Université de Nice - Sophia Antipolis
Faculté des Sciences

DEUG MIAS MP1

Programmation 2001-02

5. PROGRAMMER AVEC L'AWT [1]

A. Le concept d'«!interface!» en Java

Une *interface*, c'est un peu comme une classe mais dont les méthodes restent «!abstraites!», l'interface ne fournit que leurs en-têtes. Considérons par exemple l'interface!`Comparable`, qui est fournie avec l'API de Java 2. Vérifiez avec un navigateur [*Menu Démarrer/Java/API*] qu'elle ne contient qu'une seule méthode. En réalité, il serait trivial de programmer cette interface puisqu'elle tient en deux lignes!:

```
public interface Comparable {
   public int compareTo (Object x);          // on ne donne que l'en-tête! !…
}
```

Une interface est un type, comme une classe. Mais un objet ne pourra être de type `Comparable` que s'il appartient à une classe qui *implémente* cette interface.

Une classe «!implémente une interface!» si elle définit concrètement toutes les méthodes de l'interface.

Par exemple!la classe `String`, telle qu'on la voit dans l'API, est déclarée ainsi!:

```
public final class String implements Comparable, Serializable
```

Puisque `String` implémente l'interface `Comparable`, elle promet de remplir son *contrat*!: définir concrètement la méthode d'instance `compareTo(Object x)`. Bien entendu, elle vérifiera si `x` est bien une instance de la classe `String` avant de faire la comparaison. Notez que `String` implémente une autre interface `Serializable` [grosso modo, un objet est «!sérialisable!» si on peut sauver son état sur le disque dur]. Une classe peut donc implémenter plusieurs interfaces, remplir plusieurs contrats.

Exercice 5.1 Vérifiez dans l'API que la classe `String` implémente bien l'interface `Comparable`. Comparez au toplevel la chaîne `"livre"` et la chaîne `"libres"` [la comparaison des chaînes se fait dans l'ordre lexicographique, celui d'un dictionnaire]. Traduisez en Java au toplevel la phrase suivante!:

Si la chaîne "livre" apparaît après la chaîne "libres" alors afficher "Ok"

N.B. L'intérêt majeur des interfaces en Java réside dans le fait qu'une classe peut implémenter plusieurs interfaces, mais elle ne pourra jamais «!étendre!» plusieurs classes!à la fois.

B. Les bibliothèques AWT et SWING

Vous avez déjà dessiné dans une fenêtre graphique avec la classe `Turtle` qui construisait la fenêtre pour vous. Nous allons nous dégager de cet automatisme en programmant directement avec 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 partiel de la dernière page 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` par exemple sait faire tout ce que sait faire un `TextComponent`, peut-être même un peu plus ou un peu différemment.

C. Construction d'une fenêtre

Nous allons commencer cette incursion dans l'API graphique de Java de manière très simple, en construisant une fenêtre AWT destinée à montrer le résultat d'une expérience statistique. Pour l'instant, limitons-nous à la fenêtre!!

