
13    }
14  }
```

Contrairement à ce qu'on pourrait croire, ce programme affiche les lignes suivantes :

AFFICHAGE

```
Donnez le mot de passe : zqdf!!
Accès refusé
```

Pour comprendre cet exemple, il faut savoir interpréter la méthode `readString` et l'opérateur `==` :

- la méthode `readString` de la classe `Console` réalise une saisie. Quand l'utilisateur tape un texte, la méthode crée un **nouvel objet** de type `String` dans le tas et renvoie la référence vers cet objet. La figure 9.7 représente l'état de la mémoire après l'exécution de la ligne 7, dans le cas où l'utilisateur saisit le texte `"zqdf ! !"`. On est ici dans la situation des **jumeaux**. Les deux objets `String` représentent la même chaîne de caractères. Ils semblent donc totalement identiques. Pourtant, ils sont distincts et correspondent donc à des **références différentes** (ils occupent des emplacements différents dans le tas) ;
- l'opérateur `==` effectue une comparaison du **contenu** des variables. Dans le cas d'objets, **il compare donc les références**. De ce fait, même si les objets représentent la même chaîne de caractères, ils correspondent à des références distinctes : les contenus des variables `motDePasse` et `tentative` sont donc différents et l'expression `tentative==motDePasse` vaut donc `false`, ce qui explique l'affichage obtenu.

Il est donc impossible en général de comparer deux chaînes de caractères en utilisant l'opérateur `==`. Cet opérateur est très utile, car il permet de comparer deux objets. Mais dans certains cas, il ne convient pas. Comment faire par exemple pour comparer les chaînes de caractères représentées par deux objets différents ?

Certains types objets, comme par exemple les `String`s, définissent une méthode d'instance `equals` qui autorise la comparaison des valeurs représentées par les objets.

Pour la classe `String`, on dispose de la méthode d'instance suivante :

`boolean equals(Object o)`

Si l'objet paramètre `o` est de type `String`, la méthode compare la chaîne de caractères représentée par l'objet appelant avec celle représentée par `o`. Elle renvoie `true` si les chaînes sont identiques. Dans tous les autres cas (si les chaînes sont distinctes ou si `o` ne désigne pas un objet de type `String`), la méthode renvoie `false`.

REMARQUE

Le type `Object` est une sorte de *joker* pour tous les types objets. Son utilisation est assez délicate et, dans le présent ouvrage, nous nous contenterons de quelques mentions brèves. Le lecteur intéressé par des précisions pourra se reporter à [10].

Voici un exemple d'application de la méthode considérée :

Exemple 9.28 :

On reprend le principe de l'exemple 9.27, mais en tenant compte du fait qu'on ne doit pas comparer les objets référencés par `motDePasse` et `tentative`, mais les chaînes qu'ils représentent :

MotDePasse

```
1  import dauphine.util.*;
2  public class MotDePasse {
3    public static void main(String[] args) {
4      Console.start();
5      String motDePasse="zqdf!!";
6      System.out.print("Donnez le mot de passe : ");
7      String tentative=Console.readString();
8      if (tentative.equals(motDePasse)) {
9        System.out.println("Accès autorisé");
10     } else {
11       System.out.println("Accès refusé");
12     }
13   }
14 }
```

Cette version fonctionne parfaitement, grâce à la méthode `equals`.

REMARQUE

Il important de noter que seuls certains types objet proposent une méthode `equals` qui autorise une comparaison des valeurs représentées. Nous verrons à la section 9.4.8 que les autres types possèdent une méthode `equals` qui réalise une comparaison des références, exactement comme l'opérateur `==`.

Il est parfois utile de savoir si une variable d'un type objet quelconque contient effectivement une référence vers un objet ou bien la référence `null`. Pour ce faire, il suffit simplement de comparer le contenu de la variable avec `null`. Si la variable s'appelle `x`, on écrira donc simplement `x==null`, expression qui vaut `true` si et seulement si la variable `x` contient la référence spéciale `null`, c'est-à-dire ne désigne aucun objet.

Pour terminer provisoirement avec les comparaisons, voici un dernier exemple d'application :

Exemple 9.29 :

Dans l'exemple 9.6, nous avons définit une version de la classe `Couleur` qui permet la transformation d'un entier représentant une couleur en une chaîne de caractères décrivant cette couleur. Grâce à la méthode `equals`, il est possible d'effectuer la transformation inverse. Pour ce faire, nous proposons la version suivante de la classe :

Couleur

```
1  public class Couleur {
2    public static final int ERREUR=-1;
3    public static final int CARREAU=0;
4    public static final int COEUR=1;
5    public static final int PIQUE=2;
6    public static final int TREFLE=3;
7    public static final String CARREAU_S="Carreau";
8    public static final String COEUR_S="Coeur";
9    public static final String PIQUE_S="Pique";
10   public static final String TREFLE_S="Trèfle";
11   public static String toString(int couleur) {
12     switch(couleur) {
13     case CARREAU:
14       return CARREAU_S;
15     case COEUR:
16       return COEUR_S;
17     case PIQUE:
18       return PIQUE_S;
19     case TREFLE:
20       return TREFLE_S;
21     }
22     return "Erreur";
23   }
24   public static int parseCouleur(String texte) {
25     if(texte.equals(CARREAU_S)) {
26       return CARREAU;
27     }
28     if(texte.equals(COEUR_S)) {
29       return COEUR;
30     }
31     if(texte.equals(PIQUE_S)) {
32       return PIQUE;
33     }
34     if(texte.equals(TREFLE_S)) {
35       return TREFLE;
36     }
37     return ERREUR;
38   }
39 }
```

Nous avons profité de l'ajout de la méthode `parseCouleur` pour introduire une constante ne correspondant à aucune couleur pour traiter les cas d'erreurs. Nous avons aussi ajouter des

constantes pour la traduction en `String`s des couleurs. Ceci permet de changer la traduction (par exemple pour traduire le programme en anglais), avec le minimum de changement dans le programme (chaque texte apparaît une seule fois).

9.3.8 Le nettoyage du tas

Considérons un programme très simple :

```
String u="abcd";
u="efgh";
```

L'objet de type `String` qui représente la chaîne "abcd" n'est plus référencé après l'exécution de la deuxième ligne. Plus précisément, il n'existe plus dans la mémoire de variable qui contient une référence vers cet objet. De ce fait, il n'existe plus aucun moyen d'utiliser cet objet. On peut alors se demander ce que le processeur va en faire.

En fait, `Java` est muni d'un mécanisme appelé le *garbage collector*, terme américain qui signifie **éboueur**[8]. Le rôle de cet éboueur est de ramasser les ordures, c'est-à-dire les objets qui ne sont plus utilisés. Ce mécanisme est régulièrement mis en marche par le processeur et assure que la mémoire n'est pas encombrée par des objets inutiles.

9.4 Chaînes de caractères modifiables : le type `StringBuffer`

9.4.1 Les `String`s ne sont pas modifiables

L'opération de concaténation ne modifie pas les objets `String` avec lesquels elle travaille. Pour nous en convaincre, analysons un exemple de programme très simple :

