Utilisation filius

Michel Billaud (michel.billaud@u-bordeaux.fr, michel.billaud@laposte.net)

1er Juillet 2020

Table des matières

1 Objets

Partons d'un exemple concret.

Le programme ci-dessous (écrit en Processing) fait afficher 3 boutons marqués "Plus", "Moins" et "Quitter", ainsi qu'un compteur. Quand on clique sur "Plus" ou "Moins", la valeur du compteur change.

1.1 Le code principal

Le code principal est le suivant :

```
/*
 * Introduction programmation orientés objets
```

```
 */

// variables globales -------------------------------------------------

Button incButton  = new Button(400, 100, "Plus",    color(255, 128, 0));
Button decButton  = new Button(400, 200, "Moins",   color(  0, 128, 0));
Button quitButton = new Button(400, 300, "Quitter", color(128,   0, 0));

int counter = 0;

// --------------------------------------------------------------------

void setup()
{
  size(600, 400);
}

void drawCounter()
{
  textSize(64);
  fill(255);
  text(counter, 200, 200);
}

void mousePressed()
{
  if (incButton.contains(mouseX, mouseY)) {
    counter += 1;
  } else if (decButton.contains(mouseX, mouseY)) {
    counter -= 1;
  } else if (quitButton.contains(mouseX, mouseY)) {
    exit();
  }
}

void draw()
{
  background(0);
  incButton.draw();
  decButton.draw();
  quitButton.draw();
  drawCounter();
}
```

on y voit que les “Buttons” sont

- créés par un appel à `new`, avec leurs paramètres spécifiques (position, texte, couleur)
- utilisés par l’intermédiaire de leurs **méthodes**
  - `draw()` qui les fait se dessiner,
  - `contains` qui leur demande si ils contiennent une position.

Notation : pour demander à `Button` référencé par une *variable* d'exécuter *action*, on utilise la syntaxe *variable.action(paramètres)* que vous avez déjà rencontrée.

1.2 La classe `Button`

Les boutons sont matérialisés par 3 variables `incButton`, `decButton` et `quitButton` de type `Button`.

Ce type `Button` ne fait pas partie de la bibliothèque de Processing, mais est défini par le programmeur d'application lui-même.

Ce type est décrit par la **classe** `Button` :

```
**
 * Un bouton apparait sous forme d'un texte dans une boite de couleur.
 * La boite est de taille fixe.
 */

class Button
{
  final float WIDTH = 100.0;
  final float HEIGHT = 50.0;

  float  x, y;
  String label;
  color  c;

  /**
   * Constructeur.
   */
  Button(float anX, float anY, String aLabel, color aColor) {
    x     = anX;
    y     = anY;
    label = aLabel;
    c     = aColor;
  }

  /**
   * Dessine-moi un bouton (lol).
   */
  void draw() {
    fill(c);
    rect(x, y, WIDTH, HEIGHT);
    fill(255);
    textSize(20);
    textAlign(CENTER, CENTER);
    text(label, x + WIDTH/2, y + HEIGHT/2);
  }

  boolean contains(float mx, float my) {
```

```
    return (x <= mx) && (mx <= x + WIDTH)
      && (y <= my) && (my <= y + HEIGHT);
  }
}
```

Cette classe contient essentiellement

- la déclaration des **attributs** (variables et constantes) qui font partie du bouton,
- un **constructeur** qui indique comment les variables sont initialisées par l'appel à `new`,
- les **méthodes** (fonctions) que l'on peut demander à un bouton d'exécuter.

Remarque : une *méthode* indique comment un `Button` exécute une action. Les instructions de la méthode ont accès aux attributs de l'objet. Par exemple, dans `contains`, on compare une position *(mx,my)* avec sa position *(x,y)* et ses dimensions *(WIDTH, HEIGHT)*.

1.3 Interface de programmation vs. implémentation

Lors de la conception d'un programme, lorsqu'on introduit une classe, la question importante est de savoir ce que le "programmeur d'application" va faire de cette classe.

Ici, pour un `Button`, on va

- le créer avec certaines caractéristiques (position, texte, couleur),
- le faire se dessiner,
- lui demander si un point est situé dans son rectangle.

Nous avons donc un constructeur et deux méthodes qui constituent **l'interface d'utilisation** (ou de programmation, API)de la classe.

D'un autre côté, la classe contient l'**implémentation**, c'est-à-dire les *détails internes* de réalisation d'un bouton, sous la forme

- d'attributs qui servent à mémoriser les informations relatives à un `Button`,
- des méthodes avec leur code.

