Plan du cours Java Swing

- Chapitre 3 : principes d'une application Swing, dessin dans une fenêtre, applet.
- Chapitre 4 : composants de base, boutons, textes, menus. Containers généraux et afficheurs.
- Chapitre 5 : gestion des événements, actions, tables et listes.
- Chapitre 6 : boîtes de dialogues, architecture MVC, dessins et images , animation.

Chapitre 3 : Principes généraux

- Packages IHM: les JFC
- Architecture MVC
- Fenêtre, cadres et panneaux
- Composants de base et containers
- Principes d'une application Swing
- Dessiner et utiliser les composants de base
- Les applets

Les packages IHM de Java

java.awt : (awt : abstract window toolkit, java1.1) bas niveau. Anciens composants, dits composants lourds (ils sont opaques et affichés en dernier).

javax.swing : java 1.2. Composants légers = pas de code natif (écrits entièrement en java).

Swing étant construit par dessus awt, il vaut mieux connaître les deux.

J F C

Les classes Swing font partie d'un ensemble plus vaste de classes pour les IHM qui a été appelé les *Java Fondation Classes*. Les JFC contiennent : java.awt, et javax.swing.

Le package java.awt.geom pour dessiner du 2D, java.awt.dnd pour le *drag and drop*

javax.accessibility pour développer des applications adaptées aux personnes handicapées,

javax.swing.plaf pour modifier le *look and feel*, etc.

Java SE, Java EE et Java ME

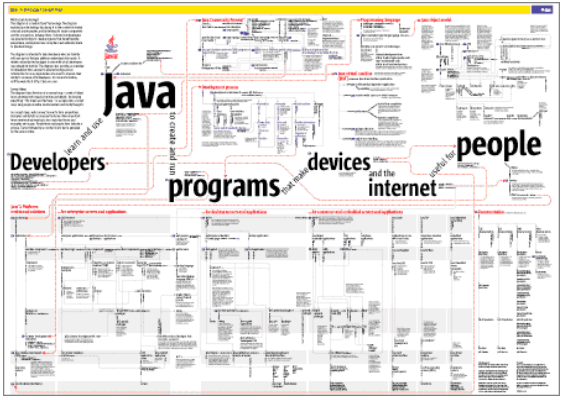
J2SE (Java 2 Standard Edition) a fourni le cadre Java général pour des applications devant tourner sur un poste de travail. On y trouve toutes les API de base, mais également toutes les API spécialisées dans le poste client (JFC et donc Swing, AWT et Java2D), ainsi que des API d'usage général comme JAXP (pour le parsing XML) et JDBC (pour la gestion des bases de données).

> J2SE a été renommé Java SE par Sun.

> J2EE (Java 2 Entreprise Edition) -> Java EE

> J2ME (Java 2 Micro Edition) -> Java ME

Technologies Java



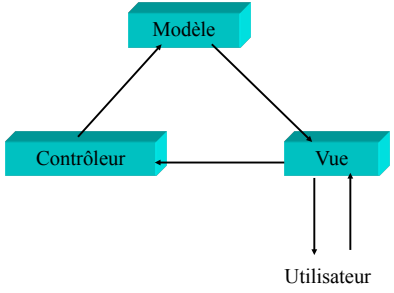
Rappel: Architecture MVC

La conception des classes Swing s'appuie assez librement sur l'architecture MVC (Modèle/Vue/Contrôleur). Cette architecture est apparue dans le contexte du langage Smalltalk (1980).

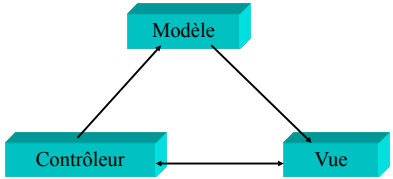
> On y distingue pour un composant:

- Le Modèle qui gère et stocke les données (abstraites).
- Une (ou des) Vue(s) qui implante(nt) une représentation (visuelle) à partir du modèle.
- Le Contrôleur qui se charge des interactions avec l'utilisateur et modifie le modèle (et la ou les vues).

