

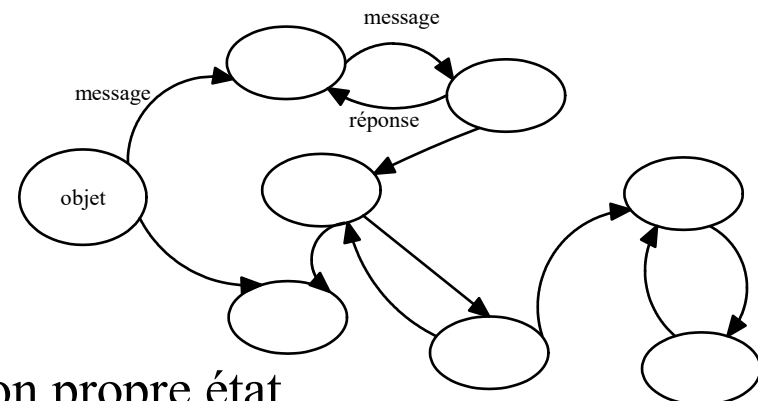


# Programmation répartie en Java

- Interfaces (SWING)
- Interactivité (événements)
- Ressources sur internet (URL, HTTP, web services)
- Persistance (sérialisation)
- Communication entre programmes (flots, sérialisation, sockets)
- Parallélisme (threads)

# Programmation Orientée Objets (rappels)

- ❖ Objets (classes).
- ❖ État (propriétés).
- ❖ Savoir faire (méthodes).
- ❖ Messages (appel de méthodes).
- ❖ Héritage.
- ❖ Polymorphisme.



- ❖ Objets interagissant, chacun ayant son propre état.
- ❖ Les objets du programme interagissent en s'envoyant des messages les uns aux autres (appels de méthodes).

# Domaines d'utilisation de java

- Applications interactives multi plateformes
- Applications pouvant être exécutés dans un navigateur Web (Applets)
- Accès à des services Web
- Applications côté serveur (Servlets)
- Mobiles (Android, ...)
- Embarqué (lecteurs Blu-ray, javacards ...)

# Caractéristiques de Java

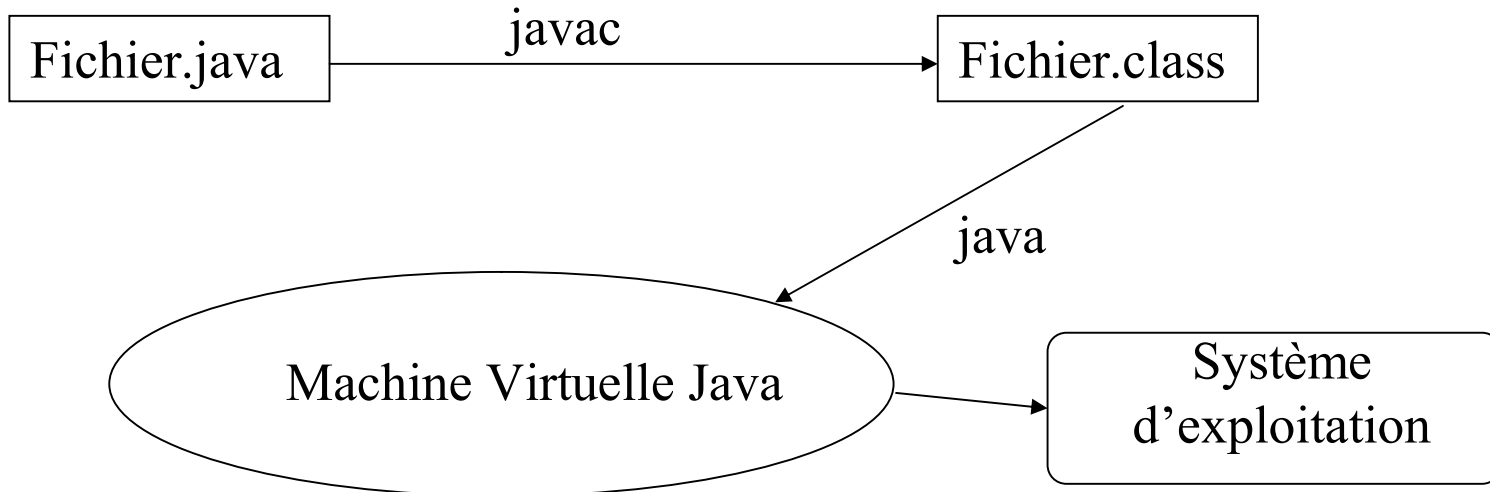
- ❖ Langage Orienté Objets général.
- ❖ Syntaxe inspirée du C/C++.
- ❖ Langage multi plateforme
- ❖ Langage interprété
- ❖ Gestion de la mémoire dynamique (sans pointeurs)
- ❖ API distribuée très vaste.
- ❖ Multitâche (threads)
- ❖ Gestion des exceptions
- ❖ Interfaces, événements, images, sons ...
- ❖ Introspection
- ❖ Gratuit (<http://www.oracle.com/technetwork/java/index.html>)

# Java et les réseaux

- ❖ Langage portable (versions spécialisées pour téléphones, capteurs, ...)
- ❖ HTTP
  - ❖ Possibilité d'écrire des applications sur le client (Applets)
  - ❖ Possibilité d'écrire des applications sur le serveur (Servlets)
  - ❖ Possibilité d'accès à des Web Services
- ❖ Accès à des ressources (URL)
- ❖ Communication C/S (Sockets)
- ❖ Sérialisation d'objets (état)
- ❖ Chargement dynamique de code (ClassLoader) + introspection
- ❖ Appel de méthodes distantes ( RMI)
- ❖ Sécurité (limitations + signatures)

# Compilation et exécution

- Les fichiers sources en Java se terminent par l'extension ".**java**". Le nom du fichier est le même que celui de la classe (pas d'édition de liens).
- Le compilateur Java transforme le code source Java en **byte-code**.
- la **Machine Virtuelle Java** exécute (interprète) ce byte-code en utilisant le système d'exploitation de la machine.

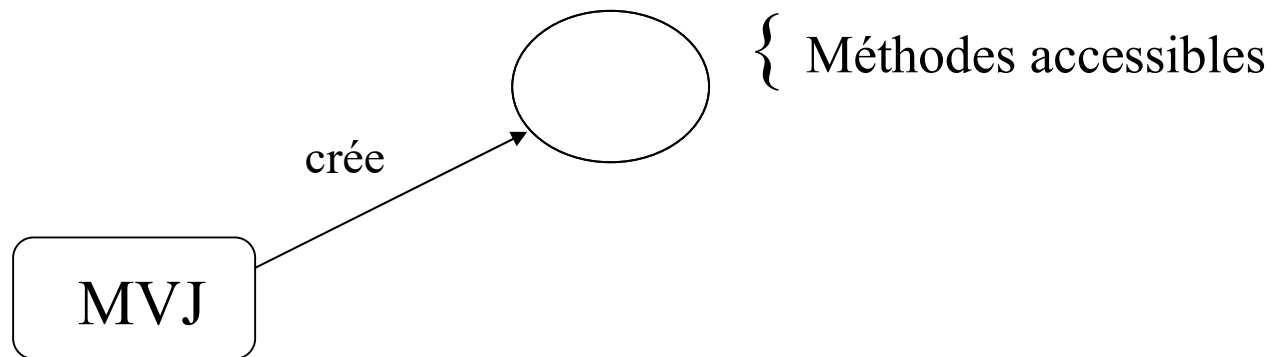


# Syntaxe de java

- **Types primitifs** : Java utilise cinq types primitifs: entiers, réels en virgule flottante, booléens, caractères et chaînes.
  - Entiers : **byte** , **short** , **int** , **long**
  - Réels en virgule flottante : **float** , **double**
  - Booléens : **boolean** (true , false)
  - Caractères : **char** ( ' a ' )
  - Chaînes : "ceci est une chaîne"
- **Tableaux** : En Java on peut déclarer des tableaux de tous types:
  - char** s[]; **int** i[];
  - Et des tableaux d'objets : Image diaporama [];
  - Ainsi que des tableaux de tableaux: **int** table[][];
  - En Java un tableau est un objet. Il a une méthode: **length** qui permet de connaître la taille du tableau.
  - Pour créer un tableau vide en Java : **int** liste[] = **new int**[50];
  - Les tableaux en Java sont **statiques**. Pour des tableaux **dynamiques** on utilise la classe **Vector**.

