Fiche 10 : affichage et interaction, bis

IPO – Introduction à la programmation objet

Thibaut Balabonski @ Université Paris-Saclay
`https://www.lri.fr/~blsk/IPO/`

V1, automne 2022

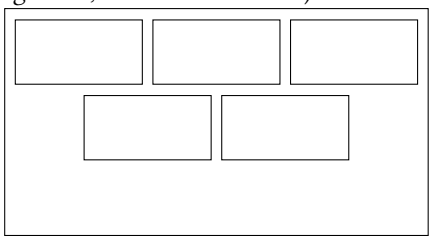
Rappel de la structure principale Une fenêtre d'une application graphique Java est matérialisée par un objet `JFrame`. Une telle fenêtre contient plusieurs éléments, dont notamment :

- un conteneur principal, avec les éléments à afficher (`contentPane`),
- une barre de menu, optionnelle,
- une « vitre » (`glassPane`), invisible par défaut mais recouvrant toute la fenêtre et pouvant servir à bloquer ce qu'elle recouvre.

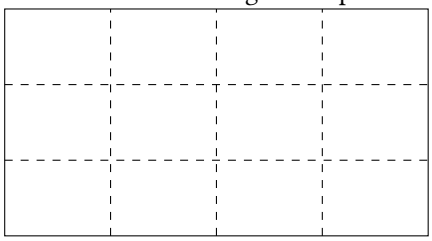
On ajoute au `contentPane` les éléments à afficher, par exemple des instances de `JPanel`.

Disposition des éléments Les conteneurs recevant des éléments peuvent être configurés pour disposer leur contenu d'une manière précise. Ceci vaut pour le `contentPane` d'une fenêtre aussi bien que pour chaque `JPanel`. Rappel : dans un cas comme dans l'autre, chaque nouvel élément doit être ajouté avec la méthode `add` du conteneur. La description d'une disposition est un objet de la classe `LayoutManager`. Java en propose plusieurs sous-classes concrètes prédéfinies. En voici trois exemples courants.

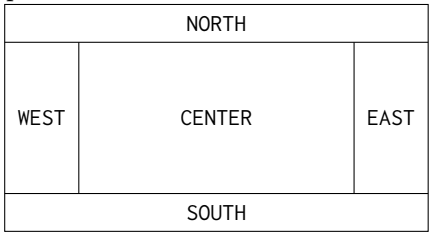
- `FlowLayout` dispose les éléments l'un à la suite de l'autre, dans l'ordre de leur ajout, en passant à la ligne lorsque la ligne courante est pleine, similairement à la manière dont on organiserait les mots d'un paragraphe. Les lignes sont rééquilibrées lorsque l'on modifie la taille du conteneur. On peut spécifier l'espace entre les éléments ainsi que leur alignement (par exemple : à gauche, à droite ou centrés).



- `GridLayout` dispose les éléments en une grille régulière. La taille des « cases » contenant les éléments est calculée en fonction de la taille globale du conteneur (et modifiée avec elle). Comme avec `FlowLayout`, les éléments sont placés dans les cases dans l'ordre des appels à `add`. On peut spécifier l'espace entre les cases, ainsi que le nombre de lignes *ou* le nombre de colonnes (on indique `0` pour ne pas spécifier l'un de ces paramètres, et si on donne des valeurs non nulles pour les deux, alors seul le nombre de lignes est pris en compte).



- `BorderLayout` dispose les éléments en fonction de cinq points de repère : le centre du conteneur et ses quatre bords. Les éléments sont dimensionnés en fonction de la position choisie et de la présence ou non d'éléments dans les zones voisines. On peut spécifier l'espace entre les éléments.



  Pour insérer un élément dans un tel conteneur, on donne en deuxième paramètre à la méthode `add` l'une des cinq constantes `CENTER`, `NORTH`, `WEST`, `SOUTH`, `EAST`. L'absence d'indication équivaut à `CENTER`.