```
String u="a",v="b",w;
w=u+v;
u+="bcde";
```

La ligne 2 réalise la concaténation des chaînes désignées par les variables `u` et `v`. Cette opération **fabrique un nouvel objet** de type `String` qui représente le résultat de la concaténation. La figure 9.8 donne l'état de la mémoire après la deuxième ligne.

La ligne 3 est plus intéressante. On pourrait croire en effet qu'elle modifie la chaîne de caractères représentée par l'objet `String` auquel `u` fait référence. Ce n'est pas du tout le cas. En fait, l'instruction `u+="bcde" ;` est interprétée comme `u=u+"bcde" ;`. L'évaluation de l'expression provoque la création d'un nouvel objet `String` qui représente le résultat de la concaténation. De ce fait, l'objet que désignait `u` avant l'affectation reste inchangé. La figure 9.9 représente l'état de la mémoire après l'exécution de la ligne 3.

REMARQUE

Nous n'indiquons ici que le cas de l'opérateur de concaténation, mais il faut noter qu'il est **strictement impossible de modifier la chaîne de caractères représentée par un objet de type `String`**.

[8] en français, il est assez commun d'utiliser une traduction un peu "niaise" pour le dispositif en question, à savoir ramasse-miettes, ou, pire encore, glaneur de cellules.

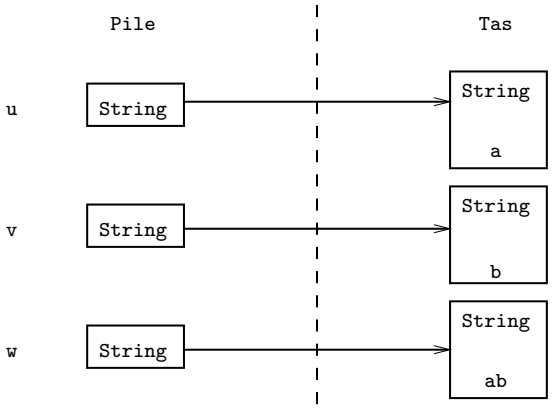

FIG. 9.8 – Résultat d'une concaténation simple (`w=u+v ;`)

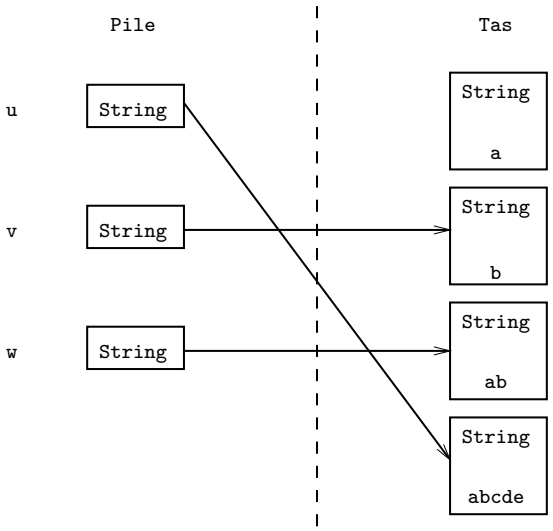

FIG. 9.9 – Résultat d'une concaténation (`u+="bcde" ;`)

9.4.2 Problème d'efficacité

Comme nous venons de le voir, les objets `String` ne sont pas modifiables, et se comportent donc comme des valeurs classiques (c'est-à-dire d'un type fondamental).

Ceci a un inconvénient majeur que nous allons évoquer grâce à l'exemple qui suit :

Concat

```
1 public class Concat {
2   public static void main(String[] args) {
3     String message;
4     message="a"+"b"+"c"+"d";
5     System.out.println(message);
6   }
7 }
```

On se contente ici de mettre bout à bout quatre chaînes réduite chacune à un caractère, afin de fabriquer une dernière chaîne qui est affichée. Il est intéressant de comprendre comment l'opération est effectuée. Comme pour tout calcul, l'ordinateur est simplement capable "d'additionner" deux chaînes. Donc, l'addition est réécrite[9] avec des parenthèses en :

```
message=((("a"+"b")+"c")+"d");
```

Quelle est la conséquence de ces trois additions sur la mémoire de l'ordinateur ? Tout d'abord, le processeur doit créer quatre objets `String` correspondant chacun à une des quatre chaînes de départ. Il effectue alors la première addition : il crée un objet `String` correspondant au texte `ab`. La deuxième addition se solde par la création d'un sixième objet `String` correspondant au texte `abc` et pour finir, un septième objet `String` est créé, correspondant au texte `abcd` et la variable `message` reçoit une référence sur cet objet. Dans cette opération, le processeur a donc créé deux objets inutiles, correspondant aux textes intermédiaires `ab` et `abc`. Or, la création de chaque objet implique la recopie des textes en question depuis les objets d'origine vers le nouvel objet : ici, on copie 5 lettres pour rien. Bien sûr, ce chiffre est faible, mais ceci est une simple conséquence des textes d'origine qui sont tous réduits à une seule lettre. On voit bien qu'avec des textes plus longs, on recopierait de nombreuses lettres de façon inutile. Les `String`s ne permettent donc pas une solution efficace, à la fois en terme de temps nécessaire au calcul, mais aussi en terme de mémoire occupée.

Quelle solution proposer ? Il faudrait pouvoir modifier un objet de type `String`. En effet, on commencerait alors par créer un tel objet correspondant au texte `ab` (2 copies de lettres), puis on ajouterait la lettre `c` (une copie) et la lettre `d` (une copie), soit un total de quatre copies. Dans la version précédente, le total est de cinq plus la dernière concaténation qui provoque quatre copies : neuf copies en tout.

Fort heureusement, il existe une classe, `StringBuffer`, qui propose des chaînes de caractères **modifiables**. Nous allons voir dans la suite du texte comment utiliser cette classe. La principale conséquence de son existence est la possibilité d'avoir des concaténations (et d'autres opérations) plus efficaces. Mais, son utilisation pose des problèmes intéressants qui font que cette classe ne peut pas remplacer la classe `String` mais seulement la compléter.

REMARQUE

Nous sommes confronté de nouveau à la différence entre un concept et sa représentation informatique. En `Java`, il existe plusieurs types fondamentaux pour représenter un entier relatif (`int` et `long` par exemple). De la même façon, il existe plusieurs types objet pour représenter une chaîne de caractères : `String` et `StringBuffer`.

[9] En fait, elle *serait* réécrite de cette façon si les `StringBuffer` n'existaient pas.

9.4.3 Le type StringBuffer et les constructeurs associés

StringBuffer

Nous allons donc utiliser une nouvelle classe : **StringBuffer**. Le premier problème qui se pose est celui de la création d'un objet instance de la classe **StringBuffer**. En effet, comme nous l'avons indiqué à la section 9.1.5, seul le type objet **String** possède des valeurs littérales. De plus, les types **String** et **StringBuffer** ne sont pas directement compatibles, comme le montre l'exemple suivant :

Exemple 9.30 :

On tente naïvement d'utilisateur une valeur littérale de type **String** pour fabriquer un objet de type **StringBuffer** :

BadString

```
1 public class BadString {
2   public static void main(String[] args) {
3     StringBuffer message="Bonjour";
4   }
5 }
```

Le compilateur refuse le programme et affiche le message suivant :

ERREUR DE COMPILATION