Architecture MVC en Smalltalk

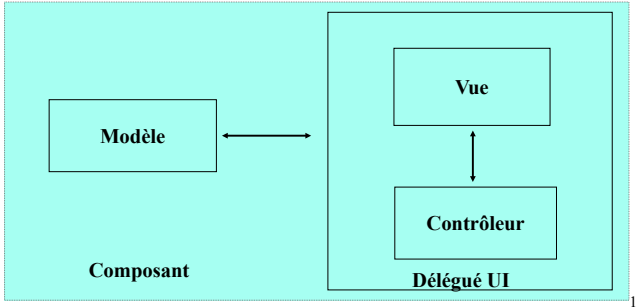


Architecture MVC



Dans certains cas, le contrôleur dépend fortement de la vue et cette architecture peut dégénérer en une architecture dite «Modèle Séparable» ou Modèle/Vue.

Architecture des composants Swing



Architecture des composants Swing

Les composants (légers) de Swing comprennent

- un modèle (Model)
- un délégué Interface Utilisateur (UI delegate)
- un composant qui étend JComponent.

Un modèle peut ainsi comporter plusieurs vues.

Un composant peut aussi avoir plusieurs types de modèles. (ex: JTable , JList, JScroolbar, JTree, etc. et même JButton. Pour instancier les JTree ou les JTable, on doit implémenter une interface TreeModel ou TableModel).

Architecture MVC

> Les Contrôleurs sont ici des écouteurs d'événements (cf. chapitre 6).

> Le délégué UI est responsable de l'apparence visuelle du composant et de ses interactions avec l'utilisateur.

Avec un délégué UI différent on obtient une nouvelle apparence, d'où la possibilité de s'adapter aux différents systèmes de fenêtrages (-> plaf = *plugable* look & feel).

Pour implanter ce type d'architecture, on utilise le patron classe/interface: Observable/Observer.

Observer/Observable

```
public class View implements Observer {

    public void update(Observable object, Object arg) {
          // méthode qui est déclenchée
          //  quand l'objet observable (le modèle) est modifié
    }
    // reste de la définition de la classe
}
public class Model extends Observable {
          // définition + appels à setChanged et NotifyObservers
}
```


classe Observable (ici le modèle)

> **addObserver(Observer o)**
> deleteObserver(Observer o)
> deleteObservers()
> **notifyObservers(Object arg)**
> **notifyObservers()**
> int countObservers()
> **protected setChanged()**
> boolean hasChanged()
> **protected clearChanged()**

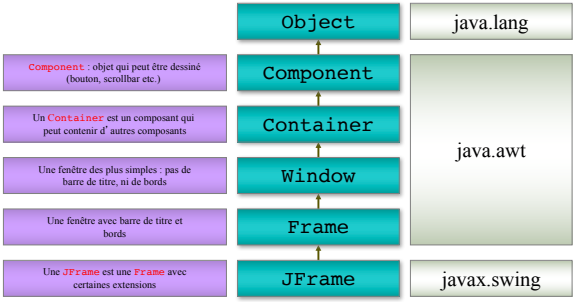
Cadres d'application ou fenêtres

Cadres JFrame et superclasses

Une fenêtre en java est représentée par la classe `Window` (package java.awt). Mais on utilise rarement `Window` directement car cette classe ne définit ni bords ni titre pour la fenêtre.

Dans une application Swing standard, on instanciera un cadre `JFrame` qui permet d'avoir une fenêtre principale avec un titre et une barre de menu.

JFrame et superclasses



JFrame et superclasses

> Un `Container` est un composant (`Component`) qui peut contenir d'autres composants.
> Puisque qu'un cadre `JFrame` hérite de `Container`, un `JFrame` peut contenir des composants.
> De même une barre de menus, qui est un `Jcomponent`, peut contenir des menus, qui eux-mêmes contiennent les items de menus, etc.