Par défaut, le `contentPane` est configuré avec `BorderLayout`, et les `JPanel` avec `FlowLayout`. Pour changer la configuration, on utilise la méthode `setLayout`, qui prend en paramètre un objet de l'une des classes étendant `LayoutManager`.

```
panel.setLayout(new GridLayout(0, 4)); // 4 colonnes, nb de lignes selon nb d'éléments
```

Éléments pour l'affichage On a déjà mentionné la classe `JPanel`, qui est avant tout un conteneur pouvant regrouper d'autres éléments graphiques. Comme vu au cours précédent, on peut également étendre cette classe et spécialiser sa méthode d'affichage `paintComponent` pour y inclure des dessins arbitraires.

Pour afficher un court texte et/ou une image, on peut utiliser la classe `JLabel`. Le texte initial peut être donné en paramètre au constructeur, et mis à jour avec `setText`.

```
JLabel info = new JLabel("info");
...
info.setText(message);
```

Pour afficher un texte de plusieurs lignes, on a `JTextArea`. Par défaut, un texte de cette dernière forme est éditable par l'utilisateur, mais on peut désactiver cette possibilité.

```
JTextArea painVieuxFourneaux =
  new JTextArea("Sarmentine (farine de meule)\nFleurimeuline du Papé\nCâlinette");
painVieuxFourneaux.setEditable(false);
```

Boutons et actions Un élément interactif élémentaire est le bouton cliquable, donné par la classe `JButton`. Il peut afficher un court texte et/ou une image, à la manière d'un `JLabel`, et est supposé déclencher une action lorsqu'il reçoit un clic de l'utilisateur. Le déclenchement des actions d'un bouton suit le même principe que la réaction aux événements du clavier ou de la souris.

- Le bouton est une source d'événements, ici de type `ActionEvent`.
- D'autres objets peuvent s'inscrire comme récepteurs auprès du bouton pour recevoir une notification à chaque clic. On utilise pour cela la méthode **`void`** `addActionListener(ActionListener r)` du bouton.
- Les récepteurs doivent implémenter l'interface `ActionListener`, qui demande d'implémenter une unique méthode **`void`** `actionPerformed(ActionEvent e)`, qui sera appelée à chaque clic en guise de notification.

```
class Compteur implements ActionListener {
  private int compteur;

  public Compteur() {
    ...
    JButton b = new JButton("+1");
    panel.add(b);
    b.addActionListener(this);
    ...
  }

  public void actionPerformed(ActionEvent e) {
    this.compteur++;
  }
}
```

Ordinairement, une application contiendra plusieurs boutons utilisés pour déclencher des actions différentes. On peut imaginer regrouper ces différentes actions dans une seule méthode `actionPerformed` : il faudrait pour cela, lors de la réception d'une notification, consulter la source (`e.getSource()`) puis chercher le bouton correspondant.

```
class Compteur implements ActionListener {
  private int compteur;

  public Compteur() {
    ...
    JButton b1 = new JButton("+1");
    JButton b2 = new JButton("0");
    b1.addActionListener(this);
    b2.addActionListener(this);
    ...
  }

  public void actionPerformed(ActionEvent e) {
    if (e.getSource() == b1) { this.compteur++; }
    if (e.getSource() == b2) { this.compteur = 0; }
    ...
  }
}
```

Cette stratégie nécessite vite d'écrire une multitude de tests et multiplie les opportunités d'erreur. Une meilleure approche consiste à définir une « micro-classe » implémentant l'interface `ActionListener` pour chacun des boutons à créer, et à associer à chaque bouton une instance de la bonne classe (note : « micro-classe » n'est pas un terme technique officiel).

```
class Bouton1 implements ActionListener {
  public Bouton1(Compteur c) { ... }
  public void actionPerformed(ActionEvent e) {
    c.compteur++;
  }
}
class Bouton2 implements ActionListener {
  public Bouton2(Compteur c) { ... }
  public void actionPerformed(ActionEvent e) {
    c.compteur = 0;
  }
}

class Compteur {
  int compteur;
  public Compteur() {
    ...
    JButton b1 = new JButton("+1");
    JButton b2 = new JButton("0");
    b1.addActionListener(new Bouton1(this));
    b2.addActionListener(new Bouton2(this));
    ...
  }
}
```