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 réutiliser 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.*;                       // importer java.awt.Frame
class  MP1Frame  extends  Frame  {       // MP1Frame est une sous-classe de Frame
   MP1Frame(String title) {              // le constructeur
      super(title);                      // invocation du constructeur de Frame
      this.setTitle(title);              // et 3 messages à l'instance en construction
      this.setBackground(Color.lightGray); // couleur du fond
      this.setLocation(20,30);           // positionnement en (20,30)
   }
}
```

On fait du Java graphique avec une doc de l'API sous les yeux!: un livre[1] ou 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`.

• 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 ont un fond gris clair, 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 ton titre, change ta couleur de fond, 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 5.2 a) Allez au toplevel et créez un objet f de type `MP1Frame`, de titre `"Essai MP1Frame"`.

b) Demandez à voir la valeur de la variable `f`. Notez qu'elle est *hidden* [cachée] et *resizable* [redimensionnable]…

c) Demandez lui de se montrer!: «!`f`, montre-toi!!!». Comment dit-on «!montrer!» en anglais!?…

d) Redemandez la valeur de `f`. Qu'est devenu le *hidden*!?

e) Modifiez sa dimension à la souris avec le coin inférieur droit de la fenêtre.

f) Redemandez la valeur de `f`. Notez le changement de taille.

g) Changez le titre de la fenêtre en `"Nouveau titre"`. Ah, comment fait-on!? Cherchez dans la bonne classe de l'API… Le titre de la fenêtre a-t-il bien changé!? Que faire!?…

h) Essayez de fermer la fenêtre!! Si vous avez des problèmes!: `System.exit(0)` au toplevel [un «!Reset Interactions!» seul ne suffit pas]. Il nous faudra donc programmer la case de fermeture!qui est inactive par défaut!!!!

i) Ajoutez une méthode `main(…)` à la classe précédente, qui se contente de définir une fenêtre `f` et lui demande de se montrer. Puis rajoutez en dernière ligne du constructeur `MP1Frame(…)` 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 vérifiez que vous ne pouvez plus redimensionner la fenêtre… Nous opterons désormais pour des fenêtres non redimensionnables.

N.B. *a) Vous noterez que le «!`show`!» se fait dans le `main(…)`, pas dans le constructeur qui se contente de… construire.*
b) Comprenez-vous pourquoi il ne faut surtout pas de `System.exit(0)` à la fin du `main(…)`! ?

La philosophie des interfaces graphiques!: tout commence au moment où le `main(…)` termine!!
C'est alors que les composants vont vivre leur vie, à l'écoute des évènements…

D. Problème!technique : la case de fermeture

La case de fermeture n'est pas automatiquement activée. 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

[1] *L'une des références classiques est «!Java in a Nutshell!» chez O'Reilly, notamment le volume 2 [en français malgré son titre anglais].*

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 *écouteur* à la fenêtre, mécanisme recevant les évènements et acceptant de traiter certains d'entre eux. Comment faire ? Il se trouve que Java utilise pour cela 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 <u>rien</u> 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)! {}          // idem  <-----------------
   ... ... ... ...
   public void windowIconified(WindowEvent e) {}         // idem
   public void windowOpened(WindowEvent e) {}            // idem
}
```

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!:

i) Définir un *écouteur* comme instance d'une sous-classe anonyme de `WindowAdapter`[3] qui va redéfinir la méthode `windowClosing(…)`! pour qu'elle fasse quelque chose d'intéressant, ici terminer le programme [mais ce pourrait être autre chose].

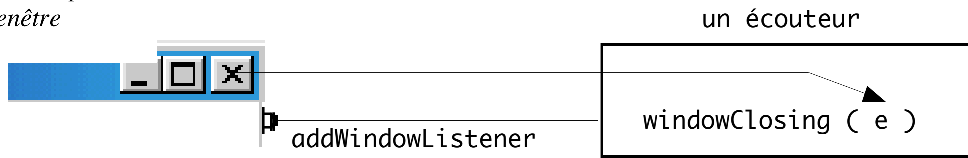
ii) Enregistrer cet écouteur [`addWindowListener`] auprès de la fenêtre en construction [`this`]!:

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

class  MP1Frame  extends  Frame  {
   MP1Frame(String title) {
      ... ... ...
      // 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 auprès de la fenêtre



<u>Explication</u>!: 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)`

Exercice 5.3 Vérifiez que maintenant la case de fermeture est active!! Ouf.

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

[3] *Si l'on choisit d'utiliser une sous-classe anonyme de WindowAdapter plutôt que de WindowListener, c'est bien entendu pour éviter d'avoir à implémenter les 7 méthodes!! On se contente de profiter des 7 implémentations triviales de WindowAdapter, en en redéfinissant une seule.*

E. Label!: un composant textuel pour afficher du texte

La classe `Label`[4] est une sous-classe de la classe `Component` [package `java.awt`]. Un objet de la classe `Label` sera donc un composant pouvant contenir une seule ligne de texte, pour afficher un message ou le résultat d'un calcul.

```
Label label = new Label("Ceci est un texte",Label.CENTER);
```

Les constantes `CENTER`, `RIGHT`, `LEFT` de la classe `Label` permettent de choisir l'*alignement* du texte. Ce texte n'est pas modifiable par l'utilisateur, seulement par le programme, avec la méthode d'instance `setText(…)`! :

```
label.setText("Et ceci est le nouveau texte");
```

Pour ajouter un composant à une fenêtre `f`, on utilise la méthode `add(…)` héritée de la classe `Container`! :

```
f.add(label);
```

Vous trouverez comme il se doit les détails croustillants de la classe `Label` dans l'API…

☞ Ah, encore une chose! Nos fenêtres vont en général recevoir plusieurs composants!: du texte, des boutons, des zones à dessiner, etc. Comment Java disposera-t-il ces composants!? De haut en bas, de gauche à droite, dans une grille!? En fait il va utiliser le *`LayoutManager`* [gestionnaire de disposition] que nous attacherons à la fenêtre. Le plus simple est **`FlowLayout()`** qui ajoute les composants les uns à la suite des autres [autrement dit, la disposition ne nous intéresse pas beaucoup]. Dans cette feuille, c'est celui que nous choisirons en exprimant dans le constructeur de la fenêtre!:

```
this.setLayout(new FlowLayout())! ;
```

Vous verrez peut-être ultérieurement le gestionnaire plus élaboré `BorderLayout` avec ses points cardinaux.