# Exécution d'un programme

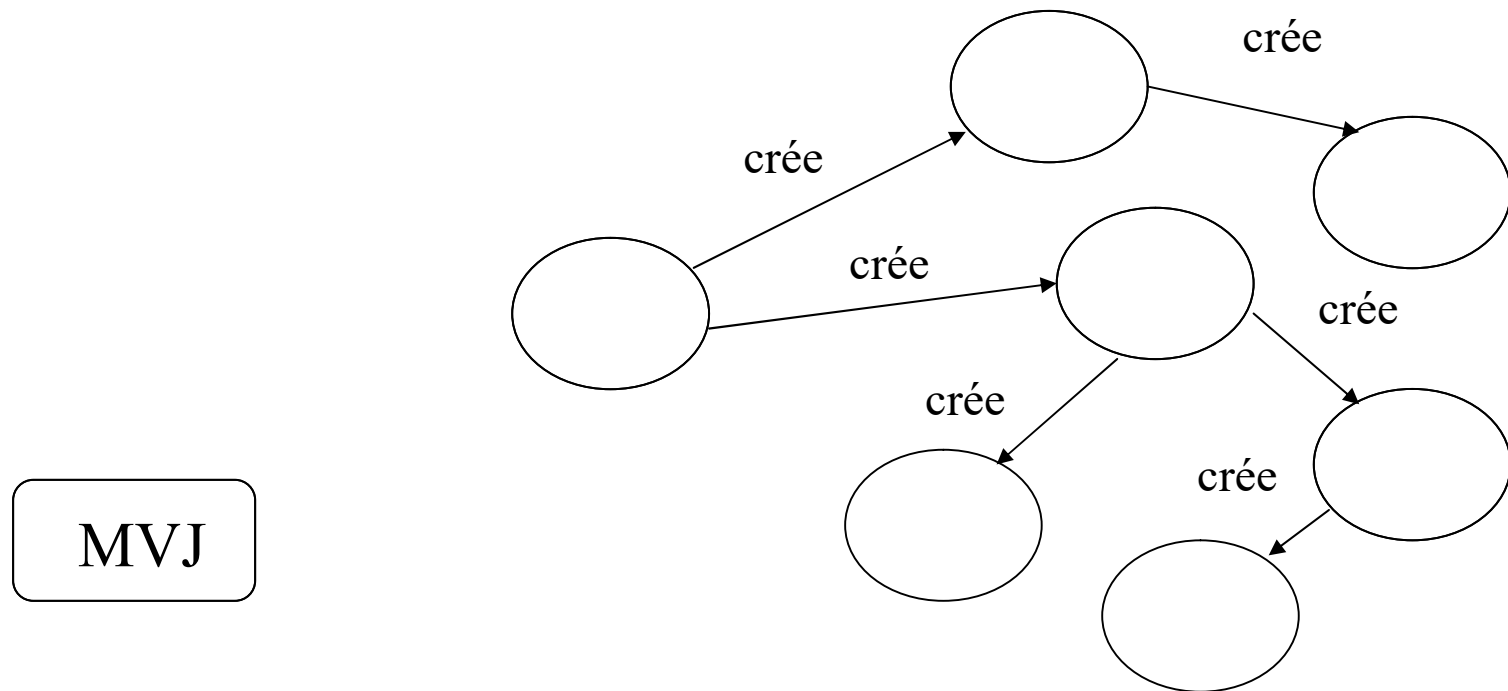
- Un premier objet est créé par l'environnement d'exécution (MVJ ou plateforme)
- Il doit posséder une ou plusieurs méthodes que cet environnement peut appeler





# Exécution d'un programme

- Cet objet crée d'autres objets
- Eux-mêmes créent d'autres objets ...



# Exécution d'un programme

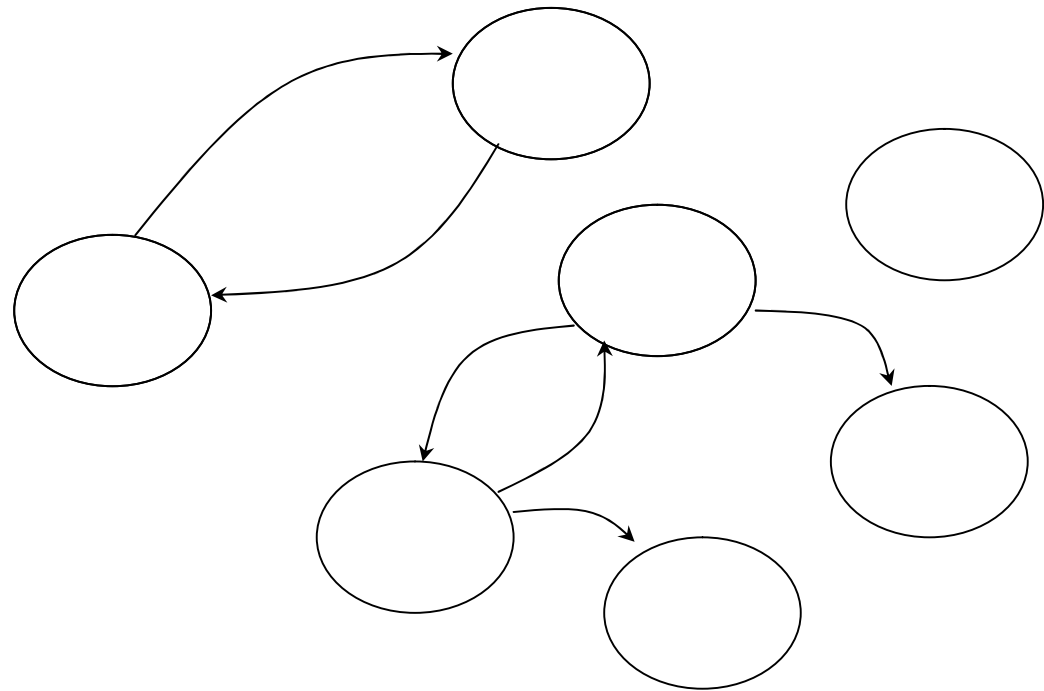
- Certains objets disparaissent en cours d'exécution
- La MVJ récupère la place en mémoire



# Exécution d'un programme

- Les objets interagissent par des messages
  - Appel de méthode + paramètres
  - Retour de méthode

MVJ



# Problème de visibilité des objets

- Un objet *a* crée un objet *b* qui contient un objet *c* :  
*a* peut envoyer des messages à *b* mais pas à *c* (il n'y a pas accès car il est interne à *b*)
  - Il faut que *b* propose des méthodes d'utilisation de *c*
- Un objet *a* crée un objet *b* qui crée un objet *c* :  
*a* peut utiliser *b* mais pas *c* (il ne le connaît pas)
  - **Solution directe** : Il faut que *b* propose une méthode d'accès à *c* (getter)
  - OU
  - **Solution de type service** : Il faut qu'il existe un objet *x* qui connaît *c* et soit connu de *a*. Cet objet offre une méthode d'accès à *c*.*c* est un **service** qui **s'enregistre** auprès de *x* et *a* **utilise ce service** en se servant de *x* comme annuaire.