Java propose une écriture (très) allégée pour ce processus, qui permet d'éviter de définir explicitement ces micro-classes à usage unique : on donne en paramètre à `addActionListener` une *lambda-expression*, similaire à une fonction Caml, qui décrit la méthode `actionPerformed` correspondante. On peut ainsi écrire « `e -> { ... }` » pour désigner « une instance d'une classe implémentant `ActionListener` avec la méthode **`void`** `actionPerformed(ActionEvent e) { ... }` » (et on ne donne pas de nom à la classe implémentant `ActionListener`, qui ne servira nulle part ailleurs).

```
class Compteur {
  private int compteur;

  public Compteur() {
    ...
    JButton b1 = new JButton("+1");
    JButton b2 = new JButton("0");
    b1.addActionListener(e -> { compteur++; });
    b2.addActionListener(e -> { compteur = 0; });
    ...
  }
}
```

Éléments interactifs Au-delà de `JButton`, la bibliothèque Swing propose une variété d'éléments interactifs : case à cocher (`JCheckBox`), champ de saisie de texte (`JTextField`), sélection dans une liste (`JComboBox`), qui sont encore des sources de `ActionEvent` et peuvent être traités de manière similaire (avec des méthodes spécifiques pour obtenir les informations sur ce qui a été entré ou sélectionné).

Un programme complet Combinons tous les éléments précédents pour écrire un programme interactif complet. On utilisera un certain nombre d'éléments des bibliothèques Java.

```
import javax.swing.*; // JFrame, JPanel, JLabel, JButton, JCombobox, JTextField
import java.awt.*;    // BorderLayout, FlowLayout, Color, Graphics
import java.awt.event.*; // MouseEvent, MouseListener, MouseMotionListener
import java.util.ArrayList;
import java.util.List;
```

Note : l'interface `ActionListener` et la classe `ActionEvent` n'apparaissent pas, car l'écriture allégée des actions ne les fera pas apparaître explicitement.

La fenêtre principale est composée de trois zones :

— une feuille centrale sur laquelle on peut cliquer pour dessiner,
— une barre d'outils en haut, permettant de configurer le pinceau,
— une barre d'information en bas.

La barre d'outils est connectée à la feuille pour en mettre à jour les paramètres, et la feuille et la barre d'information sont toutes deux à l'écoute des actions de la souris (du moins, tant que la souris survole la feuille).

```
public class Gribouille {
  public static void main(String[] args) {
    // Fabrication de la fenêtre elle-même
    JFrame frame = new JFrame("Gribouille");
    frame.getContentPane().setLayout(new BorderLayout()); // redondant avec défaut
    frame.setSize(800, 600); // définition explicite de la taille (largeur, hauteur)
    frame.setVisible(true);
    frame.setDefaultCloseOperation(WindowConstants.EXIT_ON_CLOSE);

    // Création et placement des éléments (définitions ci-dessous)
    Feuille feuille = new Feuille();
    Info info = new Info();
    Outils outils  = new Outils(feuille);
    frame.add(feuille, BorderLayout.CENTER);
    frame.add(outils, BorderLayout.NORTH);
    frame.add(info, BorderLayout.SOUTH);

    // Préparation à la réception des événements
    feuille.addMouseListener(feuille);
    feuille.addMouseMotionListener(info);
  }
}
```