Exercice 5.4 Le but de cet exercice est d'écrire une minuscule interface graphique constituée d'une! `MP1Frame` contenant un `Label`. Il s'agit de calculer une approximation de π par une méthode de Monte-Carlo[5] et d'afficher le résultat non par un `System.out.println(…)` désormais démodé!☺ mais directement dans le `Label`…

a) Ouvrez dans DrJava une nouvelle classe `PiCalc` qui sera une sous-classe de `MP1Frame`, avec une variable privée `label` de type `Label`, un entier privé `nbExp` initialisé à `50000` représentant le nombre d'expériences, et un constructeur!:

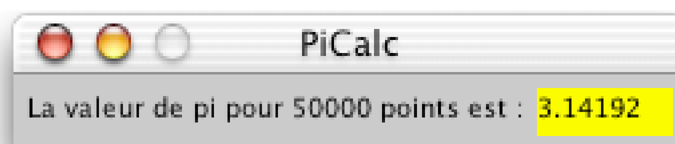
```
PiCalc(int x, int y)
```

Une instance de `PiCalc` sera une fenêtre de titre `"PiCalc"`, dont la position [en anglais *location*] du coin supérieur gauche sera aux coordonnées `(x,y)` lorsque la fenêtre se montrera. Dans ce contructeur, vous commencez par opter pour le gestionnaire simplifié de disposition. Puis vous initialisez la variable `label` à un texte centré vide sur fond [*background*] jaune que vous incorporez à la fenêtre en cours de construction. Une fois tous les composants ajoutés dans le constructeur, vous terminez par une instruction `this.pack()` demandant à la fenêtre de faire un joli paquet-cadeau et de calculer les tailles et emplacements de tous ses composants [avec l'aide du `LayoutManager`]. Le `main(…)` est identique à celui de `MP1Frame` [dont on ne se servira plus!!]. Exécutez!!

b) Il y a eu un problème, la fenêtre est toute petite, non redimensionnable, et il faudra peut-être même un `System.exit(0)` pour forcer sa fermeture!! Râté…Essayez d'initialiser le `label` non pas à la chaîne vide mais à une chaîne contenant suffisamment d'espaces…Ce n'est pas une méthode très «!propre!» mais ça ira!!

c) Dans une procédure `calc()`, calculez une valeur approchée de π de la manière suivante. Vous tirez au hasard `nbExp` fois deux réels x et y dans [-1!,+1] puis vous testez si $x^2+y^2<1$ c'est-à-dire si le point (x,y) pris au hasard dans le carré-unité est en fait tombé à l'intérieur du cercle-unité!! La probabilité de tomber dans le cercle étant égale au rapport des aires de ces deux figures, vous êtes capables d'en déduire une valeur approchée de π, n'est-ce pas!?… Vous faites donc ce calcul dans la méthode `calc()` puis vous faites afficher l'approximation trouvée de π dans le `label`. La méthode `calc()` est invoquée en dernière ligne du constructeur plutôt que dans le `main()` qui se contente de donner naissance à l'interface graphique…

d) Ajoutez un autre `label` devant le précédent, contenant un message pour que ce soit plus joli!:



N.B. Le «!look and feel!» d'une fenêtre AWT dépend du système d'exploitation [ici MacOS, pour lequel la case de fermeture est à gauche, alors qu'elle est à droite dans une fenêtre Windows].

---

[4] *Label signifie «!étiquette!» en anglais…*

[5] *Une!» méthode de Monte-Carlo!» est une simulation probabiliste basée sur un grand nombre d'expériences.*

F. TextField!: un composant textuel pour entrer du texte

Un `Label` était destiné à remplacer les `System.out.println(…)`. De même, un `TextField` va remplacer les `Console.readLine(…)` en permettant d'entrer une *chaîne de caractères* [`String`]. Réglons tout d'abord le point suivant!: et si l'on veut entrer un `nombre` ? Il se trouve qu'à chaque type primitif [`int`, `double`, `char`, `boolean`, etc] est associée une «!**classe enveloppante**!» [*wrapper class*] destinée à couvrir les cas où une méthode attend un objet et où l'on ne dispose que d'une donnée primitive. Par exemple, `Integer x = new Integer(2002)` sera l'objet de type `Integer` associé à `2002` qui est un `int`.