```
BadString.java:3: incompatible types
found   : java.lang.String
required: java.lang.StringBuffer
    StringBuffer message="Bonjour";
                         ^
1 error
```

Le compilateur indique donc clairement que les deux types **String** et **StringBuffer** ne sont pas compatibles.

Les constructeurs

Pour tous les types objet excepté **String**, le seul moyen de créer un objet est de passer par un **constructeur**. Un **constructeur** est une méthode spéciale d'une classe qui se charge de créer un objet du type correspondant à la classe. Le nom de cette méthode est toujours le nom de la classe. Une classe peut posséder plusieurs constructeurs à condition que chaque constructeur demande des paramètres différents et possède donc une signature unique (c'est la situation normale des méthodes, voir la remarque 3.4.2). L'utilisation générale d'un constructeur est :

Nom du type de la variable *identificateur*=new *Nom de la Classe*(*paramètres*);

L'instruction **new** indique qu'on demande au processeur de créer un objet. Le nom du type de la variable est aussi le nom de la classe. Le constructeur portant nécessairement le nom de la classe, l'expression qui suit le symbole d'affectation = correspond à l'utilisation du constructeur. Dans certains cas, on utilise un constructeur sans paramètre, mais les parenthèses restent obligatoires. Dans la pratique, les constructeurs jouent pour les objets le rôle joué par les valeurs littérales pour les types fondamentaux.

Pour la classe `StringBuffer`, on utilisera avant tout les constructeurs suivants :

- `StringBuffer()`.
  Ce constructeur sans paramètre fabrique un objet `StringBuffer` correspondant à la chaîne de caractères vide.
- `StringBuffer(String s)`.
  Ce constructeur crée un objet `StringBuffer` correspondant à la chaîne de caractères représentée par `s`.

Exemple 9.31 :

Voici un exemple simple d'utilisation du second constructeur :

Create

```
1 public class Create {
2   public static void main(String[] args) {
3     StringBuffer message=new StringBuffer("Bonjour");
4     System.out.println(message);
5   }
6 }
```

Comme on peut le deviner, ce programme affiche `Bonjour`.

REMARQUE

On remarque dans l'exemple précédent qu'on peut afficher directement un objet de classe `StringBuffer`. En fait, de façon générale, on peut tout afficher avec `System.out.println`, le processeur se chargeant de traduire "automatiquement" le paramètre de la méthode en une chaîne de caractères. Nous reviendrons sur ce point au chapitre 12.

Le cas de `String`

Il faut noter que même si il est pratique d'utiliser les valeurs littérales de type `String`, la classe correspondante définit un ensemble de constructeurs dont voici un sous-ensemble utile :

- `String()`.
  Ce constructeur sans paramètre fabrique un objet `String` correspondant à la chaîne de caractères vide.
- `String(String s)`.
  Ce constructeur crée un objet `String` correspondant à la chaîne de caractères représentée par `s`. Les deux objets sont totalement indépendants.
- `String(StringBuffer s)`.
  Ce constructeur crée un objet `String` représentant la même chaîne de caractères que l'objet `s` paramètre. Les deux objets sont totalement indépendants et une modification du `StringBuffer` n'aura aucune influence sur l'objet `String` ainsi créé.

REMARQUE

Il faut noter que l'utilisation d'un constructeur a deux effets. Le constructeur fabrique un objet dans le tas, et l'expression de création (de la forme `new` *Nom de la Classe*`(`*paramètres*`)`) prend pour valeur la référence vers l'objet nouvellement créé. On peut donc utiliser cette expression partout où une référence vers un objet est attendue. On peut donc écrire : `int x=(new StringBuffer("ABCD")).length() ;`

L'exemple proposé n'est bien sûr par très pertinent, mais, dans la pratique, il est parfois utile de pouvoir créer un objet pour l'utiliser directement, sans placer sa référence dans une variable. Cela peut être le cas par exemple quand on doit utiliser un objet comme paramètre d'une méthode.

Nous avons d'ailleurs déjà utilisé cette technique aux chapitres 6 et 8. En effet, nous avons utilisé la méthode `setColor` de la classe `Graphics` en lui transmettant directement un objet `Color` créé par un constructeur. On écrit par exemple `g.setColor(new Color(110,40,25))`.

9.4.4 Modification d'un objet de type `StringBuffer`

Avant de présenter les méthodes d'instance[10] qui rendent le type `StringBuffer` utile, commençons par une bonne nouvelle : les méthodes d'instance `length` et `charAt` de la classe `String` existent aussi pour la classe `StringBuffer` et se comportent de la même façon. Notons que cette situation est exceptionnelle : il est très rare que deux classes différentes définissent des méthodes de même nom et avec le même comportement.

Étudions maintenant des méthodes d'instance qui permettent de **modifier** un objet de type `StringBuffer`, car ce sont elles qui font la différence avec les `String` :

`void setCharAt(int n,char c)`

Cette méthode permet de changer un caractère dans la chaîne. Si par exemple `message` est un `StringBuffer`, l'appel `message.setCharAt(0,'A')` remplace la première lettre de la chaîne par un `A`.

`StringBuffer append(type x)`

Cet ensemble de méthodes permet d'ajouter la représentation de `x` en chaîne de caractères à la fin de l'objet `StringBuffer` appelant.

REMARQUE

La méthode `setCharAt` ne permet pas de créer des caractères. Si on dispose par exemple d'une variable `sb` faisant référence à un `StringBuffer` représentant le texte `"tot"`, on ne peut pas ajouter une lettre à ce texte en faisant `sb.setCharAt(3,'o')`. Cet appel de méthode provoque au contraire une erreur car on tente de modifier un caractère qui n'existe pas. Voir l'exemple 9.32 pour une illustration du problème.

Exemple 9.32 :

On considère la tentative de création d'un caractère proposée dans la remarque précédente :

BadSetCharAt

```
1 public class BadSetCharAt {
2   public static void main(String[] args) {
3     StringBuffer sb=new StringBuffer("tot");
4     sb.setCharAt(3,'o');
5     System.out.println(sb);
6   }
7 }
```

Le compilateur accepte sans problème le programme. Par contre, une erreur d'exécution se produit, ce qui engendre l'affichage suivant :

ERREUR D'EXÉCUTION

```
Exception in thread "main" java.lang.StringIndexOutOfBoundsException:
String index out of range: 3
```

[10] Nous ne présenterons ici que quelques méthodes parmi les nombreuses qui propose la classe `StringBuffer`.

```
        at java.lang.StringBuffer.setCharAt(StringBuffer.java:362)
        at BadSetCharAt.main(BadSetCharAt.java:4)
```

Voici maintenant des exemples d'application des méthodes de modification :

Exemple 9.33 :

Reprenons le programme simple de concaténation proposé dans la section 9.4.2. Avec un `StringBuffer`, on écrit :

Concat2

```
1  public class Concat2 {
2    public static void main(String[] args) {
3      StringBuffer message=new StringBuffer();
4      message.append("a");
5      message.append("b");
6      message.append("c");
7      message.append("d");
8      System.out.println(message);
9    }
10 }
```

Le principe est donc simple : on ajoute peu à peu des éléments à la fin de l'objet désigné par la variable `message`. Comme cet objet est modifiable, il évolue afin de contenir à la fin du programme une représentation de la chaîne de caractères `"abcd"`. Les figures 9.10 et 9.11 illustrent l'évolution de la mémoire : le processeur ne crée pas d'objets intermédiaires correspondant aux chaînes `"ab"` et `"abc"` alors que c'était le cas avec les `String`. L'utilisation des `StringBuffer`s permet donc de rendre certains programmes plus efficaces.

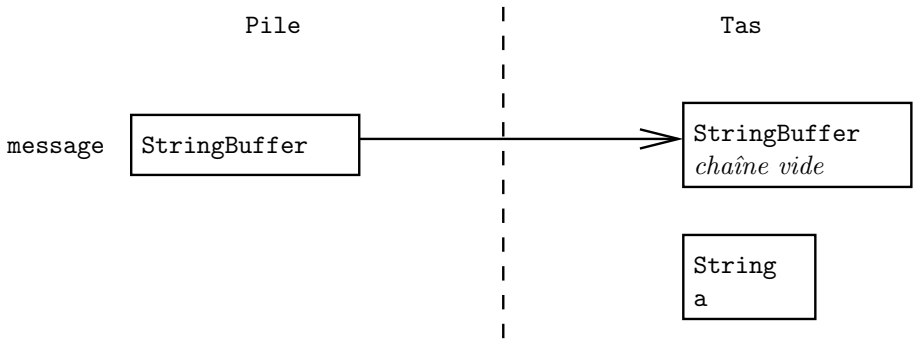

FIG. 9.10 – Mémoire dans le programme de l'exemple 9.33 **avant le premier** `append`

Exemple 9.34 :

Le programme suivant est un jeu du pendu simplifié :

Pendu