La feuille enregistre un ensemble de bulles à afficher, ainsi que la couleur et le rayon à donner à la prochaine bulle créée. Note : comme on écoute les événements de souris émis par la feuille, les coordonnées apportées par le `MouseEvent` sont relatives à la feuille et non à la fenêtre entière.

```
class Feuille extends JPanel implements MouseListener {
  private List<Bulle> bulles;
  Color couleur = Color.BLACK; // couleur par défaut
  int rayon = 10;              // rayon par défaut

  public Feuille() {
    this.bulles = new ArrayList<>();
    setBackground(Color.WHITE);                      // la feuille est blanche
    setCursor(new Cursor(Cursor.CROSSHAIR_CURSOR)); // curseur croix : plus précis
  }

  // Réaction aux clics de souris sur la feuille
  public void mouseClicked(MouseEvent e) {
    bulles.add(new Bulle(e.getX(), e.getY(), rayon, couleur));
    repaint();
  }
  public void mousePressed(MouseEvent e) {}  // Rien de particulier dans les autres cas
  public void mouseReleased(MouseEvent e) {}
  public void mouseEntered(MouseEvent e) {}
  public void mouseExited(MouseEvent e) {}

  // Servira plus bas : méthode pour tout gommer
  public void effacer() { bulles.clear(); repaint(); }

  // Affichage du dessin
  public void paintComponent(Graphics g) {
    super.paintComponent(g); // rappel : on commence en général par cela
    for (Bulle b : bulles) { // affichage du dessin proprement dit
      g.setColor(b.c);
      g.fillOval(b.x-b.r, b.y-b.r, 2*b.r, 2*b.r);
    }
  }
}
```

La barre d'outils contient trois éléments principaux :

- Un sélecteur de couleurs basé sur un menu déroulant (`JComboBox` contenant une liste de couleurs).
- Un champ de texte (`JTextField`) pour définir le rayon des bulles à créer.
- Un bouton pour tout effacer (`JButton`).

Les deux premiers éléments sont précédés d'une étiquette (`JLabel`) explicitant à l'utilisateur leur signification. Chacun des trois éléments produit des `ActionEvent`, et on spécifie l'action correspondante à l'aide d'une lambda-expression.

```
class Outils extends JPanel {
  private Feuille feuille; // référence vers la feuille pour la configurer
  public Outils(Feuille f) {
    this.feuille = f;
    setLayout(new FlowLayout()); // redondant avec défaut

    // Menu déroulant des couleurs
    this.add(new JLabel("Couleur"));
    Color[] listeCouleurs = { Color.BLACK, Color.BLUE, Color.GREEN,
                              Color.GRAY, Color.PINK, Color.RED };
    JComboBox<Color> couleurs = new JComboBox<>(listeCouleurs);
    couleurs.addActionListener(e -> { feuille.couleur =
                                      (Color)couleurs.getSelectedItem(); }
                              );
    this.add(couleurs);

    // Champ texte pour la sélection du rayon
    this.add(new JLabel("Rayon"));
    JTextField rayon = new JTextField("10");
    rayon.addActionListener(e -> { feuille.rayon = Integer.parseInt(rayon.getText()); });
    this.add(rayon);

    // Tout effacer
    JButton efface = new JButton("Effacer");
    efface.addActionListener(e -> { feuille.effacer(); });
    this.add(efface);
  }
}
```

Bande d'information : position du curseur à l'intérieur de la feuille, mis à jour en temps réel.

```
class Info extends JPanel implements MouseMotionListener {
  private JLabel infoPosition;
  public Info() {
    infoPosition = new JLabel("no info");
    this.add(infoPosition);
  }

  public void mouseMoved(MouseEvent e) {
    infoPosition.setText("x: " + e.getX() + ", y: " + e.getY());
  }
  public void mouseDragged(MouseEvent e) {}
}
```

Bulles.

```
class Bulle {
  final int x, y, r;
  final Color c;
  public Bulle(int x, int y, int r, Color c) {
    this.x = x; this.y = y; this.r = r;  this.c = c;
  }
}
```