`int ⇆ Integer, double ⇆ Double, char ⇆ Char, boolean ⇆ Boolean, ...`

N.B. Notez la première lettre en majuscule, signe que l'on est en présence d'une classe!!

Exercice 5.5 Allez butiner la classe `Integer` dans l'API et!tâchez au toplevel!de!:
a) Récupérer dans une variable `n` de type `int` l'entier `2002` qui est caché dans le `x` défini ci-dessus.
b) Récupérer dans une variable `z` de type `int` l'entier `2002` à partir de la variable `String s = "2002"`. Autrement dit, savoir convertir une chaîne contenant un `int` en un véritable `int`. *Vous savez déjà faire la réciproque qui est triviale…*

Revenons à nos moutons, ou plutôt à nos `TextField`. Ce sont donc des zones de texte *éditable* [modifiable au clavier] dans lesquelles nous allons taper une suite de caractères, pour représenter un nom de famille, une marque de baskets, un nombre d'années ou le prix en euros d'une pizza aux fruits de mer.

Exercice 5.6 A l'aide de l'API, classe `TextField`, modifiez la classe `PiCalc` de la manière suivante.
a) Rajoutez une variable privée `textfield` de type `TextField`. Elle sera destinée à contenir une zone de texte dans laquelle l'utilisateur pourra entrer le nombre `nbExp` d'expériences.
b) Dans le constructeur, rajoutez ce `textfield` à une fenêtre `PiCalc`, par exemple après les deux labels. Arrangez-vous pour que la valeur affichée *par défaut* soit égale à `50000`. Exécutez pour visualiser le résultat. Vérifiez que la zone de texte est bien éditable, mais que le fait de l'éditer et d'appuyer sur Entrée ne provoque aucune action !

Oui, le `textfield` existe bien, mais n'est pas à l'écoute des modifications!! C'est parfaitement analogue à la situation rencontrée avec la case de fermeture. Il va falloir brancher un écouteur, de telle sorte que si l'on modifie le texte et que l'on appuie sur Entrée, la modification soit prise en compte et un message ad-hoc émis à qui de droit. Enfin, c'est ce qu'il faudrait faire dans un logiciel réel!! Car pour ne pas compliquer, et parce qu'il nous reste encore un composant à détailler, le *bouton à cliquer* [`Button`], ce <u>n'est pas</u> la stratégie qui sera employée. Nous allons laisser le `textfield` tranquille et il faudra presser le bouton «!Calculer!!!» pour valider la modification et lancer le calcul. Ok!?

G. Button!: le composant «!bouton à cliquer!»

La classe `java.awt.Button` offre le composant le plus célèbre!: le bouton à cliquer. Il contient un minuscule message "Ok", "Annuler", "D'accord", "Encore"… et attend un évènement «!clic!dans moi » pour provoquer une action.

Exercice 5.7 Ajoutez un bouton `"Calculer! !"` à droite des 3 autres composants. Vous ajouterez donc une variable privée `button` de type `Button`. Compilez, exécutez, vérifiez que le bouton est bien là mais ne réagit pas aux clics…

☞ Vous commencez à connaître la chanson!de l'API : le bouton ne sera à l'écoute des clics que si vous lui branchez un écouteur!! Nous avions évoqué au §D la notion d'***événement***!: en branchant un écouteur à la fenêtre [l'écouteur était une instance d'une sous-classe anonyme de `WindowAdapter`], l'événement `e` représentant le clic dans la case de fermeture était passé à la méthode `windowClosing(…)` que nous forcions à faire un `System.exit(0)`.

