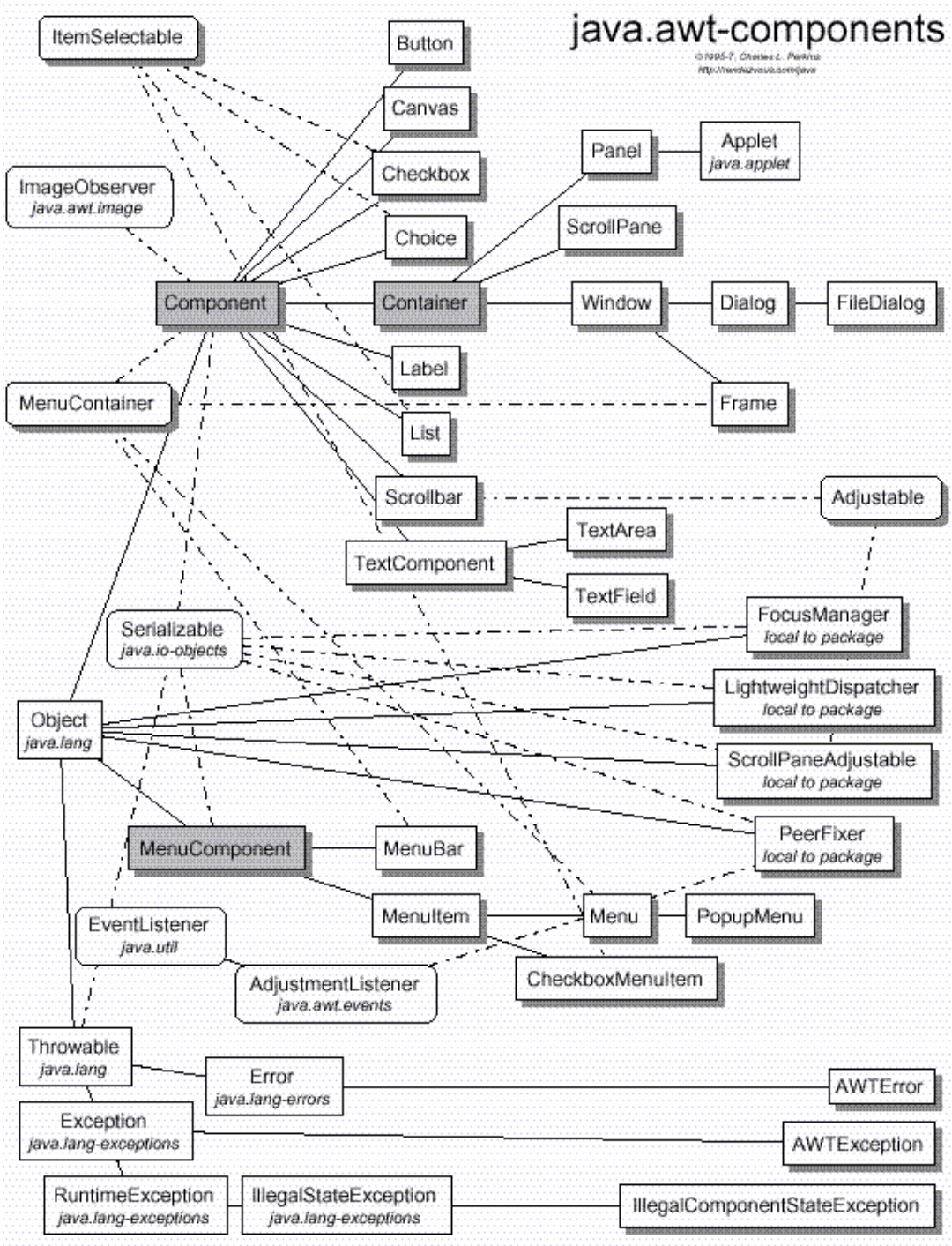
Université de Nice - Sophia Antipolis
Faculté des Sciences