Fenêtres

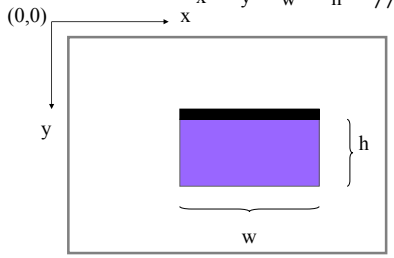
> La classe `Window` ajoute des méthodes propres aux fenêtres aux `Container` (comme le traitement de certains événements).
> Mais la fenêtre principale (cadre d'une application Swing) sera un objet `JFrame`.

JFrame: exemple

```
import javax.swing.*;
public class TryWindow {
    static JFrame myWindow = new JFrame
                        (« ceci est un titre »);
    public static void main (String[] args) {
       myWindow.setBounds(50,100,400,150); //Position, taille
   myWindow.setDefaultCloseOperation  // comportement
                        (JFrame.EXIT_ON_CLOSE);
            myWindow.setVisible(true);  // affichage
    }
}
```


JFrame: exemple

```
myWindow.setBounds(300,200,400,150);// Position
                   x   y   w   h   // et taille
```



JFrame: exemple

myWindow.setVisible(true);

`setVisible` affiche la fenêtre au premier plan, par dessus toute fenêtre déjà visible sur l'écran. `setBounds()` et `setVisible()` sont des méthodes de la classe `JFrame` héritées de `Component`. Elles sont donc disponibles sur tout composant.

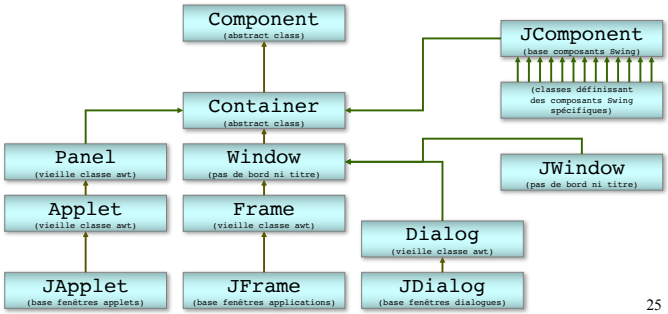
JFrame: exemple

myWindow.setDefaultCloseOperation
(JFrame.EXIT_ON_CLOSE);

`setDefaultCloseOperation` détermine ce qui se passe lorsque l'on ferme la fenêtre. Les autres valeurs possibles sont dans l'interface `WindowConstants` :

- `DISPOSE_ON_CLOSE` (détruit le `JFrame` et ses composants mais ne termine pas l'application)
- `DO_NOTHING_ON_CLOSE` (ne fait rien)
- `HIDE_ON_CLOSE` (cache la fenêtre en appelant `setVisible(false)`. C'est le défaut de `setDefaultCloseOperation())`

Composant et conteneur



Composant et conteneur

Toute classe dérivée de `Container` peut contenir des objets dérivés de `Component`; et comme `Container` dérive de `Component`, un `Container` peut contenir d'autres `Container`.

Exception: pour `Window` et ses sous-classes (cadres, dialogues ou fenêtres Swing) car une fenêtre ne peut être placée dans un Container.

JComponent

> C'est la classe de base de tous les composants Swing utilisés dans une fenêtre. Comme elle dérive de `Container`, les composants Swing sont tous des conteneurs.

> Ce sont tous des composants légers.

> Nous verrons plus loin les différents éléments qui héritent de cette classe.

Quels objets utiliser ?

> `JFrame` : le cadre principal d'une fenêtre d'application. Il peut contenir une barre de menus et d'autres composants. On l'utilise pour créer des classes de fenêtres spécifiques dans une application.

> `JDialog` : pour les fenêtres de dialogue avec l'utilisateur. Son «parent» (argument de ses constructeurs) sert essentiellement à placer la fenêtre sur l'écran (par défaut, au centre de ce parent).