```
1  import dauphine.util.*;
2  public class Pendu {
3    public static int howManyChar(StringBuffer texte,char lettre) {
4      int result=0;
5      for(int k=0;k<texte.length();k++)
6        if (texte.charAt(k)==lettre)
```

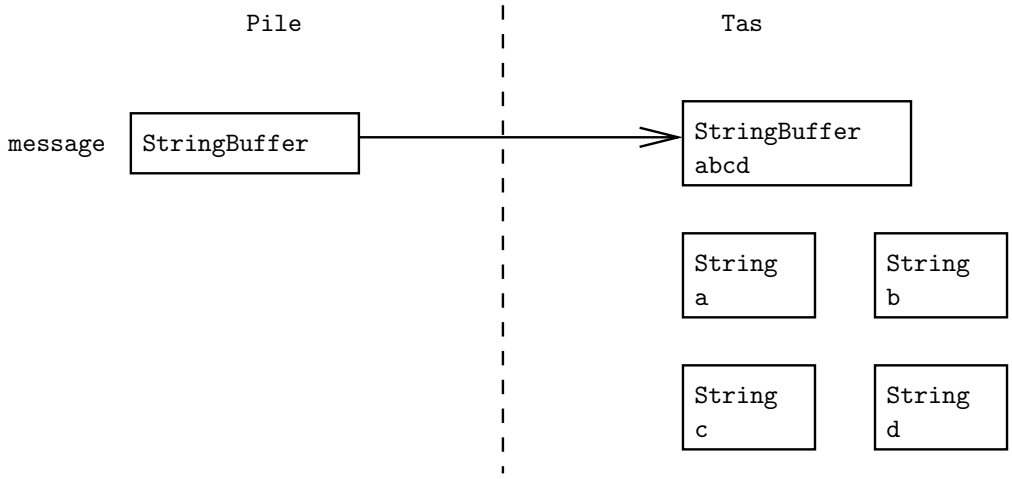

FIG. 9.11 – Mémoire dans le programme de l'exemple 9.33 **après le dernier** `append`

```
7         result++;
8       return result;
9     }
10    public static StringBuffer toGuess(String texte) {
11      StringBuffer result=new StringBuffer(texte);
12      for(int i=0;i<result.length();i++)
13        result.setCharAt(i,'?');
14      return result;
15    }
16    public static void main(String[] args) {
17      Console.start();
18      String aDeviner="Bonjour";
19      StringBuffer mystere=toGuess(aDeviner);
20      char lettre;
21      while (howManyChar(mystere,'?')>0) {
22        System.out.println("Mot : "+mystere);
23        lettre=Console.readChar();
24        for(int i=0;i<mystere.length();i++)
25          if (aDeviner.charAt(i)==lettre)
26            mystere.setCharAt(i,lettre);
27      }
28      System.out.println("Mot : "+mystere);
29      System.out.println("Bravo !");
30    }
31  }
```

Expliquons les méthodes une à une :

– `howManyChar`
  Nous avons déjà vu cette méthode dans l'exemple 9.20. La seule différence est qu'elle utilise ici un objet de la classe `StringBuffer` au lieu d'un objet de la classe `String`. Ceci ne change rien au corps de la méthode qui se base sur les méthodes d'instance `length` et `charAt` qui existent dans les deux classes. Cette méthode indique combien de fois un caractère apparaît

dans une chaîne.

- `toGuess`
  Cette méthode fabrique un `StringBuffer` qui représente la chaîne qui lui est transmise en paramètre. La chaîne du `StringBuffer` est aussi longue que la chaîne de départ, mais chaque lettre est remplacée par un point d'interrogation pour indiquer que la lettre n'est pas connue. La méthode utilisée pour fabriquer le `StringBuffer` est très simple : on commence par le créer avec pour valeur initiale le texte paramètre de la méthode. L'objet créé a donc la bonne taille. Ensuite, on transforme chaque caractère grâce à une boucle et à la méthode `setCharAt`.
- `main`
  La méthode principale comporte essentiellement une boucle qui affiche le `StringBuffer` produit par `toGuess`. A chaque tour de la boucle, on demande à l'utilisateur une lettre grâce à `readCharAskAgain`. Ensuite, grâce à une autre boucle, on compare la lettre proposée avec toutes les lettres du mot à deviner. A chaque fois que la lettre proposée est égale à celle du mot (ce qui peut bien sûr arriver plus d'une fois), on remplace dans le `StringBuffer` le point d'interrogation par la lettre devinée. La boucle s'arrête quand le `StringBuffer` ne contient plus de point d'interrogation, c'est-à-dire quand l'utilisateur a trouvé toutes les lettres.

REMARQUE

Les méthodes `append` de la classe `StringBuffer` renvoient toutes une référence sur le `StringBuffer` appelant. Si on a deux variables `a` et `b` de type `StringBuffer`, écrire `a.append("toto") ;` suivi de `b=a` est strictement équivalent à écrire `b=a.append("toto") ;`. Ceci permet d'écrire par exemple `a.append("toto").append(" et titi") ;`, ce qui va bien coller successivement à la fin de la première chaîne représentée par `a` la chaîne `toto` puis la chaîne `  et titi`.

9.4.5 Conversions

Nous avons vu à la section 9.1.4 qu'il était possible de convertir n'importe quelle valeur d'un type fondamental en une `String`, en utilisant les méthodes `valueOf` de la classe `String`. Il n'existe pas de méthodes équivalentes pour les `StringBuffer`s. De ce fait, pour convertir un entier en `StringBuffer`, on doit passer par une `String` intermédiaire et écrire par exemple :

```
int x=12;
StringBuffer s=new StringBuffer(String.valueOf(x));
```

Bien entendu, la conversion directe est impossible, comme pour les `String`s. L'affectation

```
StringBuffer s=12;
```

est donc incorrecte et rejeté par le compilateur.

Il est possible de "convertir" un `StringBuffer` en une `String`. Pour ce faire, il faut utiliser la méthode d'instance suivante :

`String toString()`
: Cette méthode (sans paramètre) renvoie une représentation sous forme de `String` de l'objet appelant (un `StringBuffer`).

On peut donc écrire les lignes suivantes :

```
StringBuffer s=new StringBuffer("abc");
String t=s.toString();
```

La chaîne de caractères obtenue est un objet **distinct et indépendant** de l'objet `StringBuffer` de départ. Il faut noter qu'avec le constructeur adapté, cette méthode de conversion est la seule technique permettant de passer de `StringBuffer` à `String`. L'exemple suivant illustre les "conversions" rejetées par le compilateur :

Exemple 9.35 :

On considère le programme suivant :

BadStringBuffer

```
1  public class BadStringBuffer {
2    public static void main(String[] args) {
3      StringBuffer s=new StringBuffer("abc");
4      String t=s;
5      String u=(String)s;
6    }
7  }
```

Le programme est rejeté par le compilateur qui affiche les messages d'erreur suivants :

Erreur de compilation