# Programmation dirigée par les événements

- **Programme traditionnel** : (scénario)
  - Commence à s'exécuter et se termine
- **Programme dirigé par les événements** : (réactif aux événements)
  - Ne s'exécute pas du début à la fin
  - Répond aux messages que lui envoie l'environnement
  - Ces messages sont envoyés quand se produisent des événements
    - Souris, clavier
    - Temps
    - Fenêtre
    - Réseau
    - Internes (levés par des objets)
    - ...
- Dans un programme traditionnel tout est prévu, dans un programme dirigé par les événements **tout dépend de ce que fait l'utilisateur.**

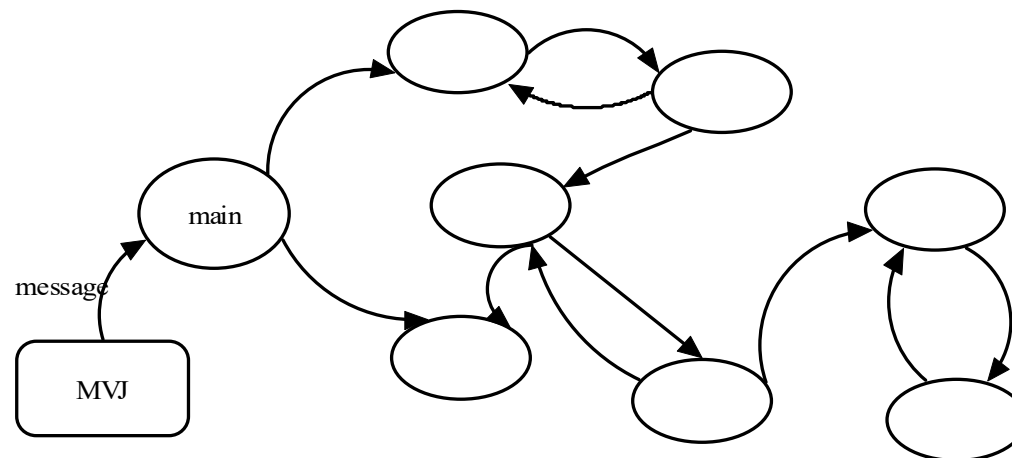
# Exécution d'un programme

- Cas d'un programme autonome de type scénario : la machine virtuelle appelle la méthode **main**.

```
static public void main(String argv[])
```

## Un seul objet ayant cette méthode doit exister

- Le scénario est contenu dans cette méthode



# Programme de type scénario

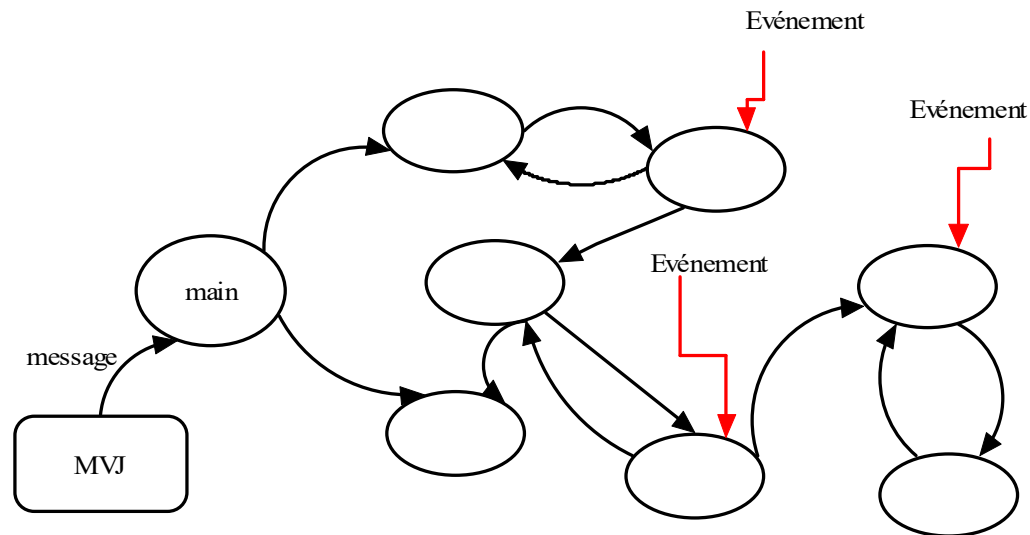
# Exécution d'un programme

- Cas d'un programme autonome de type événementiel : la machine virtuelle appelle la méthode **main**.

`static public void main(String argv[])`

Un seul objet ayant cette méthode doit exister. La méthode **main** met en place l'architecture de l'application.

- Certains des objets créés réagissent aux événements

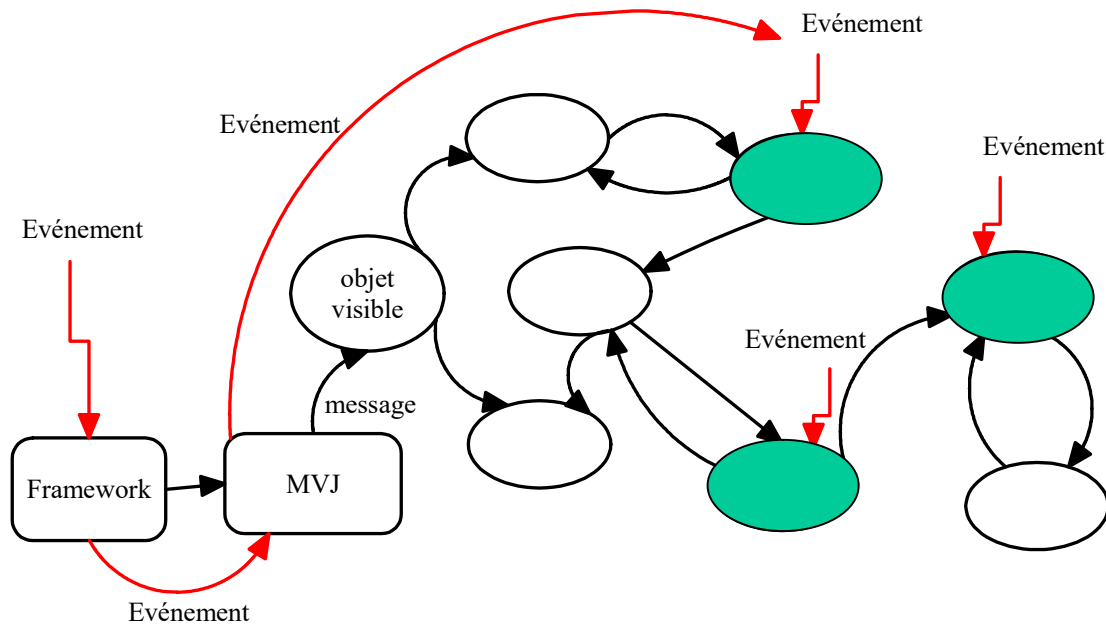


Programme  
de type  
événementiel

# Exécution d'un programme

Cas d'une **application contrôlée** : s'exécute sous le contrôle d'un **framework** (navigateur pour une applet / android pour une app). Le framework appelle, selon ses besoins, des méthodes de l'un des objets de l'application.