Quels objets utiliser ?

> `JApplet` : classe de base des applets Java 2. Les applets sont des applications java pouvant tourner à l'intérieur d'un navigateur web.

> `JComponent` : les composants Swing : menus, boutons, cases à cocher, boutons radios, ascenseurs, etc. Leurs classes commencent toujours par J pour les différencier des composants awt.

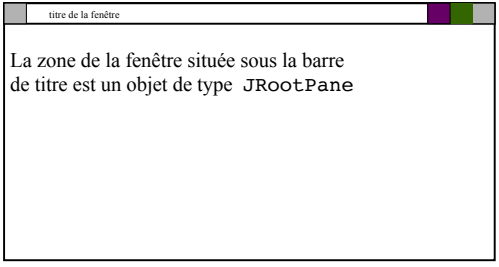
Conteneurs spécialisés: Les panneaux associés aux cadres

Panneaux

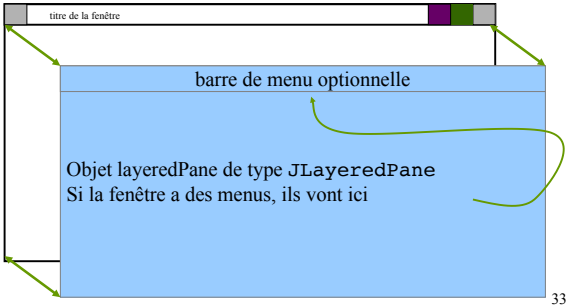
> Quand on veut ajouter des composants IHM ou dessiner dans la fenêtre d'un `JFrame`, on le fait dans un panneau (**WindowPane**) géré par l'objet `JFrame`. Même chose pour une applet.

> Il y a plusieurs panneaux dans un `JFrame` ou dans un `JApplet`.

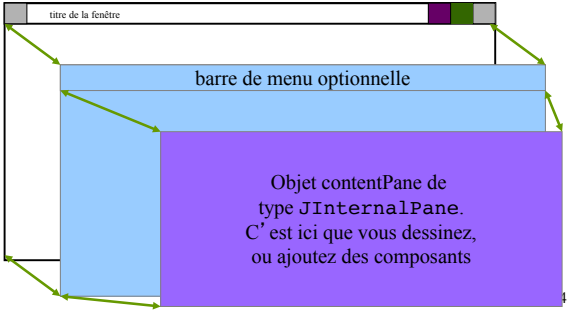
Panneaux d'un `JFrame`



Panneaux d'un `JFrame`



Le panneau principal (le contenu)



Le panneau vitré

> Il y a un autre panneau non dessiné dans le schéma précédent : le glassPane. Il est transparent et correspond à toute la zone du `JRootPane`.

> Tout ce qui est dessiné dans le glassPane apparaît au-dessus des autres dessins (utilisé pour des pop-up menus ou des animations).

Accès aux panneaux

> La classe `JFrame` contient les méthodes d'accès aux panneaux :

> `getRootPane()`

> `getLayeredPane()`

> **`getContentPane()`**

> `getGlassPane()`

> Remarque : toutes les classes de panneaux mentionnés ici sont des composants Swing.

Panneaux des `JApplet`

> Il y a aussi des panneaux dans les `JApplet` et ils fonctionnent de la même manière

> On pourra avoir une barre de menu dans la fenêtre d'affichage d'un `JApplet` et on ajoutera les composants à son panneau de contenu (`contentPane`).