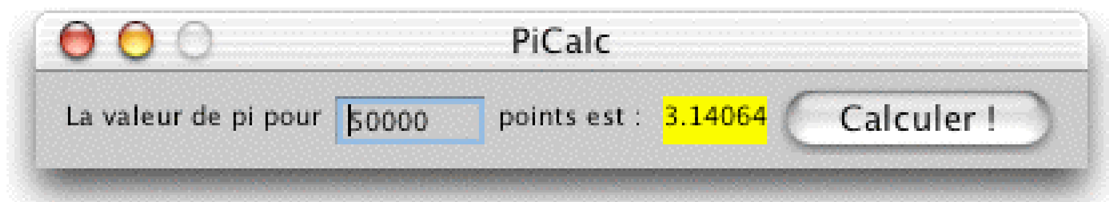
Et bien, nous reprenons presque la même idée. Lorsque vous cliquez sur un bouton pour provoquer une action, il y a production d'un événement de type `ActionEvent`. Il faut brancher un écouteur au bouton qui va se mettre à l'écoute des clics dans le bouton et réagir dès qu'un clic se produit. L'interface `ActionListener` du package `java.awt.event` ne contient qu'une seule méthode à implémenter!:

```
public interface ActionListener extends EventListener {
  public void actionPerformed(ActionEvent e);        // qu'il faudra donc implémenter dans une
}                                                    // sous-classe anonyme! !
```

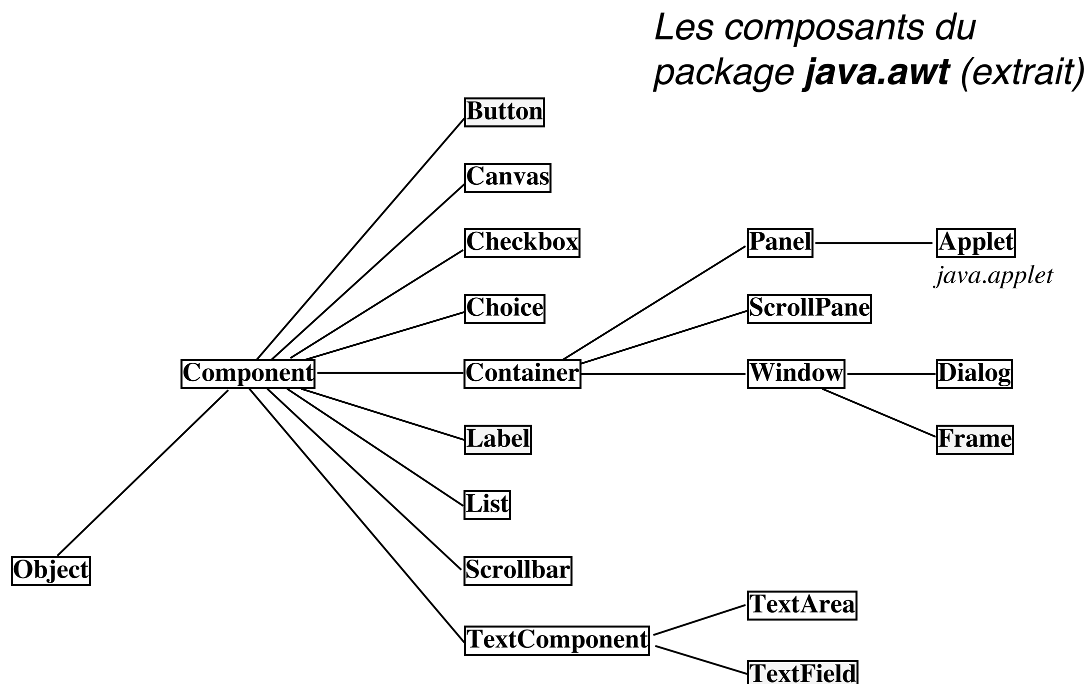
Exercice 5.8 En prenant exemple sur le code que nous avions écrit pour la case de fermeture et en vous aidant de l'API, branchez un écouteur au bouton [comment!? API → classe `Button`]. Lors d'un clic dans le bouton, la méthode `actionPerformed(…)` va lire le contenu du `textfield` [comment!? API → classe `TextField`], le convertir en `int`, mettre à jour la variable `nbExp` et passer la main à la méthode `calc()`. Simplissimo!! A vous de soigner les détails…

H. Mise en forme finale

Remodelez les composants pour que l'aspect soit un peu plus «!pro!» [mais oui, vous commencez à pouvoir produire avec Java de petits logiciels qui tiennent la route et sont diffusables sur le Net si vous avez une page Web…]. Par exemple!:



Il ne reste plus qu'à savoir dessiner dans une fenêtre en utilisant un `canvas`. Wait and see!!…



Le prochain contrôle [sur machine!? sur papier!?] proposera la construction d'une petite interface graphique. Ce thème sera bien entendu un thème essentiel de l'examen de JUIN.