DEUG MIAS MP1

Programmation 2000-01

7. DESSINER AVEC L'AWT



A. Les bibliothèques AWT et Swing

Vous avez déjà pratiqué l'API de Java au 1er semestre avec les applets. Nous allons nous dégager de ces dernières en utilisant directement le package AWT dédié à la construction d'interfaces graphiques. Ce package AWT est ancien, il utilise les

composants [fenêtres, dialogues, boutons] de l'OS sous-jacent [ici Windows], ce sont les composants dits « lourds » [*heavyweight components*]. Apparu avec le JDK 1.2 [dit « plate-forme Java 2 »], le package Swing étend l'AWT en construisant lui-même en Java ces composants, qui deviennent « légers » [*lightweight components*] et donc portables sur divers OS avec le même *look and feel*. Dans un souci de simplification, nous nous bornerons à l'utilisation de l'AWT bien que le passage à Swing ne présente pas de grande difficulté.

Les **composants** forment une classe `Component` située dans le package `java.awt`. Le schéma ci-dessus représente une hiérarchie de classes. Une `Frame` par exemple, qui représente une fenêtre avec bordure, est une `Window`, donc un `Container`, donc un `Component`, et finalement un `Object` !

L'intérêt de ce genre d'architecture propre à la programmation par objets est l'héritage des méthodes. Un `TextField` saura faire tout ce que sait faire un `TextComponent`, peut-être même un peu plus ou un peu différemment. Supposons que `B` soit une sous-classe de `A` , et o_B un objet instance de `B`. Lors d'un appel de méthode o_B`.méthode(…)`, il se peut que :

- *`méthode(…)` soit une méthode de `B` mais pas de `A`.*
- *`méthode(…)` soit une méthode de `A` mais pas de `B`. Elle sera alors trouvée par **héritage**.*
- *`méthode(…)` soit une méthode de `A` redéfinie dans `B`. Celle de `B` masquera celle de A : c'est le **polymorphisme**.*
- *`méthode(…)` ne soit définie ni dans `A` ni dans `B`. Elle sera alors recherchée par héritage en remontant dans les classes parentes de `A`, jusqu'à `Object` s'il le faut. C'est le cas par exemple de la méthode `toString()`. Si elle n'est pas trouvée, c'est une erreur !*

L'héritage et le polymorphisme sont les fondements de la programmation par objets en Java, et permettent d'écrire des systèmes extensibles.

Exercice 7.1 L'adjectif *polymorphe* juxtapose les racines grecques *poly* [plusieurs] et *morph* [forme]. Une méthode polymorphe s'adapte donc au contexte, elle « prend plusieurs formes ». Citez au moins deux cas où vous avez pratiqué le polymorphisme en Java sans le savoir…

B. Construction d'une fenêtre graphique

Nous allons commencer cette incursion dans l'API graphique de Java de manière très simple, en construisant une fenêtre AWT pour y faire des dessins. Ouvrez un nouveau projet *RealJ* nommé `TP5.jpr` dans lequel vous allez construire plusieurs fichiers commentés ci-dessous :

`MP1Frame.java`

La philosophie « objet » de Java consiste non pas à définir une fenêtre dans un programme particulier, mais une classe de fenêtres pour deux raisons : d'une part on pourra les ré-utiliser par ailleurs, et d'autre part cela s'avère intéressant si l'on souhaite avoir plusieurs fenêtres simultanément, donc plusieurs instances de la classe.

```
import java.awt.*;
class MP1Frame extends Frame     // une sous-classe de Frame
{  MP1Frame(String title)              // le constructeur
   {   super(title);                   // invocation (optionnelle) du constructeur de Frame
       this.setBackground(Color.gray); // et deux messages à l'instance « this »
       this.setLocation(20,30);
   }
}
```

• Il faut importer les classes `java.awt.*`, comme par exemple la classe `java.awt.Frame`. Souvenez-vous qu'on fait du Java graphique avec une doc de l'API sous les yeux, soit sous forme livresque, soit avec un navigateur Web.

• On définit donc une sous-classe de la classe `Frame`. Cette dernière hérite de `Window`, que l'on n'utilise jamais car de trop bas niveau. Une `Frame` a des décorations [bordure, case de fermeture, etc] que n'a pas une `Window`.