```
BadStringBuffer.java:4: incompatible types
found   : java.lang.StringBuffer
required: java.lang.String
    String t=s;
             ^
BadStringBuffer.java:5: inconvertible types
found   : java.lang.StringBuffer
required: java.lang.String
    String u=(String)s;
                     ^
2 errors
```

Les types `String` et `StringBuffer` sont jugés incompatibles.

REMARQUE

La méthode de conversion `toString` est centrale en `Java` et nous l'utiliserons pour tous les objets. Nous étudierons cette méthode plus précisément au chapitre 12.

La conversion d'un `StringBuffer` en une `String` associé aux possibilités de modification des premiers permet de proposer des versions efficaces du calcul du miroir d'une chaîne de caractères (exemple 9.21) :

Exemple 9.36 :

On propose donc le programme suivant :

MiroirStringBuffer

```
1  import dauphine.util.*;
2  public class MiroirStringBuffer {
3    public static String miroir1(String s) {
4      StringBuffer résultat=new StringBuffer();
5      for(int i=s.length()-1;i>=0;i--) {
6        résultat.append(s.charAt(i));
7      }
8      return résultat.toString();
9    }
10   public static String miroir2(String s) {
11     StringBuffer résultat=new StringBuffer(s);
```

```
12      for(int i=0;i<s.length();i++) {
13        résultat.setCharAt(i,s.charAt(s.length()-i-1));
14      }
15      return résultat.toString();
16    }
17    public static void main(String[] args) {
18      Console.start();
19      System.out.print("Entrez un texte : ");
20      String texte=Console.readString();
21      System.out.println(miroir1(texte));
22      System.out.println(miroir2(texte));
23    }
24  }
```

La méthode `miroir1` procède de façon très similaire à la méthode `miroir` de l'exemple 9.21 : le résultat est construit progressivement en ajoutant peu à peu les caractères souhaités à la fin de la chaîne résultat. Ici, on parcourt la chaîne de départ de la fin vers le début et on ajoute les caractères à la fin de la chaîne résultat. On obtient bien ainsi la chaîne de départ à l'envers. La solution est efficace parce qu'on peut modifier le `StringBuffer`.

La méthode `miroir2` utilise la technique de la modification des caractères plus que celle de l'ajout. Dans certaines circonstances, elle peut être plus efficace que la méthode `miroir1`, mais les raisons techniques de cette différence d'efficacité dépassent le cadre de ce chapitre. Disons pour simplifier que la modification de la longueur d'un `StringBuffer` prend en général plus de temps que la modification des caractères (sans ajout, ni suppression).

Voici un exemple d'affichage produit par le programme :

AFFICHAGE

```
Entrez un texte : Bozo le clown
nwolc el ozoB
nwolc el ozoB
```

9.4.6 Manipulation par référence et objets modifiables

Pour comprendre les conséquences de la manipulation par référence d'objets modifiable, étudions l'exemple suivant :

Exemple 9.37 :

On considère le programme suivant :

Echange

```
1   public class Echange {
2     public static void main(String[] args) {
3       String a="toto",b="tata",c;
4       c=a;
5       a=a+b;
6       StringBuffer d=new StringBuffer("toto"),e=new StringBuffer("tata"),f;
7       f=d;
8       d.append(e);
9       System.out.println(a+","+b+","+c);
10      System.out.println(d+","+e+","+f);
```

```
11    }
12  }
```

L'affichage produit est :

AFFICHAGE

```
tototata,tata,toto
tototata,tata,tototata
```

Le premier affichage est tout à fait classique. Il signifie qu'après les opérations, `a` désigne un objet qui représente la chaîne `tototata`, `b` désigne la chaîne `tata` alors que `c` désigne la chaîne `toto`.

Le second affichage peut paraître plus étrange. Il signifie en effet qu'après les opérations, `d` contient la chaîne `tototata`, `e` contient la chaîne `tata` alors que `f` contient la chaîne `tototata`. Pourtant, aucune opération n'est intervenue sur `f` après `f=d ;`. Pour comprendre ce qui s'est passé, il faut se rappeler que l'affectation `f=d ;` indique seulement que `f` désigne maintenant le même objet que `d`. Donc, l'opération `d.append(e)`, qui *modifie* l'objet auquel `d` fait référence, a un **effet de bord**, c'est-à-dire qu'elle semble modifier une variable qui n'apparaît pas explicitement dans le texte de l'opération. C'est parfaitement normal car l'objet modifié par l'appel de la méthode d'instance est celui auquel `f` fait *aussi* référence. La figure 9.12 donne le schéma de la mémoire pour les variables `d`, `e` et `f`. Pour les `String`, le problème ne se pose

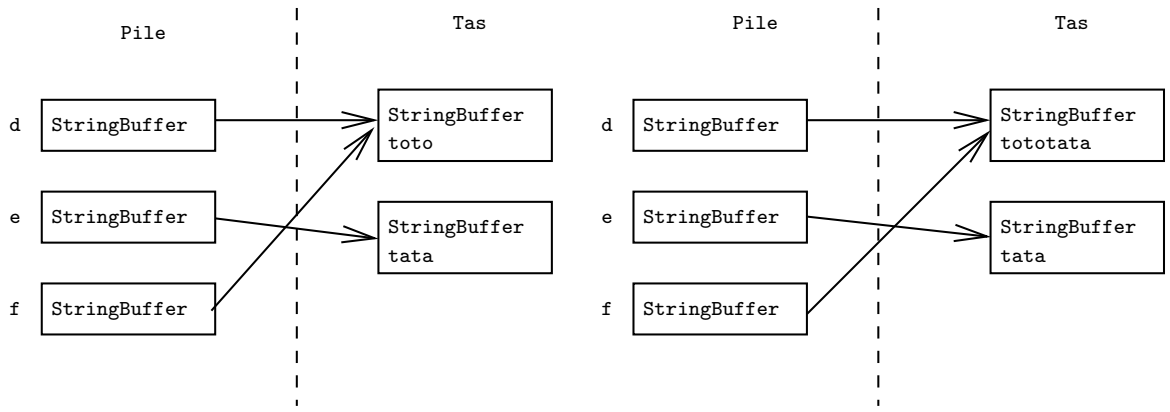

FIG. 9.12 – Modification d'un `StringBuffer`

pas car ces objets ne sont pas modifiables (cf la section 9.4.1). Donc, quand on écrit `a=a+b ;`, on indique simplement que le processeur doit fabriquer un nouvel objet obtenu en concaténant les objets auxquels `a` et `b` font référence, puis placer une référence sur cet objet dans `a`. La figure 9.13 donne le schéma de la mémoire pour les variables `a`, `b` et `c`.

Il est important de comprendre que la différence de comportement entre les `String` et les `StringBuffer` est la conséquence du caractère **modifiable** des objets la classe `StringBuffer` et du caractère **immuable** de ceux de la classe `String`. Tous les objets dont la classe propose des méthodes de modification se comporteront comme les `StringBuffer` et tous les objets dont la classe assure l'impossibilité de modification se comporteront comme les `String` (c'est-à-dire approximativement comme les valeurs des types fondamentaux).

La raison en est simple : quand une variable `a` fait référence à un objet immuable, le seul moyen de changer sa valeur apparente est de lui faire désigner un *autre* objet. De ce fait, même si d'autres variables faisaient référence au même objet que `a`, aucune opération portant sur `a` n'a d'influence

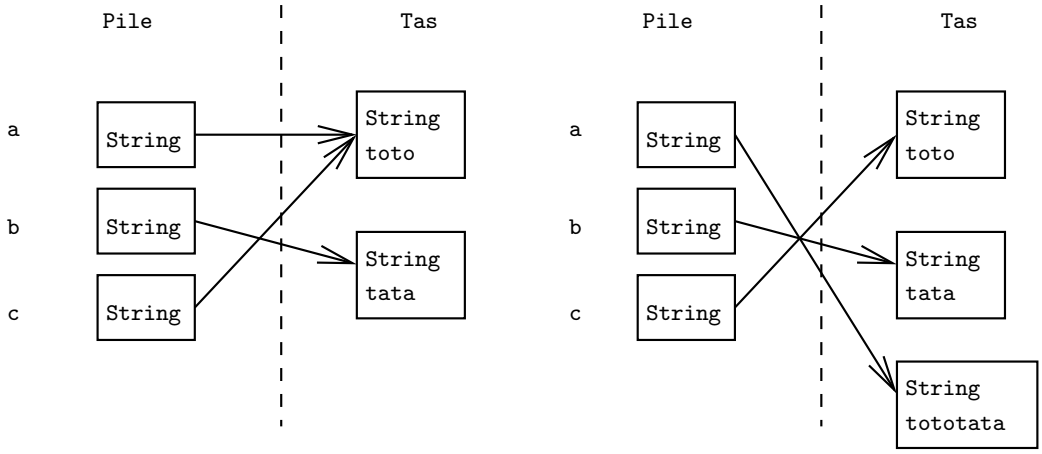

FIG. 9.13 – Les `String` ne sont pas modifiables

sur elles car le seul point commun entre ces variables et `a` est l'objet référencé qui est, par définition, immuable.

Par contre, quand deux variables font référence au *même* objet *modifiable*, on peut utiliser n'importe laquelle des deux pour modifier l'objet, ce qui a des conséquences sur la valeur *apparente* de l'autre variable. Une telle modification est un **effet de bord**.

Il ne faut cependant par croire que les effets de bord peuvent être magiques. **Il reste impossible de modifier le contenu d'une variable sans faire apparaître son identificateur à gauche du symbole d'affectation =**. La nouveauté provient du fait que le contenu non modifiable est simplement une **référence** dans le cas des objets. Quand l'objet lui-même est modifiable, on peut observer un effet de bord. Par contre, il est impossible de changer d'objet sans une utilisation directe de la variable. L'exemple suivant illustre ce point :

Exemple 9.38 :

On considère le programme suivant :

PasDEffetDeBord