- Certains objets réagissent aux événements

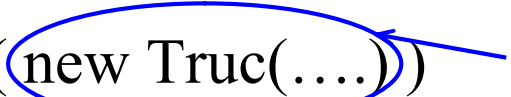




# Utiliser des objets

- Un objet est la matérialisation (**instanciation**) d'une **classe**.
- Les objets se manipulent avec des **références** (leur nom).  
`Point p; // définit p comme nom pour un objet de classe Point`  
`// (aucun objet n'est créé)`
- Les objets se créent (sont implantée en mémoire) avec l'opérateur **new**.  
`p = new Point();`  
ou `p = new Point(5,25);`
- L'accès aux éléments d'un objet (**propriétés** ou **méthodes**) se fait par l'opérateur **.** (point) : `nom_référence.propriété`  
`p.mettreEn(5,20);`  
`p1.distanceA(p2);`
- Java n'a pas de pointeurs (seulement des références)
- L'opérateur **instanceof** permet de savoir si un objet est d'une classe donnée :  
`if (p instanceof Point)` est Vrai si p est de classe Point ou d'une classe héritant de Point

# Objets et noms d'objets

- Les objets peuvent avoir **un ou plusieurs noms**. Ils peuvent aussi **ne pas avoir de nom**.
- Si  $t$  est un type primitif :
  - $m = t$  fait une copie de  $t$  dans  $m$ .
  - $a == x$  est vrai si les contenus de  $a$  et de  $x$  sont égaux
- Si  $t$  est un objet :
  - $m = t$  définit  $m$  comme **un autre nom de  $t$**  (mais ne fait pas une copie :  $m$  et  $t$  désignent **le même objet**).
  - $a == x$  est vrai si  $a$  et  $x$  sont 2 noms du même objet
  - $a.comparer(\text{new Truc}(\dots))$  

Objet de classe Truc sans nom

# Bibliothèques

- Bibliothèques de Java :
  - **java.applet**      **applets et sons**
  - **java.awt**      **graphique et interfaces**
  - **java.awt.event**      **événements d'interface**
  - **java.io**      **flux et fichiers**
  - **java.math**      **calculs**
  - **java.net**      **réseau et URLs**
  - **java.util**      **collections ...**
  - **javax.swing**      **interfaces**
  - ...

- Charger une bibliothèque :  
**import javax.swing.\*;**

```
import javax.swing.*;
```

```
class MonPremierProgramme {  
    static public void main(String argv[]) {  
        .....  
    }  
}
```

# Classes utiles de Java

Documentation en ligne :

(<https://docs.oracle.com/javase/x/docs/api/>)  $x = n^{\circ}$  de version

Il existe des classes correspondant à chacun des types primitifs :

- Byte
- Short
- Long
- Integer
- Float
- Double
- Boolean
- Character

## Exemple : la classe **Integer**

- Java - M. Dalmau - IUT de Bayonne

# Classes utiles de Java

## La classe **String**

- `String s = new String("Ceci est une chaîne de caractères")` Création
- `String concat(String) : String x = s.concat(new String("par exemple"))`  
On peut aussi utiliser `+` : `String x = s + "par exemple"` Concaténation
- `int length()` Taille
- `boolean equals(String celleQueLOnCompare)` Comparaisons
- `boolean equalsIgnoreCase(String celleQueLOnCompare)`
- `char charAt(int index)` Recherches
- `int indexOf(char ceQueLOnCherche)`
- `int indexOf(String ceQueLOnCherche)`
- `String substring(int debut, int fin)` Extraction
- `String toLowerCase()` Transformations
- `String toUpperCase()`

# Classes utiles de Java

## La classe String : conversions

- String → type primitif
  - Integer.parseInt(String) retourne un entier
  - Long.parseLong(String) retourne un entier long
  - Byte.parseByte(String) retourne un octet
  - Short.parseShort(String) retourne un entier court
  - Float.parseFloat(String) retourne un réel
  - ...

Peuvent lever une exception de classe **NumberFormatException**.

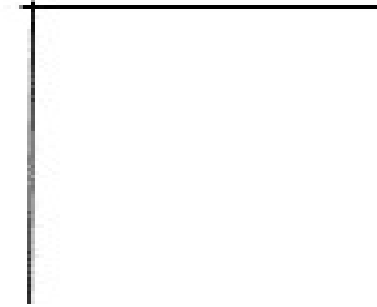
- Type primitif → String
  - valueOf(type primitif ) retourne une chaîne (String)
- String → tableau de caractères
  - toCharArray() retourne un tableau de caractères

# Classes utiles de Java

## La classe **Graphics**

**setColor**(Color couleurDeDessin)  
**drawLine**(int x1, int y1, int x2, int y2)  
**drawRect**(int x, int y, int largeur, int hauteur)  
**fillRect**(int x, int y, int largeur, int hauteur)  
**clearRect**(int x, int y, int largeur, int hauteur )  
**drawRoundRect**(int x, int y, int largeur, int hauteur, int largeurArc, int hauteurArc )  
**fillRoundRect**(int x, int y, int largeur, int hauteur, int largeurArc, int hauteurArc )  
**draw3DRect**(int x, int y, int largeur, int hauteur, boolean elevation )  
**fill3DRect**(int x, int y, int largeur, int hauteur, boolean elevation )  
**drawOval**(int x, int y, int largeur, int hauteur )  
**fillOval**(int x, int y, int largeur, int hauteur )  
**drawArc**(int x, int y, int largeur, int hauteur, int angleInicial, int angleArc )  
**fillArc**(int x, int y, int largeur, int hauteur, int angleInicial, int angleArc )  
**drawPolygon**(int[] pointsX, int[] pointsY, int nbrPoints )  
**fillPolygon**(int[] pointsX, int[] pointsY, int nbrPoints )  
**drawString**(String chaine, int x, int y )  
**copyArea**(int xOrigine, int yOrigine, int largeur, int hauteur, int xDest, int yDest)

0,0





# Classes utiles de Java

## La classe **Color**

- instances statiques de Color : Color.blue, Color.yellow, etc.
- construire un objet Color : new Color(int r, int v, int b);
- obtenir les composantes de rouge, vert et bleu : int getRed() ...
- brighter() y darker()

## La classe **Font**

- construire une fonte :  
maFonte = new Font("Serif", Font.BOLD | Font.ITALIC, 12)  
styles : **Font.BOLD**, **Font.ITALIC**, **Font.PLAIN**

## La classe **System**

- System.out.println("message" ) // affiche un texte à l'écran
- System.out.println(objet) // affiche le contenu de l'objet à l'écran

# Les images avec Java

## La classe **Image**

- dimensions :

- `int getHeight(ImageObserver)` // observateur (la fenêtre)
  - `int getWidth(ImageObserver)` // observateur (la fenêtre)

Peu utile

- copies :

- `Image getScaledInstance(int, int, int)` // hauteur ou -1, largeur ou -1 et mode

- `Image.SCALE_DEFAULT`

- `Image.SCALE_FAST`

- `Image.SCALE_SMOOTH`

- `Image.SCALE_REPLICATE`

- `Image.SCALE_AVERAGING`

} mode

- méthode de la classe **Graphics** :

- `drawImage(Image, int, int, ImageObserver)` // image, coordonnées du coin supérieur gauche et observateur (en général la fenêtre où on affiche)

# Les images avec Java

## La classe **ImageIcon** (SWING)

Permet de définir des icônes que l'on peut ensuite placer dans divers composants.