Principes généraux d'une application Swing

Un exemple d'application

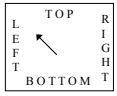
(Extrait des Java Tutorials de Sun)


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

public class SwingApplication {
   private static String labelPrefix = "Number of button clicks: ";
   private int numClicks = 0;

   public Component createComponents() {
      final JLabel label = new JLabel(labelPrefix + "0    ");
      JButton button = new JButton("I'm a Swing button!");
      button.setMnemonic(KeyEvent.VK_I);           // Alt + I
      button.addActionListener(new ActionListener() {
         public void actionPerformed(ActionEvent e) {
            numClicks++;
            label.setText(labelPrefix + numClicks);
         }
         }); label.setLabelFor(button); // le label décrit le bouton
```

```
/* Astuce pour mettre de l'espace entre un container
     * et son contenu : on met ici le contenu dans un JPanel
     * qui a une bordure vide mais non nulle.
     */
    JPanel pane = new JPanel();
    pane.setBorder(BorderFactory.createEmptyBorder(
                                // sens inverse des aiguilles d'une montre
                        30,  //top
                        30,  //left
                        10,  //bottom
                        30)  //right
                        );
    pane.setLayout(new GridLayout(0, 1)); // gère la disposition
    pane.add(button);                     // des éléments
    pane.add(label);                      // ajoutés ici en (ligne,colonne)
    return pane;
  } /* fin de createComponents */
```




```
    // Terminer la construction du JFrame et le faire apparaître

    frame.addWindowListener(new WindowAdapter() {
      public void windowClosing(WindowEvent e) {
        System.exit(0);
      }
    });
    frame.pack();      // la fenêtre prend sa taille
                       // en fonction de celles de ses composants
    frame.setVisible(true);
  }
}
```

```
public static void main(String[] args) {

    try {
      UIManager.setLookAndFeel(
        UIManager.getCrossPlatformLookAndFeelClassName());
    } catch (Exception e) { }

      //Creer une fenêtre principale grâce à JFrame
      //et lui ajouter un contenu
      // (les composants propres à l'application)

    JFrame frame = new JFrame("SwingApplication");
    SwingApplication app = new SwingApplication();
    Component contents = app.createComponents();
    frame.getContentPane().add(contents, BorderLayout.CENTER);
      // par défaut, le Layout Manager du contentPane d'un JFrame est un