```
1  public class PasDEffetDeBord {
2    public static void main(String[] args) {
3      StringBuffer a=new StringBuffer("TOTO");
4      StringBuffer b=a;
5      b.append('!');
6      System.out.println(a);
7      b=new StringBuffer("titi");
8      System.out.println(a);
9      System.out.println(b);
10   }
11 }
```

L'affectation de la ligne 4 indique que `a` et `b` font références au même objet. La ligne 5 provoque de ce fait un effet de bord, qu'on observe depuis `a` grâce à la ligne 6. Par contre, la ligne 7 se contente de changer le **contenu** de `b`, en lui donnant pour valeur une référence vers un **nouvel** objet. Le contenu de `a` ne peut pas être changé par cette opération. L'affichage total obtenu est donc le suivant :

Affichage

```
TOTO!
TOTO!
titi
```

9.4.7 Méthodes de classe et références

Les effets de bord apparaissent encore plus directement quand on utilise des méthodes, comme l'illustre le programme suivant :

Bord

```
1  public class Bord {
2    public static void changeDouble(double a) {
3      a=a+5.5;
4    }
5    public static void changeString(String a) {
6      a=a+" et tata";
7    }
8    public static void changeStringBuffer(StringBuffer a) {
9      a.append(" et tata");
10   }
11   public static void main(String[] args) {
12     double x=0.4;
13     String s="toto";
14     StringBuffer sb=new StringBuffer("toto");
15     changeDouble(x);
16     changeString(s);
17     changeStringBuffer(sb);
18     System.out.println(x);
19     System.out.println(s);
20     System.out.println(sb);
21   }
22 }
```

L'affichage produit est le suivant :

Affichage

```
0.4
toto
toto et tata
```

La première ligne n'est pas une surprise. En effet, on sait que la modification des paramètres d'une méthode n'a pas de conséquence dans la méthode qui l'appelle car les paramètres sont des variables locales de la méthode appelée dans lesquelles on recopie la valeur des paramètres effectifs (revoir la section 7.3.3). Donc, lors de l'appel de `changeDouble`, la valeur de `x`, à savoir `0.4` est recopiée dans la variable `a` (le paramètre formel). Ensuite, cette variable est modifiée, mais ceci n'a aucune incidence sur la variable `x` de la méthode `main`. Le problème est maintenant de savoir ce qui se passe quand le paramètre d'une méthode est un objet.

Comme nous l'avons déjà indiqué à la section 9.3.5, **on ne transmet jamais un *objet* d'une méthode à une autre : seules les références sont transmises**. Quand une méthode comme

`changeString` possède un paramètre formel de type `String` (c'est-à-dire que le paramètre formel va faire référence à un objet), il y a, comme pour les types fondamentaux, création d'une variable classique (dans la pile). Cette variable est initialisée avec la *référence* qui forme le paramètre effectif lors de l'appel. Donc, le paramètre formel fait référence **au même objet** que le paramètre effectif. La figure 9.14 représente l'évolution de la mémoire lors de l'exécution de la méthode `changeString`. Le premier dessin représente la mémoire juste au début de la méthode, quand la référence a été recopiée. On voit bien que les variables `a` et `s` font référence au même objet. Le second dessin montre le résultat de `a=a+" et tata";` qui crée un nouvel objet auquel `a` va faire référence, sans aucune incidence sur `s`. Comme ce mode de transfert est le même pour tous les types objet, il

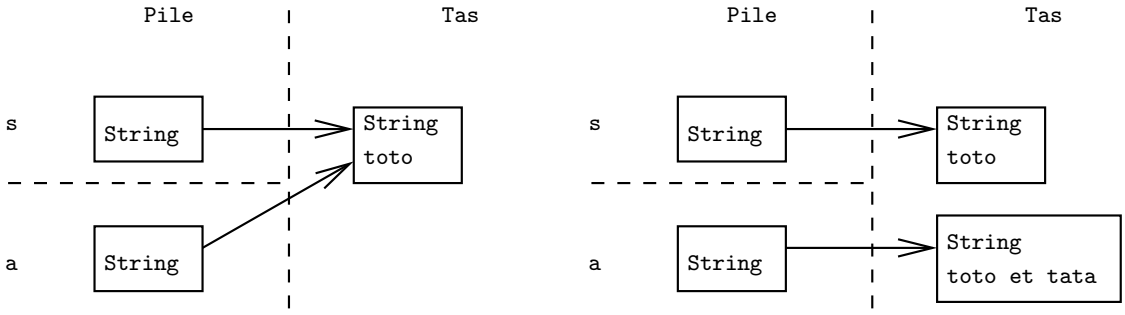

FIG. 9.14 – Passage d'un paramètre `String`

est évident que le comportement est différent si les objets sont modifiables. En effet, comme le paramètre formel fait référence *au même objet* que la variable d'origine de la méthode appelante, toute modification de l'objet par la méthode appelée a une répercussion dans la méthode appelante. C'est exactement ce qui se passe dans le programme présenté ici. Les variables `sb` et `a` (de la méthode `changeStringBuffer`) font référence au même `StringBuffer`. La modification de cet objet par `a.append(" et tata");` est donc visible dans la méthode `main`. Ce comportement est illustré par la figure 9.15.

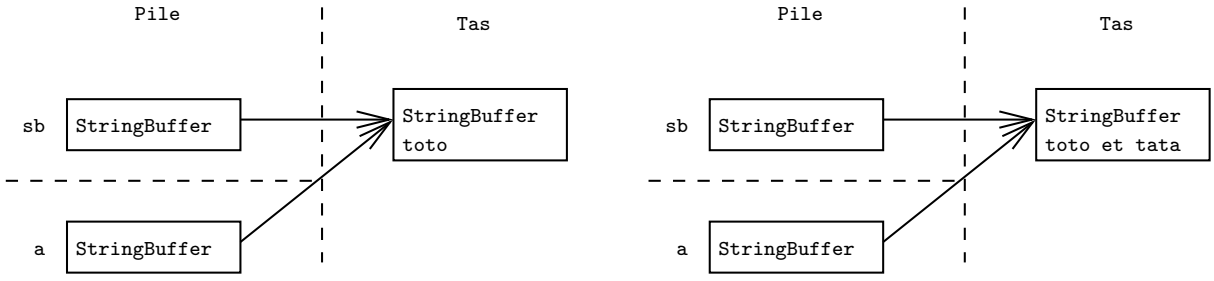

FIG. 9.15 – Passage d'un paramètre `StringBuffer`

REMARQUE

Nous remarquons que les figures 9.14 et 9.15 que la barrière qui sépare les zones mémoires des méthodes ne se prolonge pas dans le tas. En effet, comme nous l'avons indiqué à la section 9.3.3, le tas ne peut pas être découpé en zones. Il est constitué d'un seul bloc et il n'est pas possible de protéger une partie des opérations réalisées par une méthode : dès qu'on possède une référence vers un objet, on peut manipuler cet objet depuis n'importe quelle méthode.

Il faut noter que les problèmes d'effet de bord concernent aussi les résultats d'une méthode. On peut le voir grâce à une version modifiée du programme quie nous venons d'étudier :

BordReturn