- création :
  - ImageIcon(String) // depuis un fichier
  - ImageIcon(URL) // depuis une URL
  - ImageIcon(Image) // depuis une image
- dimensions :
  - int getIconWidth()
  - int getIconHeight()
- affichages :
  - Image getImage()
  - void paintIcon(Component,Graphics,int,int)

# Les sons avec Java

## La classe **AudioClip**

Les sons sont des objets de la classe **AudioClip**

- Création
  - `objetSon = Applet.newAudioClip(URL);`
- Jouer le son
  - `objetSon.play();`
  - `objetSon.loop();`
- Arrêter le son
  - `objetSon.stop();`

Support pour .wav seulement dans l'API standard

# Les sons avec Java (autre solution)

## Les classes **AudioSystem** et **Clip**

- Créer le clip

```
Clip monClip = AudioSystem.getClip();  
AudioInputStream ligne = AudioSystem.getAudioInputStream(new File("nom"));  
monClip.open(ligne);
```

- Jouer le son
  - monClip.**start()**;
- Arrêter le son
  - monClip.**stop()**;

# Classes – création

- **Une classe contient:**
  - Des **propriétés** de types primitifs ou des références (noms d'objets).
  - Des **méthodes**
- **Déclaration d'une classe :**

```
[public] class nom_de_classe {  
    déclarations_de_propriétés;  
    déclarations_de_méthodes;  
}
```
- **Définition d'une propriété :**

```
[modificateur d'accès] Classe_de_la_propriété nom_de_la_propriété
```
- **Définition d'une méthode :**

```
[modificateur d'accès] Classe_valeur_retour nom_de_méthode (liste_arguments) {  
    bloc_de_code;  
}
```

Si une méthode ne retourne rien Classe\_valeur\_retour vaut **void**

Si une méthode n'a pas d'arguments on met **()**.

# Modificateurs d'accès

- ❖ **public** – Tout le monde peut accéder à l'élément. Si c'est un membre, tout le monde peut le voir et le modifier (à éviter). Si c'est une méthode tout le monde peut l'utiliser.
- ❖ **private** – Cet élément n'est accessible que depuis les méthodes de la classe.
- ❖ **protected** - Cet élément n'est accessible que depuis les classes dérivées (héritage) ou celles appartenant au même paquetage (**package** nom).
- ❖ Si on ne met rien l'élément n'est accessible que depuis les classes appartenant au même paquetage (pas les classes dérivées)

# Exemple de classe

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

*bibliothèques*

```
class LigneDeTexte {
```

*classe*

```
    private String texte; // le texte
```

```
    private Color couleurTexte; // sa couleur
```

*propriétés*

```
    public void changerTexte(String t) { // changer le texte
        texte=t; }
```

```
    public void changerCouleur(Color c) { // changer la couleur
        couleurTexte =c; }
```

```
    public String obtenirTexte() { // retourne le texte
        return texte; }
```

*méthodes*

```
    ....
```

```
}
```



# Classes – Constructeurs

- Quand un objet est créé (instanciation) l'appel du constructeur est implicite. Il est **automatiquement** fait quand on utilise l'opérateur **new**. Les **constructeurs** ont des caractéristiques spéciales:
  - Le nom du constructeur est le même que celui de la classe.
  - Il doit être public
  - Il peut accepter des paramètres.
  - Il ne retourne pas de valeur (mais on ne met pas void).
  - Le constructeur ne **peut pas être invoqué explicitement** sauf au début d'un autre constructeur de la même classe ou d'une classe fille.
- Une classe peut définir **plusieurs constructeurs**. Java utilisera celui dont les arguments correspondent.

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

```
class LigneDeTexte {
```

```
    private String texte; // le texte
```

```
    private Color couleurTexte; // sa couleur
```

```
    public LigneDeTexte(String t, Color c) { // création avec texte et couleur
```

```
        texte=t; // le texte
```

```
        couleurTexte=c; // la couleur
```

```
    }
```

```
    public void changerTexte(String t) { // changer le texte
```

```
        texte=t; }
```

```
    public void changerCouleur(Color c) { // changer la couleur
```

```
        couleurTexte =c;      }
```

```
    public String obtenirTexte() { // retourne le texte
```

```
        return texte;      }
```

```
}
```

Constructeur

# Surdéfinition de méthodes

Une classe peut avoir plusieurs méthodes ayant le même nom si on peut les différencier par les types des arguments ou leur nombre. On dit que la méthode est **surdéfinie**.

```
public LigneDeTexte(String t) { // création avec texte en noir
    texte=t; // le texte
    couleurTexte=new Color(0,0,0); // la couleur est noire
}
public LigneDeTexte(String t, int r, int v, int b) { // création avec texte et composantes
    texte=t; // le texte
    couleurTexte=new Color(r,v,b); // la couleur est créée
}
```

Constructeurs  
surdéfinis

```
public String obtenirTexte(int taille) { // retourne une partie du texte
    return texte.substring(0, taille);
}
public void changerCouleur(int r, int v, int b) {
    couleurTexte=new Color(r,v,b);
}
```

Méthodes  
surdéfinies

# Héritage

- L'héritage permet de créer de nouvelles classes à partir de classes existantes.
- Les objets des **classes dérivées** contiennent leurs propriétés et méthodes propres ainsi que ceux de la **classe de base**.  

```
class Fille extends Mere {  
    déclarations_de_propriétés ; // propres à Fille  
    déclarations_de_méthodes ; // propres à Fille  
}
```
- Lorsqu'une méthode qui existait dans Mere est re-définie dans Fille on parle de **surcharge**.

**L'héritage N'EST PAS la composition**

# Héritage

- Tout ce que contient la définition de Mere sera repris dans la classe Fille, de plus, Fille contient ses propres propriétés et méthodes.
- Un objet de classe Fille **EST** un objet de classe Mere.
- Construction d'objets de classes dérivées  
Quand on crée un objet d'une classe dérivée on crée implicitement un objet de la classe de base
- Surcharge de méthodes  
Quand on invoque une méthode **surchargée** c'est celle de la classe Fille et non celle de la classe Mere qui sera utilisée
- Utilisation de **super**  
**super** représente une référence interne implicite à la classe mère
  - **super**.nom\_méthode(params); appelle une méthode de la classe mère
  - **super**(params); doit être placé en première ligne d'un constructeur pour construire la classe mère avec paramètres.

# Héritage multiple (interface)

- En Java il n'y a pas d'héritage multiple
- Une **interface** définit un comportement (ne contient que des entêtes de méthodes). Ces méthodes devront être définies,  
**interface** nom\_d\_interface { // **interface** remplace **class**  
    type\_retour nom\_méthode ( liste\_arguments );  
    ...  
}
- Classe qui utilise des interfaces (**implements** remplace **extends**)  
    class NomDeClasse **implements** NomDInterface1, NomDInterface2 ,... {  
        // contenu de la classe NomDeClasse **toutes les méthodes doivent être écrites**  
    }
- Classe qui utilise un héritage et des interfaces (**1 extends** et **N implements**)  
    class NomDeClasseFille **extends** NomDeClasseMere  
        **implements** NomDInterface1, NomDInterface2 ,... {  
            // contenu de la classe NomDeClasseFille  
        }

# Polymorphisme

- Un objet de classe Fille peut se **substituer** à un objet de classe Mere
- Quand on invoque une méthode qui existe dans Mere et dans Fille  
c'est celle de la classe Fille qui sera utilisée au lieu de celle de la classe Mere
- Opérateur **cast**  
Truc f = **(Truc)m**; /\* m est converti en objet de classe Truc ce qui a un sens si  
m peut être converti ainsi par exemple si m appartient à une classe qui hérite  
de Truc \*/
- **La classe Object**  
En Java il existe une classe de base qui est la racine de toute la hiérarchie et dont  
héritent toutes les classes même si ce n'est pas explicitement dit en utilisant  
**extends**.  
Cette classe s'appelle **Object** et contient quelques méthodes de base utilisées par les  
collections et par la machine virtuelle (sérialisation, synchronisation ...).

# La référence this

Il est souvent utile de disposer d'une référence vers l'objet qui s'exécute.  
C'est faisable en utilisant **this**.

**this** est une référence implicite à l'objet lui-même

C'est un autre nom de l'objet lui-même.