      // BorderLayout qui dispose les éléments ajoutés le long des bords
      // ou au centre
```

Principes généraux

Pour la création des composants :

1. On crée d'abord le `JFrame`
2. Puis son contenu, construit ici dans un `JPanel`, qui sera ajouté au panneau de base du JFrame: le `contentPane`. (panneau de contenu).
3. Les événements sur les composants sont traités par des procédures appelées écouteurs (*listener*) d'événements.

Principes généraux

Pour organiser l'affichage de plusieurs composants, on utilise un conteneur (container):

1. On crée les composants primitifs.
2. On crée un container (ici le JPanel)
3. On définit la mise en page des composants dans ce container en lui associant un afficheur (*Layout Manager* = gestionnaire d'affichage).
4. On ajoute ensuite les composants primitifs au container.

Arbre des composants

> Quand on ajoute un composant 2 à un composant 1, le composant 1 est le parent du composant 2.

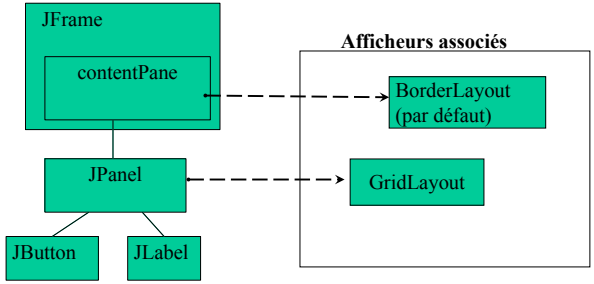
> Le parent contient aussi géométriquement les composants qui lui sont ajoutés.

> la méthode `getParent()` sur un composant retourne le parent (de type `Container`) qui le contient.

→ On a donc une arborescence de composants.

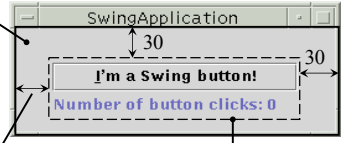
Arbre des composants

SwingApplication



Arbre des composants

Le `JPanel` est mis au centre du panneau du `JFrame` par le `BorderLayout`



Le `JPanel` a un bord (vide) d'une certaine taille (ici 30)

Les composants ajoutés au `JPanel` sont disposés sur une colonne par le `GridLayout`

Dessiner et utiliser les composants de base

Attributs des composants

> positions x et y, largeur width , hauteur height : tous de type `int`.
> le nom name : de type `String`.
> foreground et background : de type `Color`, définissent les couleurs de l'objet.
> font : la police utilisée pour les textes.
> cursor : l'apparence du pointeur de souris quand celui-ci est sur l'objet.

Attributs des composants

> enabled : indique si le composant est actif et accessible.
> visible : indique si l'objet est visible ou non sur l'écran.
[-> setEnabled(boolean) et setVisible(boolean)]
> valid. Seuls les composants valides peuvent être affichés à l'écran → utilisé pour 'désafficher' un composant. Un objet n'est pas valide si la disposition des entités qui le composent n'est pas encore déterminée.
[-> validated() et invalidate()]

Attributs des composants

> Tous ces attributs sont privés. On les modifie et on y accède via des accesseurs (méthodes souvent en `set` ou `get`).

> Exemples