1.4 Modificateurs d'accès

On a tout intérêt différencier explicitement ce qui fait partie de l'interface "public de ce qui à usage interne de l'objet. On ajoute des *modificateurs d'accès". Exemple :

```
class Button
{
  // ...
  private float  x, y;
  private String label;

  public Button(float anX, float anY, String aLabel, color aColor) {
    ...
  }
```

```
  public void draw() {
    ...
  }

  public boolean contains(float mx, float my) {
    ...
  }
}
```

Le mot-clé `private` signifie “accessible seulement depuis le code de la classe”.

Le compilateur refusera alors - avec raison - qu’on écrive

```
quitButton.label = "Bye";      // pas possible
```

pour modifier directement le `label` d’un bouton, puisque nous nous le sommes interdit.

Les modificateurs d’accès sont un mécanisme de **protection** du programmeur contre une utilisation incorrecte de la classe.

1.5 Méthodes à usage interne

Pour une meilleure qualité de programmation, la méthode `Button ::draw()` pourrait se décomposer :

```
public void draw() {
    drawBackground();
    drawLabel();
}

private void drawBackground() {
    fill(c);
    rect(x, y, WIDTH, HEIGHT);
}

private void drawLabel() {
    fill(255);
    textSize(20);
    textAlign(CENTER, CENTER);
    text(label, x + WIDTH/2, y + HEIGHT/2);
}
```

Les deux méthodes auxiliaires n’étant pas destinées à être appelées de l’extérieur, on les déclare donc `private`. Elles sont invoqués directement depuis `draw()`, et exécutées par la même instance.

1.6 La référence `this`

Dans le code des méthodes, le mot-clé `this` est une référence vers l’instance qui exécute la méthode (*this object*) :

1. On aurait pu écrire, avec le même effet

```
public void draw() {
   this.drawBackground();
   this.drawLabel();
}
```

mais c'est inutile dans ce cas. On ne le fait donc pas.

2. En revanche, on emploie souvent `this` dans les constructeurs, pour initialiser des attributs à partir de paramètres qui ont le même nom. Exemple

```
Button(float x, float y, String label, color c) {
  this.x     = x;
  this.y     = y;
  this.label = label
  this.c     = c;
```

En effet, si un paramètre et un attribut portent le même nom, le `this` lève l'ambiguité.

1.7 Résumé, Terminologie

En Java il y a les données de *types natifs* prédéfinis (`int`, `boolean`, `char`, `double`, …), les tableaux, et les objets.

1. Un **objet** est une entité qui regroupe des variables (**attributs**) et du code (les **méthodes** qu'on peut lui appliquer).
2. Une **classe** est une déclaration de type pour des objets, qu'on appelle **instances** de la classe.
3. Une classe contient la liste de déclarations des attributs et des méthodes, ainsi que des constructeurs.
4. Les constructeurs servent à donner un état initial à une objet.
5. Le code des méthodes a accès aux attributs et méthodes définis dans la classe (exemple : les coordonnées et les dimensions dans `contains`).
6. Une classe est aussi une déclaration de type pour des **variables**, qui peuvent contenir une **référence** à une instance, ou la valeur spéciale `null`.
7. Pour demander à un objet référencé par une variable (invocation d'une méthode), on utilise la syntaxe *variable.methode(paramètres).*
8. Dans une méthode, le mot-clé `this` est une référence à l'instance elle-même. On l'omet généralement quand il n'y a pas d'ambiguité, tant commme "préfixe" pour les attributs, ou comme "sujet" pour l'invocation d'une méthode.
9. Le modificateur d'accès `private` permettent de limiter l'accès à des détails considérés comme "internes à la classe".

2 Interfaces et polymorphisme

2.1 Un exemple concret

Le programme *Processing* suivant fait afficher des “éléments”. De nouveaux éléments (ronds, carrés) seront ajoutés en cliquant sur la fenêtre.

2.1.1 Affichage des d’éléments

Nous disposons d’un `ArrayList` d’éléments, déclaré ainsi

```
ArrayList<Element> elements = new ArrayList<Element>();
```

et l’affichage se fait par une boucle, qui appelle le méthode `paint` de chaque élément :

```
void setup()
{
  size(300, 200);
}

void draw()
{
  background(0);
  for (Element e : elements) {
    e.paint();
  }
}
```

La méthode `paint()` fait donc partie de l’API (interface de programmation) du type `Element` (qui sera déclaré plus loin).

2.1.2 Ajout d’un élément

De nouveaux éléments sont ajoutés dans l’`ArrayList` en cliquant sur la fenêtre.

```
void mousePressed() {
  if (mouseButton == LEFT) {
    elements.add(new Circle(mouseX, mouseY));
  } else {
    elements.add(new Square(mouseX, mouseY));
  }
}
```

remarquez

1. que ce sont des instances de classes différentes (`Circle` et `Square`),
2. que ces instances sont référencées par un tableau d’`Element`

Il y a donc une certaine *compatibilité* entre les types `Circle`, `Square` et `Element`.