• Pour utiliser cette nouvelle classe `MP1Frame`, il faudra l'instancier avec `new`, dans une autre classe [cf plus bas].

• Etant dans une sous-classe de `Frame`, on pourra profiter de toutes les méthodes de `Frame`. Mais notre classe est un peu particulière : nos fenêtres sont grises, apparaissent au point de coordonnées <20,30>, etc.

• Cette classe ne contient qu'un constructeur, qui reçoit le nom `title` de la fenêtre, et dont la première ligne invoque en utilisant `super` le constructeur `Frame(String title)` de la classe-mère [cf. Feuille 4].

• Viennent ensuite une cascade de *messages à l'instance `[this]` en construction* : change ta couleur de fond [gris], et lorsque tu apparaîtras, fais-le au point de coordonnées <20,30>. Toutes ces méthodes sont documentées dans l'API. Certaines d'entre elles peuvent ne pas être dans la classe `Frame`, mais héritées d'une classe parente !

Exercice 7.2 Qui sont les classes formant l'ascendance de MP1`Frame` ?

`MP1FrameTest.java`

C'est la classe principale [*Set As Main*], qui se contente de construire la fenêtre principale de l'interface graphique :

```
class MP1FrameTest
{   public static void main (String[] args)
   {   MP1Frame frame = new MP1Frame("Ma première fenêtre");   // on créé une fenêtre[1]
       frame.show();                                          // et on l'invite à se montrer
   }
}
```

Exercice 7.3 a) Compilez puis exécutez. Le cas échéant, agrandissez la fenêtre à la souris en la tirant par son coin inférieur droit. Pouvez-vous fermer la fenêtre en cliquant dans sa case de fermeture ? Vous n'y parvenez pas ? Utilisez le bouton « stop » de *RealJ*, nous verrons comment faire plus bas.

b) Rajoutez en dernière ligne du constructeur `Frame(…)` un message interdisant à l'instance d'être redimensionnée. Vous chercherez la méthode adéquate dans l'API [taille se dit *size* en anglais !]. Recompilez et regardez à l'endroit de la case de redimensionnement. Essayez maintenant de modifier la taille de la fenêtre… Nous opterons ultérieurement pour des fenêtres non redimensionnables.

C. Petit problème : comment gérer la case de fermeture ?

Pourquoi la case de fermeture n'est-elle pas active ? Lorsque vous cliquez sur cette case, vous générez un **événement** [en anglais *event*]. Un événement peut être un clic de souris, un redimensionnement de fenêtre, la pression sur une touche du clavier, etc. Les évènements sont regroupés en classes Java et sont donc des objets pouvant être passés en paramètres à des méthodes et placés dans des variables. Il y a divers types d'événements : de type souris [*MouseEvent*], de type « action » assez général [*ActionEvent*], liés aux fenêtres [*WindowEvent*] etc. mais n'entrons pas dans ces détails trop tôt.

L'événement de type *WindowEvent* résultant d'un clic dans la case de fermeture d'une fenêtre tombe dans l'oreille d'une sourde : la fenêtre n'entend pas. Pour cela, nous allons brancher un *adaptateur* à la fenêtre, mécanisme recevant les évènements et acceptant de traiter certains d'entre eux. Comment faire ? Il se trouve que Java utilise des **interfaces**, classes ne contenant que des en-têtes « abstraites » de méthodes, par exemple[2] :

```
public interface WindowListener extends EventListener
{  public void windowActivated(WindowEvent e);     // invoquée si activée
   public void windowClosing(WindowEvent e);       // invoquée à la fermeture  <------
   ... ... ... ...
   public void windowIconified(WindowEvent e) ;    // invoquée à l'icônification
   public void windowOpened(WindowEvent e);        // invoquée à l'ouverture
}
```