**this(paramètres)** est une référence explicite au constructeur de la classe

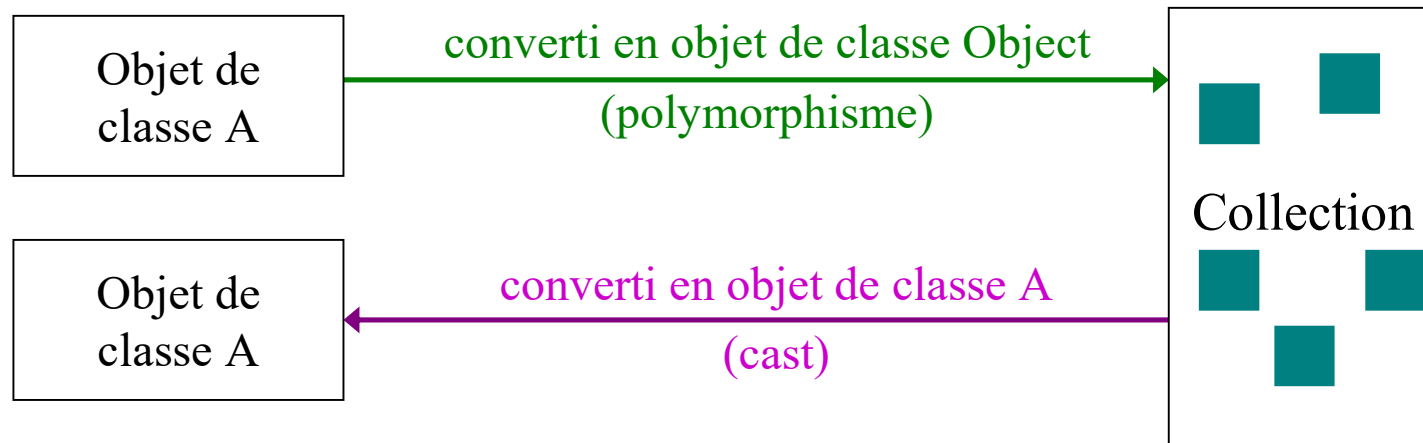
- Ne peut être utilisé qu'en début d'un autre constructeur



# Collections

Java possède des classes qui permettent de faire des collections d'objets.

Les objets de ces collections sont de classe **Object** ainsi les collections peuvent être utilisées pour toutes classes d'objets



# Classe Vector (tableau dynamique)

Un Vector est un tableau dont la taille s'adapte au contenu

- Déclaration

```
Vector<A> v;
```

Tous les éléments de la collection sont de classe A (ou d'une classe dérivée de A), A peut être Object.

- Création

```
v = new Vector<A>();
```

La collection est créée vide

- Vecteurs non typés

```
Vector<Object> v; // tout objet est de classe Object
```

# Classe Vector

- Vector()
- void insertElementAt(Object, int)
- void addElement(Object)
- boolean contains(Object)
- void copyInto(Object[])
- Object elementAt(int)
- Object firstElement()
- Object lastElement()
- int indexOf(Object,int)
- int lastIndexOf(Object, int)
- void removeElementAt(int)
- void removeAllElements()
- void removeElement(Object)
- void removeRange(int, int)
- void setElementAt(int, Object)
- boolean isEmpty()
- int size()

# Exemple d'utilisation d'un Vector typé

```
import java.util.*; // pour utiliser les collections
```

```
// Déclaration de la collection
```

```
Vector<String> mots=new Vector<String>(); // collection de chaînes
```

```
// ajout d'éléments dans la collection
```

```
String texte1=new String("Bonjour");
```

```
mots.addElement(texte1); // mettre le chaîne dans la collection
```

```
mots.addElement(new String("rien"));
```

```
// récupération des éléments
```

```
String premier = mots.elementAt(0);
```

```
String second = mots.elementAt(1);
```

# Exemple d'utilisation d'un Vector non typé

- Mettre des objets dans une collection

**// Déclaration de la collection**

**Vector<Object> trucs=new Vector<Object>(); // collection de trucs divers**

**// ajout d'éléments dans la collection**

**String texte1=new String("Bonjour");**

**trucs.addElement(texte1); // mettre le chaîne dans la collection**

**Color couleur=new Color(255,100,200);**

**trucs.addElement(couleur); // mettre la couleur dans la collection**

**trucs.addElement(new Integer(25)); // mettre l'entier dans la collection**

# Exemple d'utilisation d'un Vector non typé

- Utiliser les objets d'une collection

.....

```
// récupération du 1er élément qui est une chaîne  
String premier = (String) trucs.elementAt(0);
```

....

```
// récupération du 2ème élément qui est une couleur  
Color coul = (Color) trucs.elementAt(1);
```

.....

```
// enlever le 1er élément  
trucs.removeElementAt(0);
```

.....

# Classe HashMap (tableau indexé)

Les éléments de la collection sont associés à des clés. Les éléments sont des objets quelconques ainsi que les clés

- Déclaration

`HashMap<A,B> h;`

Tous les éléments de la collection sont de classe B (ou d'une classe dérivée de B) et les clés sont des objets de classe A

Attention : la classe A doit avoir une méthode **equals** permettant de savoir si deux clés sont égales (les classes String, Long, Integer ... ont ça).

- Création

`h = new HashMap<A,B> ();`

La collection est créée vide

- Exemples :

`HashMap<String, B>` collection dont les clés sont des noms

`HashMap<Long, B>` collection dont les clés sont des entiers longs

# Classe HashMap

- `HashMap()` // construction
- `HashMap(int)` // taille initiale
- `Object get(Object)` // clé
- `void put(Object, Object)` //clé , valeur
- `void remove(Object)` // clé
- `void clear()` // tout enlever
- `boolean isEmpty()`
- `int size()`



# HashMap

- **Création :**

```
HashMap<String, Color> table = new HashMap<String, Color>();
```

- **Ajout d'éléments :**

```
table.put("rouge", new Color(255,0,0,));
```

Remplace si y est déjà

- **Recherche d'éléments :**

```
Color coul = table.get("bleu");
```

Revoie null si n'existe pas

- **Parcours de la table :**

```
Set cles = table.keySet(); // Ensemble de clés
```

```
Iterator it = cles.iterator(); // Itérateur de parcours de la table
```

```
while (it.hasNext()){ // Tant qu'il reste des éléments
```

```
    String cle = it.next(); // Récupérer la clé suivante
```

```
    Color coul = map.get(cle); // Récupérer l'élément correspondant
```

```
}
```

# Exceptions

Les exceptions sont le mécanisme par lequel on peut contrôler dans un programme en Java les conditions d'**erreurs** qui se produisent.

Pour capturer une exception on utilise la construction **try** / **catch**, de la façon suivante:

```
try {  
    code qui peut lever une exception  
}  
catch (Classe_Exception nom_exception) {  
    code qui s'exécutera si l'exception se produit  
}
```

- **Exemple : conversion d'une chaîne s en un entier v :**

```
try { v=Integer.parseInt(s) } // tenter de convertir s en entier  
catch (NumberFormatException ie) { v=0; } // si on ne peut pas on prend 0.
```

# Exceptions

Si le code peut lever plusieurs exceptions on mettra plusieurs blocs catch

```
try {  
    code qui peut lever une exception  
}  
catch (Classe_Exception1 nom_exception) {  
    code qui s'exécutera si l'exception se produit  
}  
catch (Classe_Exception2 nom_exception) {  
    code qui s'exécutera si l'exception se produit  
}
```

Attention : l'ordre des blocs catch dépend des classes d'exception (les plus précises en premier en raison du polymorphisme)

# URLs (Uniform Resource Locator)

- Forme écrite:

**protocole**://**machine**[:**port**]/**path**[#**partie**][**paramètres**]

**http**://**onesearch.sun.com**/**search/onesearch/index.jsp**?**qt=jmf&x=6&y=10**

- La classe **URL** :

**URL**(String)

**URL**(String, String, String)

String toString()

String getFile()

String getHost()

int getPort()

String getProtocol()

} **MalformedURLException**

```
URL chemin=null;
try {
    chemin = new
        URL("http://docs.oracle.com/javase/1.5.0/docs/api/");
    // utiliser l'URL chemin
}
catch (MalformedURLException mue) {
    // ce que l'on fait si chemin n'est pas une URL
}
```

# Interface graphique avec SWING

Une interface graphique est construite à base de **composants d'interface**. Ces composants permettent à l'utilisateur d'interagir avec l'application.