```
myWindow.setName("ma fenêtre");
String theName = myWindow.getName();
```


Taille et position d'un composant

> `x` et `y` sont de type `int` mais certaines méthodes utilisent un objet de type `Point.`

> `width, height` sont également de type `int`, mais on pourra aussi utiliser un objet de type `Dimension.`

> tous ces attributs peuvent être réunis dans une même instance de la classe `Rectangle.`

Taille et position d'un composant

> void setBounds(int x, int y, int width, int height)

> void setBounds(Rectangle rect)

> Rectangle getBounds()

> void setSize(Dimension d)

> Dimension getSize()

> setLocation(int x, int y) [(Point p)]

> Point getLocation()

Taille et position d'un composant

Un composant possède une taille minimale, et si sa taille lui devient inférieure, il n'est plus affiché.

> pour tous les `Component` on a `getMinimumSize(), getMaximumSize()`

> `setMinimumSize` et `setMaximumSize` existent aussi pour les `JComponent.`

Environnement d'un composant: `getToolkit()`

- cette méthode retourne un objet de type `Toolkit` qui renseigne sur l'environnement dans lequel tourne l'application (ex : on récupère la taille de l'écran sur le `Toolkit` avec `getScreenSize()`.).
- permet par exemple de positionner une fenêtre au milieu de l'écran indépendamment de la taille de l'écran.
- à partir de l'objet Toolkit, on peut aussi par exemple charger une image [cf. getImage(URL)]

Caractéristiques visuelles

- La représentation visuelle d'un composant dépend :
  - des caractéristiques de sa classe au moment où il est instancié (background, etc.)
  - de ce qu'on dessine dans le composant
- On dessine dans un composant awt en implémentant sa méthode `paint(Graphics g)` qui est appelée automatiquement quand le composant doit être redessiné.
  (Attention: il ne faut pas dessiner ailleurs, ni dans le constructeur, ni dans le main …)

La classe `Graphics`

- Le paramètre `Graphics` de la méthode `paint` permet de définir les caractéristiques des dessins effectués dans le composant : épaisseur de trait, couleurs, zone atteinte par le dessin (clip), etc.
- Il fournit également des méthodes permettant de dessiner: par exemple des contours de formes (méthodes en `drawXXX`) ou des formes remplies (`fillXXX`).
- On peut aussi utiliser un objet de type `Graphics2D`.

Dessin dans un `JComponent`

Attention: le dessin dans un `JComponent` Swing est un peu plus compliqué que celui dans un composant awt parce que les composants Swing sont légers, ont des bords, et font du *double buffering*.

- La méthode `paint` d'un `JComponent` invoque `paintComponent, paintBorder` et `paintChildren`. Si le composant a un délégué UI, `paintComponent` invoquera sa méthode `update`.

Dessin dans un `JComponent`

IMPORTANT:

Ainsi, pour préserver un bon affichage des bords et des enfants, on dessinera dans un `JComponent` en redéfinissant sa méthode `paintComponent(g)` et en invoquant la méthode de son parent, i.e. `super.paintComponent(g)` au début de cette méthode.

(sinon, il peut se produire n'importe quoi!)

Caractéristiques graphiques

> void setBackground(Color)
> Color getBackground()
> void setForeground(Color): couleur pour tout ce qui apparaît sur le composant (ex. couleur du texte pour un bouton, couleur des traits du dessin de formes)
> Color getForeground()
> Void setXORMode(Color alter)
> void setCursor(Cursor)
> void setFont(Font)
> Font getFont()

Classe `Color`

> On peut définir la couleur par niveaux d'intensité de Rouge, Vert, Bleu (RGB).
  - `Color monBlanc = new Color(255,255,255);`
> Il y a aussi des couleurs nommées prédéfinies dans la classe `Color` : white, red, pink, etc.
  - `aWindow.setBackground(Color.pink);`
> java.awt définit la sous-classe de `Color` , `SystemColor.` On y trouve des couleurs standards permettant d'uniformiser les interfaces aux différentes applications : `Window, WindowText` etc.

Classe `Cursor`

> Elle contient des constantes (`final static`) pour des curseurs standards : `DEFAULT_CURSOR, CROSSHAIR_CURSOR,` etc.

> Cursor monCurseur = new Cursor
(Cursor.TEXT_CURSOR);

Classe `Font`

Classe assez complexe. Une fonte (ou police de caractères) peut être vue comme une collection de "glyphes".

> Distinction entre un caractère et un glyphe (= forme définissant l'aspect).

Ex: Les caractères non latins peuvent nécessiter plusieurs glyphes pour l'affichage d'un seul caractère.

Classe `Font`

Une police de caractères peut avoir plusieurs variantes, comme `heavy`, `medium`, `oblique`, `gothic`, ou `regular`. Les variantes d'une police ont des traits typographiques similaires et peuvent être reconnues comme faisant partie d'une même famille.

Finalement, une collection de glyphs d'un style particulier forme une variante de fonte; une collection de variantes forme une famille de fontes ; et une collection de famille de fontes forme l'ensemble des fontes disponibles sur le système.

Classe `Font`

- Un objet `Font` contient une table qui associe à toute valeur de caractère unicode le (ou les) code(s) des glyphes correspondants.
- Création d'une fonte :