Par exemple, l'API fournit dans le package `java.awt.event` la classe `WindowAdapter` qui implémente trivialement ces 7 méthodes en ne leur faisant rien faire [leur texte entre accolades est vide !] :

```
public abstract class WindowAdapter implements WindowListener
{  public void windowActivated(WindowEvent e) {}     // ne fait rien
   public void windowClosing(WindowEvent e) {}       // does nothing  <-----------------
   ... ... ... ...
   public void windowIconified(WindowEvent e) {}     // shénme dongxi
   public void windowOpened(WindowEvent e) {}        // es macht nichts
}
```

Le fait de déclarer cette classe *abstraite* sans que les méthodes ne le soient est là pour obliger le programmeur écrivant une sous-classe de `WindowAdapter` à redéfinir *au moins l'une* de ces méthodes. C'est d'ailleurs ce que vous allez faire :

- utiliser une sous-classe anonyme de `WindowAdapter` qui va redéfinir la méthode `windowClosing(…)` pour qu'elle fasse quelque chose, à savoir terminer le programme.
- Instancier avec `new` cette classe anonyme pour qu'elle produise un écouteur d'évènements-fenêtres.
- Enregistrer cet écouteur auprès de la fenêtre.

```
import java.awt.*;
import java.awt.event.*;

class MP1Frame extends Frame
{  MP1Frame(String title)            // le constructeur
   {  ... ... ...
      // on enregistre un écouteur de la fenêtre avec une sous-classe anonyme de WindowAdapter
      // qui va redéfinir la méthode vide windowClosing(…) :
      this.addWindowListener( new WindowAdapter()
                              {  public void windowClosing(WindowEvent e)
                                 {  System.exit(0);
                                 }
                              });
```

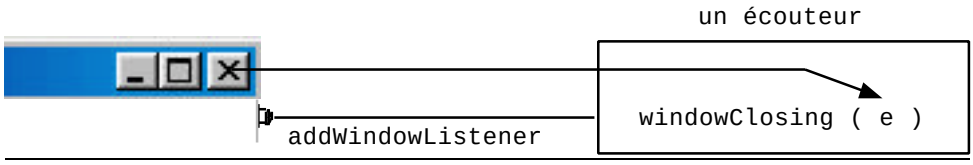
Un écouteur des évènements-fenêtres

Enregistrement de l'écouteur

[1] Il est courant chez les programmeurs Java de noter une variable d'une classe avec le nom de la classe, mais débutant par une minuscule. Ceci évite de passer des heures à trouver des noms adéquats, et de plus indique vite le type de la variable…

[2] Ce texte de l'interface `WindowListener` est extrait de *Java Foundation Classes in a Nutshell* [publié en français chez O'Reilly].

```
}
```



Mécanique : vous cliquez sur la case de fermeture de la fenêtre. Ceci produit un événement `e` de type `WindowEvent` qui est reçu par l'écouteur qui le passe à la méthode `windowClosing(…)` qui provoque un `System.exit(0)` et termine le programme.

Exercice 7.4 Vérifiez que maintenant la case de fermeture est active ! Ouf.

D. Ajout d'un « canvas » : une toile pour peindre !

Le peintre ne dessine pas sur le cadre [cadre = *frame* en anglais], il utilise une toile de lin blanche. La classe `Canvas` [toile = *canvas* en anglais] sert à cela. Un *canvas* est un composant graphique [`Canvas` est une sous-classe de `Component`] au même titre qu'un bouton à cliquer par exemple. Notre but est d'installer un *canvas* à l'intérieur d'une *frame*, tout comme le peintre… Avant de l'installer, fabriquons-le. Que dit l'API ?