- ❖ Les **contenants** contiennent les **composants d'interface**.
- ❖ On n'utilise pas de positions fixes pour les composants, mais ils sont disposés au moyen d'un placement contrôlé (**layouts**)
- ❖ L'accès se fait par le clavier et la souris au moyen de la gestion des **événements**

Dans SWING :

- les **composants** de l'interface sont des instances de la classe **JComponent** ou de l'une de ses classes filles. **JComponent** est fille de **Container**
- Les **contenants** sont des instances de la classe **Container** ou de l'une de ses classes filles. **Container** est fille de **Component**.

# La fenêtre : classe JFrame

~~La fenêtre associée~~

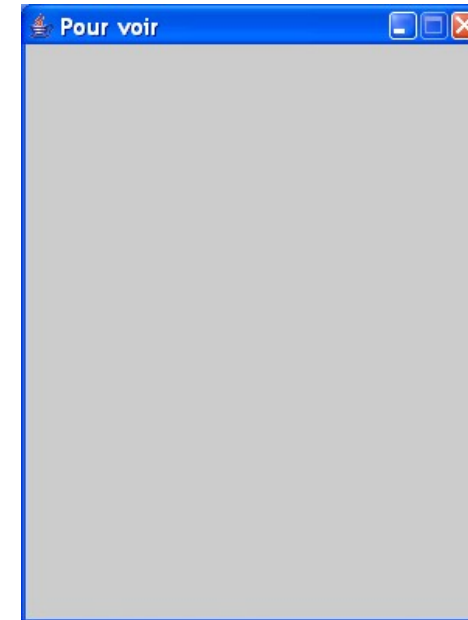
- JFrame(String)
- Container getContentPane()
- setContentPane(Container)
- void dispose()
- Image getIconImage()
- void setIconImage(Image)
- String getTitle()
- void setTitle(String)
- void toBack()
- void toFront()
- void pack()
- boolean isResizable()
- void setResizable(boolean)
- void setJMenuBar(MenuBar)
- void setVisible(boolean)

# Exemple de création de fenêtre

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

class TestFenetre extends JFrame {

    public TestFenetre (String nom) {
        super(nom);
        setSize(300,400);
        setResizable(false);
        setVisible(true);
    }
}
```



```
class ProgrammePourVoir {
    static public void main(String argv[]) {
        new TestFenetre("Pour voir");
    }
}
```

# Fermeture de la fenêtre

- La fenêtre se ferme par la croix en haut à droite
- Quand elle se ferme le programme continue de fonctionner. C'est normal car un programme peut ouvrir plusieurs fenêtres.
- Pour que le programme s'arrête quand une fenêtre se ferme :  
`setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);`
- On peut aussi associer un écouteur d'événements à cette fermeture qui terminera le programme par : `System.exit(0);`  
Mais on pourra faire autre chose avant (sauvegarder)



# Les contenants

~~6671112~~ composition

## La classe Component

boolean isShowing()  
void setVisible(boolean)

boolean isEnabled()  
void enable()  
void disable()

Color getForeground()  
void setForeground(Color)  
Color getBackground()  
void setBackground(Color)

Font getFont()  
void setFont(Font)

Cursor getCursor()  
void setCursor(Cursor)

Dimension getSize() // utiliser .width y .height  
void resize(int, int)  
void setLocation(int , int)  
Point location() // utiliser .x y .y

Graphics2D getGraphics()  
void repaint()  
void repaint(int, int, int, int)

**Graphics2D** hérite de **Graphics** mais possède des méthodes supplémentaires (rotation ...)

# Les composants

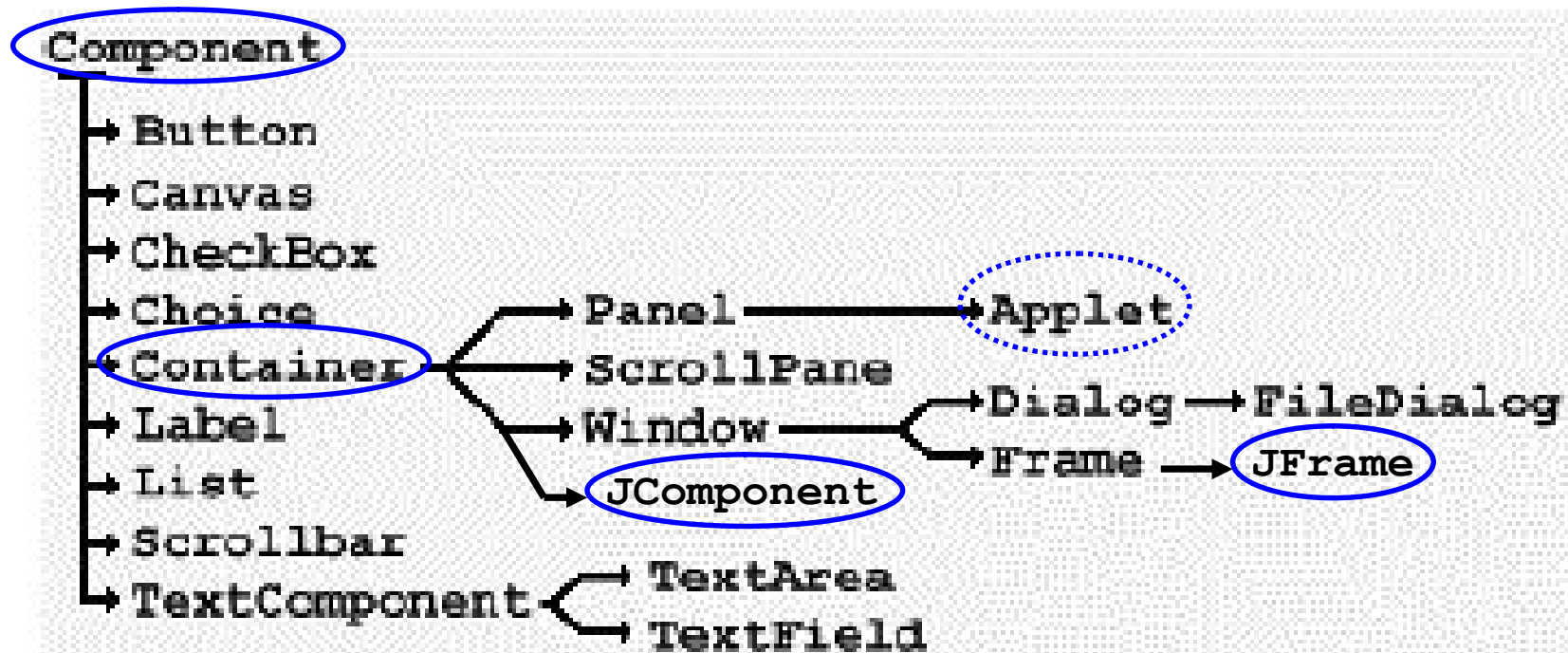
## La classe JComponent (h rite de Component)

```
void setPreferredSize(Dimension)  
void setMinimumSize(Dimension)  
void setMaximumSize(Dimension)  
JRootPane getRootpane()
```