2.1.3 Déclaration d’interface

Le type `Element` est déclaré comme suit

```
interface Element {
  void paint() ;
}
```

c'est une *déclaration d'interface **(au sens Java), c'est-à-dire une liste de prototypes de méthodes qui devront être présentes dans les classes qui déclarent** implémenter** cette interface.

Autrement dit : tout `Element` aura une méthode `paint()`.

Rappel : un prototype de méthode, c'est l'entête de la méthode, avec son type de retour, son nom, la déclaration des paramètres (et des exceptions qu'elle peut lever).

2.2 Implémentations

Toute classe qui implémente cette interface doit

- le déclarer dans son entête (mot-clé `implements`),
- définir les méthodes de l'interface, de la manière qui convient (la méthode `paint()` de `Circle` tracera un rond).

C'est ce qu'on retrouve dans

```
class Circle implements Element
{
  float x, y ;
  Circle(float x, float y) {
    this.x = x ;
    this.y = y ;
  }
  @Override
  void paint() {              // trace un rond rouge
    fill(255, 0, 0) ;
    ellipse(x, y, 20, 20) ;
  }
}

class Square implements Element
{
  float x, y ;
  Square(float x, float y) {
    this.x = x ;
    this.y = y ;
  }
  @Override
  void paint() {          // trace un carré jaune
    fill(255, 255, 0) ;
    rectMode(CENTER) ;     // centrage
    rect(x, y, 20, 20) ;
  }
}
```

(l'annotation `@Override`, facultative dans les vieilles versions de java, est fortement conseillée)

2.3 Le polymorphisme

L'introduction de l'interface `Element` rend le code **polymorphe**

```
ArrayList<Element> elements = new ArrayList<Element>() ;

void draw()
{
  background(0) ;
  for (Element e : elements) {
    e.paint() ;
  }
}
```

il est indépendant du type effectif des objets qui sont dans l'arraylist `elements`. Du coup, le code est facilement extensible : on peut introduire des types supplémentaires (`Triangle`, ...) sans avoir à toucher cette partie du code.

Dans la boucle, l'invocation `e.paint()` fera exécuter le code de la méthode `paint` qui convient à l'objet référencé par `e`.

Le **polymorphisme**, c'est la possibilité d'utiliser la même interface indifféremment sur des objets de types différents. Les objets manipulés peuvent prendre plusieurs (grec : poly) formes.

2.4 Résumé

- une déclaration d'`interface` définit un **type** pour des variables,
- elle contient une liste de prototypes de `méthodes`, sans les instructions.
- une interface n'est pas une classe, on ne peut pas en créer des instances par `new`,
- les variables feront référence à des objets, instances de classes qui *implémentent* l'interface,
- il peut y avoir plusieurs classes, c'est le **polymorphisme**
- les classes spécifient (mot-clé `implements`) les interfaces qu'elles implémentent, et définissent leurs méthodes de la manière qui leur est spécifique.

L'utilisation d'interfaces pour séparer

- la définition d'une liste de fonctionnalités attendues
- la réalisation de ces fonctionnalités (implémentation)

permet de d'écrire du code polymorphe, dans lequel les données sont traitées sans avoir à connaître leur type concret, ce qui facilite ensuite l'extension du code.

2.5 Compléments

1. une classe peut implémenter plusieurs interfaces. Exemple

```
interface Drawable {
   void draw() ;
}

interface Animable {
   void animate() ;
}

class Character implements Drawable, Animable
{
   @Override
   void draw() {
      ....
   }
   @Override
   void animate() {
      ....
   }
}
```

2. Une interface peut “étendre” (regrouper) des interfaces existantes, et y ajouter des méthodes

```
interface InteractiveElement implements Drawable, Animable
{
    void onClick() ;
}
```

2.6 Travail

Modifier le programme pour que

- l’ajout d’un rond ou d’un carré se fasse quand on clique à un endroit où il n’y a rien.
- si on clique sur quelque chose qui existe déjà, on peut faire un glisser-déposer.

Indications

1. quand on clique, construire la liste des éléments présents à cet endroit

```
void mousePressed() {
    underTheMouse.clear() ;
    for (Element e : elements) {
        if (e.contains(mouseX, mouseY)) {
            underTheMouse.add(e) ;
        }
    }
    ...
}
```

on peut alors savoir si on en a au moins un, ou pas.

2. lorsqu'on déplace la souris en tenant le bouton, décaler les éléments sélectionnés

```
void mouseDragged() {
    for (Element e : underTheMouse) {
        e.translate(mouseX - pmouseX, mouseY - pmouseY);
    }
}
```

3 Héritage