> *An application must subclass the Canvas class in order to get useful functionality such as creating a custom component. The paint method must be overridden in order to perform custom graphics on the canvas.*

En clair la classe `Canvas` n'existe que pour être étendue, avec redéfinition [*override*] de la méthode `paint(…)`. Mais c'est bien ce que vous faisiez avec la classe `Applet` au 1er semestre, n'est-ce pas ?…

```
import java.awt.*;
class TP7Canvas extends Canvas
{  private int taille;                        // on utilise un canvas carré
   TP7Canvas (int taille)
   {   this.taille = taille;                  // taille == le côté du carré
       this.setBackground(Color.white);       // une toile blanche
   }
   public Dimension getPreferredSize()        // cf exercice 6
   {   return new Dimension(taille,taille);
   }
   public void paint(Graphics g)              // comme dans les applets,
   {   g.setColor(Color.red);                 // g est un « contexte graphique »
       g.drawLine(0,0,taille,taille);
       g.setColor(Color.blue);
       int x = (int)(taille * 0.8 * Math.random()), y = 10 + (int)( taille * 0.8 * Math.random());
       g.drawString("Salut les matheux !...", x, y);
   }
}
```

Exercice 7.5 A la première ligne du constructeur, nous n'avons pas invoqué explicitement le constructeur de la classe-mère `Canvas`. Qu'est-ce que Java a fait exactement à notre place ?

• Nous avons donc redéfini la méthode `paint(…)` de la classe-mère `Canvas`, qui ne dessinait rien par défaut. La nôtre, dans la sous-classe `TP7Canvas`, dessine une ligne et un texte dans le *canvas*.

Problème : à quel moment s'effectue ce dessin ? A quel moment est appelée la méthode `paint(…)` puisque nulle part nous ne l'appelons nous-mêmes ? Réponse : quand le système le décide ! Par exemple quand la fenêtre a été partiellement recouverte et revient au premier plan, ou bien lorsqu'on la redimensionne.

• Essayez d'icônifier la fenêtre et de la ramener au premier plan, vous verrez que cela active `paint(…)`, puisque le texte se déplace. Plus précisément, cela invoque la méthode `update(…)` qui fait deux choses : effacer le canvas puis invoquer `paint(…)`. Mais nous n'avons pas besoin de ces subtilités pour l'instant, `paint(…)` nous suffira !

Exercice 7.6 a) Cherchez dans l'API la classe `Dimension`. Que représente un objet de type `Dimension` ?
b) Nous avons redéfini la méthode `getPreferredSize()`. Dans quelle classe de l'API se trouvait-elle ? A quel endroit nous en servons-nous ? A quoi sert-elle ?

E. Ajouter le *canvas* à la *frame*

Il ne reste qu'à incorporer le *canvas* à la *frame* comme le peintre insère une toile dans un châssis en bois. Vous allez le faire vous-même. Editez le fichier `MP1FrameTest.java`, et juste avant de montrer [`show`] la fenêtre dans le `main(…)`, incorporez les lignes suivantes :

```
TP7Canvas canvas = new TP7Canvas(200);
frame.add(canvas);
frame.pack();
```

La première ligne définit la variable `canvas`, instance de la classe `TP7Canvas`. La seconde l'ajoute à la *frame*. La méthode `add` n'est pas dans la classe `Frame`. Si l'on remonte avec *Explorer* dans la hiérarchie des classes parentes, on ne la trouve pas non plus dans `Window`, mais dans `Container`. Le message `add` est envoyé à un objet de type `Container` [à un *conteneur*, en particulier à une `Frame`] pour lui signifier de bien vouloir accepter un nouveau *composant* de type `Component`, dont `Canvas` est une sous-classe : un *canvas* peut-être un *composant* d'un *conteneur*… Ouf, il faut vous familiariser un peu avec ces classes, vous comprenez maintenant pourquoi nous nous limitons à l'AWT ? L'instruction `pack()` est obligatoire après avoir ajouté tous les composants à un conteneur, pour que Java puisse les répartir de manière harmonieuse et « compacter » le tout.

Exercice 7.7 *Une méthode de Monte-Carlo*. Pour calculer une valeur approchée du nombre π, on inscrit un cercle dans un carré 300x300, et on place 10000 points au hasard dans celui-ci. La probabilité de tomber dans le cercle est égale à $\pi/4$ [pourquoi ?]. Faites l'expérience dans un canvas et faites afficher à la `Console` la valeur approchée de π obtenue…
Dupliquez `TP7Canvas.java` et nommez sa copie `MonteCarloCanvas.java`…