```
1  public class BordReturn {
2    public static String changeString(String a) {
3      return a+" et tata";
4    }
5    public static StringBuffer changeStringBuffer(StringBuffer a) {
6      return a.append(" et tata");
7    }
8    public static void main(String[] args) {
9      String s="toto";
10     StringBuffer sb=new StringBuffer("toto");
11     String u=changeString(s);
12     StringBuffer t=changeStringBuffer(sb);
13     System.out.println(s);
14     System.out.println(sb);
15     System.out.println(u);
16     System.out.println(t);
17     t.setCharAt(0,'T');
18     System.out.println(sb);
19     System.out.println(t);
20   }
21 }
```

Le nouveau programme produit l'affichage suivant :

AFFICHAGE

```
toto
toto et tata
toto et tata
toto et tata
Toto et tata
Toto et tata
```

Les quatre premières lignes de l'affichage sont maintenant sans surprise. On sait en effet que le calcul de la concaténation `a+" et tata"` ne modifie en rien le `String` désigné par `a`, ce qui explique le premier affichage. Le second est bien entendu du à l'effet de bord induit par la modification du `StringBuffer` par l'appel `a.append(" et tata")`. Les deux affichages suivant sont logiques et correspondent aux résultats des méthodes.

Les deux derniers affichages sont plus intéressants. On sait en effet depuis la section 9.4.4 que les méthodes `append` renvoient une référence sur leur objet appelant. De ce fait, la ligne 6 du programme précédent est équivalent aux lignes suivantes :

```
a.append(" et tata");
return a;
```

De ce fait, le résultat de l'appel `changeStringBuffer(sb)` est une référence vers l'objet auquel `sb` fait référence. Donc, la ligne 12 du programme est équivalente aux lignes suivantes (rappelons au passage que le résultat d'une méthode ne doit pas obligatoirement être utilisé) :

```
changeStringBuffer(sb);
StringBuffer t=sb;
```

Les deux derniers affichages du programme sont donc clair : il s'agit d'un nouvel effet de bord, induit par le fait que les variables `sb` et `t` font référence à un unique objet modifiable de type `StringBuffer`.

9.4.8 Retour sur les comparaisons

Cas général

Il faut bien comprendre que la manipulation par référence des objets a des conséquences sur **toutes** les opérations faisant intervenir les objets. C'est le cas en particulier des **comparaisons**, que nous avons déjà étudiées à la section 9.3.7 avec l'exemple des `String`s. Il est bien sûr impossible en général de définir un *ordre* sur des objets. Par contre, on peut plus facilement dire si un objet est *égal* à un autre. Plus précisément, si `a` et `b` sont deux variables d'un type objet, les expressions `a==b` et `a !=b` sont parfaitement correctes. Le point délicat est que la comparaison se fait **au niveau des références**. Ceci signifie qu'on ne compare pas du tout les valeurs représentées par les objets, mais au contraire les références qui permettent d'accéder aux objets. Il faut bien comprendre qu'**à chaque objet correspond une référence distincte**, même si deux objets représentent la même valeur apparente (par exemple la même chaîne de caractères). De ce fait, le phénomène des **jumeaux** évoqué à la section 9.3.7 s'applique aux objets en général et la comparaison des références permet de distinguer deux jumeaux.

Voici un exemple de ce phénomène appliqué aux `StringBuffer`s :

Exemple 9.39 :

On considère le programme suivant :

ComparaisonStringBuffer

```
1 public class ComparaisonStringBuffer {
2   public static void main(String[] args) {
3     StringBuffer a=new StringBuffer("Toto");
4     StringBuffer b=new StringBuffer("Toto");
5     System.out.println(a==b);
6     System.out.println(a.equals(b));
7   }
8 }
```

L'affichage produit est le suivant :

AFFICHAGE

```
false
false
```

Le premier affichage n'est pas étonnant : chaque utilisation du constructeur (lignes 3 et 4) produit un objet différent. Les deux `StringBuffer`s obtenus représentent donc la même chaîne, mais ils sont distincts.

Le deuxième affichage est plus étonnant, car il peut sembler en contradiction avec la section 9.3.7 : nous avons vu dans cette section que la méthode d'instance `equals` des `String`s compare les chaînes de caractères représentées par les objets intervenants. En fait, comme nous l'avions indiqué au moment de la présentation de cette méthode, seules certaines classes permettent une comparaison des valeurs. Pour les `StringBuffer`s, ce n'est pas le cas : la méthode `equals` est ici strictement équivalente à l'opérateur `==`. Nous reviendrons sur ce point au chapitre 12.

REMARQUE

Insistons sur le fait que, comme toutes les autres conséquences de la manipulation par référence, l'interprétation de la comparaison réalisée par les opérateurs `==` et `!=` s'applique à **tous** les types objet.

Cas particulier des `String`

Pour les `String`s, l'interprétation est malheureusement un peu plus complexe. Étudions en effet le programme suivant :

ComparaisonString

```
1  public class ComparaisonString {
2    public static void main(String[] args) {
3      String a="Toto";
4      String b="Toto";
5      String c="To"+"to";
6      System.out.println(a==b);
7      System.out.println(a==c);
8      System.out.println(b==c);
9      String d="To";
10     String e="to";
11     c=d+e;
12     System.out.println(a==c);
13     System.out.println(b==c);
14   }
15 }
```

L'affichage obtenu est :

AFFICHAGE