```
Font maFonte = new Font(«Serif»,
                        Font.ITALIC, 12);
```

`«Serif»` : nom de police "logique".

`Font.ITALIC`: style. Il y a aussi `Font.BOLD`, `Font.PLAIN`.

Si on veut de l'italique en gras,

```
Font.ITALIC + Font.BOLD
```

Classe `Font`

- Il existe des polices incluses dans java appelées polices logiques :
  - ✓ Serif (TimesRoman)
  - ✓ SansSerif (Helvetica)
  - ✓ Dialog
  - ✓ Monospaced (Courier)
  - ✓ Symbol
- On peut aussi utiliser un nom de police, pourvu que cette police soit disponible sur le système.

Obtenir les fontes

On peut obtenir une table des noms de fontes :

```
GraphicsEnvironment e =
    GraphicsEnvironment.getLocalGraphicsEnvironment();

String[] fontnames =
                e.getAvailableFontFamilyNames();
```

```
Albertus
Albertus Extra Bold
Albertus Medium
Arial
Arial Alternative
etc…
```


Obtenir les fontes

ou encore directement un tableau de toutes les fontes disponibles sur le système. Mais chaque police du tableau est de taille 1 (en point).

```
GraphicsEnvironment e =
    GraphicsEnvironment.getLocalGraphicsEnvironment();
Font[] fonts = e.getAllFonts();
```

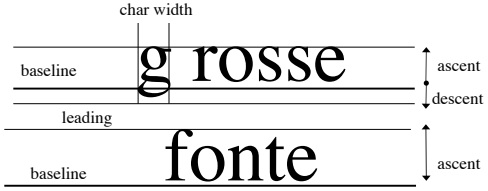
méthode `deriveFont()`

1 point = 1/72 pouce. Il faut donc en général changer la taille: pour cela, utiliser la méthode `deriveFont()`.

Exemple: obtenir la version 12 points d'une fonte récupérée dans un tableau,

```
Font maFonte =
        fonts[4].deriveFont(12.0f);
```


Métrique des fontes



height = ascent + descent + leading

Métrique des fontes

Tout `Component` possède une méthode `getFontMetrics()`.

- Cette méthode retourne l'objet de type `FontMetrics` associé à la fonte.
- Sur cet objet, on peut utiliser `getAscent()`, `getDescent()`, `getLeading()`, `getHeight()`, `charWidth(c)`, `charsWidth(char[],int,int)`, `stringWidth(String)`, etc.

Applet java
(étendre `Applet` ou `JApplet`)

S'exécute sur une machine virtuelle (celle du navigateur Web) et nécessite un fichier html.

> n'a pas de fonction `main` mais vit un cycle

> Les principales méthodes :

- ✓ **`init`**(): quand la page html est chargée la 1ère fois.
- ✓ **`start`**(): après init et chaque fois que l'on revient sur la page.(Ex. lancer un thread d'animation)
- ✓ **`stop`**(): quand on quitte la page html. (On interrompt les méthodes lancées dans start.)
- ✓ **`destroy`**(): quand l'applet est terminée. (On libère toutes les ressources.)

Exemple d'applet

```
import java.awt.*;
import java.applet.*;        // vieille applet (mais plus portable)

public class HelloWorldApplet extends Applet {
    Font f = new Font («Serif», Font.BOLD, 36);

    Public void paint (Graphics g)
    {
        g.setFont (f);
        g.setColor (Color.red);
        g.drawString (getParameter(«Chaine»),10,30);
     }
}
```


Fichier html associé

```
<HTML>
<TITLE> Applet HelloWorldApplet </TITLE>
<APPLET CODE=«HelloWorldApplet.class»
          WIDTH = 200    HEIGHT = 300 >
<PARAM    NAME=«Chaine»
          VALUE=«Hello World»   >
</APPLET>

</HTML>
```


La balise <APPLET>

```
< APPLET
          [CODEBASE = codebaseURL]
          CODE = FichierClassApplet
          [ARCHIVE = FichiersJar]
          [ALT = TexteAlternatif]
          [NAME = NomInstanceApplet]
          WIDTH = pixels          HEIGHT = pixels
          [ALIGN = alignement]
          [VSPACE = pixels]
          [HSPACE = pixels]                >
 [< PARAM NAME = appletParam1 VALUE = value >]
 [< PARAM NAME = appletParam2 VALUE = value >]
          . . .
  [Texte HTML alternatif]
  </APPLET>
```

Applet java: sécurité

> Une applet ne peut charger de librairies ou utiliser de méthodes natives.
> Elle ne peut normalement pas lire ou écrire dans des fichiers de l'hôte sur lequel elle s'exécute.
> Elle ne peut effectuer de connections réseaux sauf sur l'hôte dont elle provient.
> Elle ne peut lancer aucun programme de l'hôte sur lequel elle s'exécute.
> Elle ne peut lire certaines propriétés système.

--> ces restrictions peuvent être levées par un certificat de sécurité (cf. `jarsigner` , `policytool`).

Applet java

On a vu que les `JApplet` fonctionnaient comme les cadres `JFrame` du point de vue des panneaux. Pour ajouter des composants à un objet `JApplet`, on ajoute ceux-ci au panneau de contenu (`contentPane`) du `JApplet`. Cette opération est en général exécutée dans la procédure `init`() de l'applet.

De même, la gestion des événements (que l'on étudiera au cours suivant), est exactement la même pour les applets que pour les applications.