```
true
true
true
false
false
```

Cet affichage est en contradiction avec la règle de comparaison des références. En effet, les objets désignés par `a`, `b` et `c` sont *a priori* distincts et on ne devrait pas avoir d'affichage `true` (dans cette optique, les deux derniers affichages, correspondant aux lignes 12 et 13 du programme, sont cohérents).

En fait, `Java` réalise une simplification des occurrences des valeurs littérales de type `String` (c'est-à-dire des textes entre guillemets). Dans un programme, le compilateur analyse tous les textes et ne conserve qu'une occurrence de chacun d'eux. Dans le programme précédent, cela signifie que les deux occurrences de `"Toto"` (lignes 3 et 4) feront références **à un seul objet** `String` correspondant au texte `Toto`. De plus, le compilateur inclut dans son analyse les **expressions constantes** (cf section 2.3.5). Toute expression produisant une chaîne de caractères et ne faisant pas intervenir de variable est évaluée à la compilation et est traitée comme les valeurs littérales lors de la suppression des occurrences multiples. De ce fait, l'expression `"To"+"to"` de la ligne 5 est évaluée en `"Toto"` et `c` fera donc référence à l'unique objet `String` correspondant au texte `Toto`. Ainsi les variables `a`, `b`

et `c` contiennent-elles la même référence vers cet objet. Ceci explique les trois `true` affichés par le programme.

A la ligne 11, `c` fait référence à un objet `String` obtenu en mettant bout à bout le contenu des objets auxquels `d` et `e` font référence. L'expression de la ligne 11 n'étant pas une expression constante, le calcul a lieu lors de l'exécution du programme et il y a bien production d'un nouvel objet `String` correspondant au texte `Toto`. De ce fait, les deux `false` qui s'affichent à cause des lignes 12 et 13 sont parfaitement logiques : `c` fait référence à un nouvel objet, différent de celui auquel `a` et `b` font référence.

REMARQUE

Le mécanisme mis en œuvre par le compilateur `Java` est très complexe et il n'est pas nécessaire de le retenir parfaitement, d'autant plus que le système utilisé ne s'applique qu'aux objets de type `String` et que la description que nous en donnons est une simplification importante de la réalité.

9.5 Conseils d'apprentissage

Le présent chapitre n'est pas organisé de la même façon que les chapitres précédents. Plutôt que de présenter des outils, puis d'expliquer leurs applications possibles, nous avons préféré partir de problèmes concrets (la manipulation de texte, puis l'efficacité de celle-ci) pour introduire les concepts délicats liés à la manipulation des objets. Il en résulte un chapitre dans lequel les informations concernant tous les types objet sont mélangés avec des éléments qui s'appliquent avant tout aux types `String` et `StringBuffer`. Or ces deux types sont intéressants à la fois en tant que types objet centraux en `Java`, mais aussi comme application réaliste des techniques algorithmiques (boucles, sélection, etc.) étudiées dans les chapitres précédents. Le lecteur peut donc éprouver des difficultés à bien isoler la partie concernant la manipulation des objets et les nouvelles perspectives algorithmiques ouvertes par les opérations possibles sur les `String`s et `StringBuffer`s. Pour faciliter l'apprentissage des notions générales aux objets, voici quelques conseils :

- Il est impératif de bien comprendre que les objets sont manipulés par **référence**, c'est la véritable nouveauté :
  - quand on manipule une variable d'un type objet quelconque, celle-ci contient toujours une référence (éventuellement `null`) qui désigne l'objet qu'elle permet d'utiliser ;
  - le point le plus important à retenir est que **la sémantique des variables reste totalement inchangée** : une affectation se traduit par la copie du contenu de la variable de départ (c'est-à-dire de la référence), une comparaison par `==` se traduit par une comparaison des contenus (c'est-à-dire des références), un passage de paramètre correspond à une copie de la référence, une définition du résultat d'une méthode s'interprète aussi par une copie de la référence ;
  - la principale difficulté vient du fait que la manipulation des objets est **indirecte** alors qu'elle semble **directe** : ceci se traduit par le phénomène complexe des **effets de bord**.
- On doit aussi retenir que la création des objets est dynamiques : on doit impérativement passer par un **constructeur** et l'instruction `new` pour obtenir une référence sur un nouvel objet (sauf dans le cas exceptionnel des `String`s).
- Enfin, excepté la concaténation des `String`s, il faut noter que les opérations sur les objets passent toujours par les **méthodes d'instance** dont la sémantique est très proche de celle des méthodes de classe, mais dont le protocole d'appel est différent : on doit impérativement fournir un **objet appelant**.

Pour bien acquérir les éléments présentés dans ce chapitre, il est utile de commencer par une première étude se focalisant sur l'application proposée des objets, les types `String` et `StringBuffer`.

On peut ensuite relire le chapitre en cherchant à généraliser les notions pour bien comprendre qu'elles s'appliquent à tout type objet.

Notons qu'après ce chapitre, le lecteur est théoriquement capable d'utiliser n'importe quel type objet, en se basant bien entendu sur sa documentation (voir [12]). C'est la raison pour laquelle nous ne donnerons pas dans cet ouvrage une documentation plus complète des diverses classes utiles en `Java` (comme bien entendu les classes `String` et `StringBuffer`). Nous continuerons par contre dans les chapitres suivants (en particulier les chapitres 10 et 13) notre apprentissage du graphisme en `Java`, ce qui nous permettra d'utiliser de nombreux objets, comme nous avons déjà commencé à le faire dans les chapitres d'initiation. Nous conseillons l'apprenti programmeur de compléter l'apprentissage proposé dans le présent ouvrage par la lecture de la documentation de `Java`, en commençant par les classes déjà abordées dans ce chapitre. Ces classes sont très riches et la manipulation de leurs méthodes d'instance permettra au lecteur de travailler les techniques algorithmes importantes. Dans un deuxième temps, le lecteur pourra s'intéresser à des classes importantes en `Java`, comme par exemple `Random` qui permet la gestion des nombres aléatoires, `Locale` qui permet d'adapter un programme aux particularités de la langue de l'utilisateur (représentation des nombres, des dates, etc.), `BigInteger` et `BigDecimal` qui permettent le calcul en précision arbitraire (autant de chiffres après la virgule que nécessaire), etc.

Bibliographie

[1] Alfred Aho, Ravi Sethi, and Jeffrey Ullman. *Compilateurs Principes, techniques et outils*. InterEditions, Paris, 1989.

[2] Alfred Aho and Jeffrey Ullman. *Concepts fondamentaux de l'informatique*. Dunod, 1993.

[3] Mary Campione and Kathy Walrath. *The Java Tutorial Second Edition*. Addison-Wesley, 2000. Disponible à l'URL http://java.sun.com/docs/books/tutorial/index.html.

[4] Bruce Eckel. *Thinking in JAVA*. Prentice-Hall, 1998.

[5] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns*. Addison-Wesley, 1995.

[6] James Gosling, Bill Joy, and Guy Steele. *The Java Language Specification*. Java Series. Addison-Wesley, August 1996. Version 1.0, disponible à l'URL http://java.sun.com/docs/books/jls/index.html.

[7] Mark Grand. *JAVA Language Reference*. O'REILLY, second edition, July 1997.

[8] Tim Lindholm and Frank Yelling. *The Java Virtual Machine Specification*. Java Series. Addison-Wesley, September 1996. Disponible à l'URL http://java.sun.com/docs/books/vmspec/index.html.

[9] Fabrice Rossi. *Initiation à la programmation Java*. Université Paris-IX Dauphine, 1997.

[10] Fabrice Rossi. *Programmation Objet*. Université Paris-IX Dauphine, 1999.

[11] Sun Microsystems. *Code Conventions for the Java Programming Language*, April 1999. Disponible à l'URL http://java.sun.com/docs/codeconv/html/CodeConvTOC.doc.html.

[12] Sun Microsystems. *Java 2 SDK, Standard Edition Documentation (version 1.3)*, 2000. Disponible à l'URL http://java.sun.com/j2se/1.3/docs/index.html.

[13] Andrew Tanenbaum. *Systèmes d'exploitation*. Prentice Hall & InterEditions, 1994.

Index