```
void setToolTipText(String)
```



# Hiérarchie partielle de la classe Component



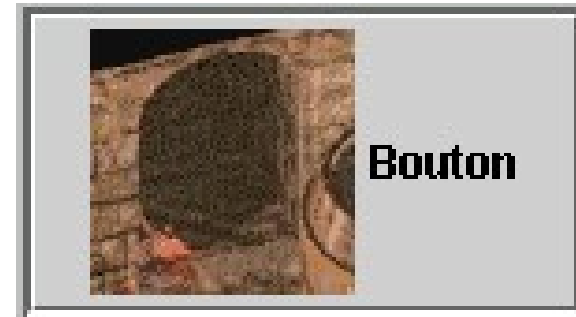
# Les Composants



Partie de la hiérarchie de JComponent

# Les composants (*Boutons*)

- JButton(String)
- JButton(ImageIcon)
- JButton(String , ImageIcon)
- String getText()
- void setText(String)



## *Cases à cocher*

- JCheckbox(String, boolean)
- void setSelected (boolean)
- boolean isSelected ()
- String getText()
- void setText(String)



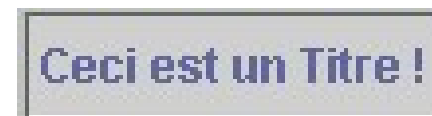
# Listes de choix

- JComboBox()
- JComboBox(Object[])
- void addItem(Object)
- int getSelectedIndex()
- Object getSelectedItem()
- void setSelectedIndex(int)



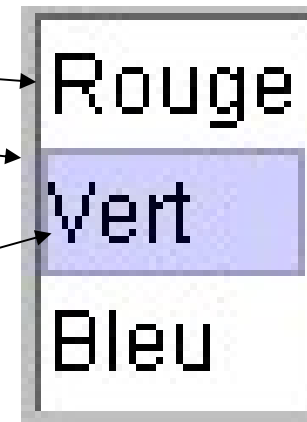
# Étiquettes

- JLabel(String)
- JLabel(ImageIcon)
- JLabel(String, ImageIcon)
- void setText(String)
- String getText()
- void setIcon(ImageIcon)



# *Listes*

- JList()
- JList(Object[])  
setListData(Object[]) →
- void setVisibleRowCount(int) →
- int getSelectedIndex() →
- int[] getSelectedIndices() →
- Object getSelectedValue()
- Object[] getSelectedValues()
- void setSelectedIndex(int)
- void setSelectedIndices(int[]) →
- void clearSelection()



# Ascenseurs

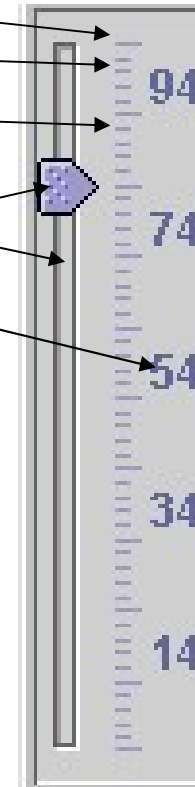
- JScrollbar(int,int,int,int,int) orientation (**JScrollbar.HORIZONTAL** ou **JScrollbar.VERTICAL**), position initiale, pas en mode page à page, valeurs minimales et maximales
- int getValue()
- void setValue(int)
- int getBlockIncrement()
- void setBlockIncrement (int)
- int getUnitIncrement()
- void setUnitIncrement (int)
- int getMaximum()
- void setMaximum (int)
- int getMinimum()
- void setMinimum (int)





# Curseurs

- `JSlider(int,int,int,int,int)` orientation (`JSlider.HORIZONTAL` ou `JSlider.VERTICAL`), valeurs minimales et maximales, position initiale.
- `void setMajorTickSpacing(int)`
- `void setMinorTickSpacing(int)`
- `void setPaintTicks(boolean)`
- `void setPaintTrack(boolean)`
- `void setPaintLabels(boolean)`
- `int getValue()`
- `void setValue(int)`
- `int getUnitIncrement()`
- `void setUnitIncrement (int)`
- `int getMaximum()`
- `void setMaximum (int)`
- `int getMinimum()`
- `void setMinimum (int)`



# *Barres de progression*

- `JProgressBar (int,int,int)` orientation (`JProgressBar.HORIZONTAL` ou `JProgressBar.VERTICAL`) , valeurs minimales et maximales.
- `int getValue()`
- `void setValue(int)`
- `int getMaximum()`
- `void setMaximum(int)`
- `int getMinimum()`
- `void setMinimum(int)`



# Les zones de texte

## *(JTextField et JTextArea)*

<ul style="list-style-type: none"><li>• void copy()</li><li>• void cut()</li><li>• void paste()</li></ul>	Copier/coller
<ul style="list-style-type: none"><li>• String getText()</li><li>• void setText(String)</li></ul>	Lire/écrire
<ul style="list-style-type: none"><li>• int getCaretPosition()</li><li>• int setCaretPosition(int)</li><li>• int moveCaretPosition(int)</li></ul>	Curseur
<ul style="list-style-type: none"><li>• setEditable(boolean)</li><li>• int getSelectionStart()</li><li>• int getSelectionEnd()</li><li>• void setSelectedTextColor(Color)</li><li>• String getSelectedText()</li><li>• void select(int,int)</li><li>• void selectAll()</li></ul>	Modification     Sélection
<ul style="list-style-type: none"><li>• Document getDocument()</li></ul>	Contenu

# Les zones de texte

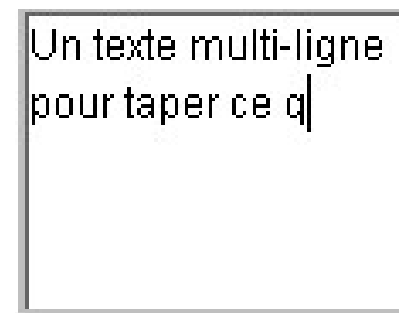
## *Zones de saisie*

- `TextField(String)`
- `TextField(String, int)`



## *Zones de texte*

- `TextArea(String, int, int)`
- `void append(String)`
- `void insert(String,int)`
- `void setTabSize(int)`
- `void setLineWrap(boolean)`
- `void setWrapStyleWord(boolean)`



# Les documents

## *La classe Document*

associée aux classes de texte (méthode `getDocument()`)

Il faut importer : `javax.swing.text`

### Opérations :

- `int getLength()`
- `String getText(int, int)` : exception `BadLocationException`
- `void remove(int, int)`

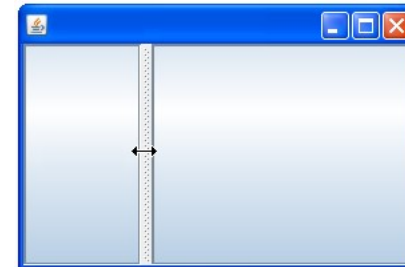
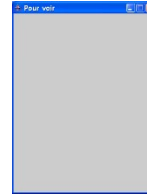
Evénements : Le document peut réagir à des événements (insertion/suppression/remplacement)

# Les contenants

- Les **contenants** contiennent et organisent la position des **composants**. De plus, les **contenants** sont, eux-mêmes, des **composants** et, comme tels, peuvent être situés dans d'autres **contenants**.
- Les **contenants** sont des instances de la classe **Container** ou d'une de ses filles.
  - void setLayout(LayoutManager) *définition du placement*
  - void add(Component) *ajout d'un composant*
  - void add(Component, Object)
  - void remove(Component) *suppression de composants*
  - void removeAll()

# Les contenants

- **JPanel** : c'est un contenant général
- **JScrollPane** : permet l'utilisation d'ascenseurs
- **JLayeredPane** : introduit une dimension de profondeur, pour placer les composants
- **JSplitPane** : divise une zone en 2 parties dont les tailles peuvent être modifiées par l'utilisateur
- **JTabbedPane** : Permet à plusieurs composants de partager un même espace avec des onglets.



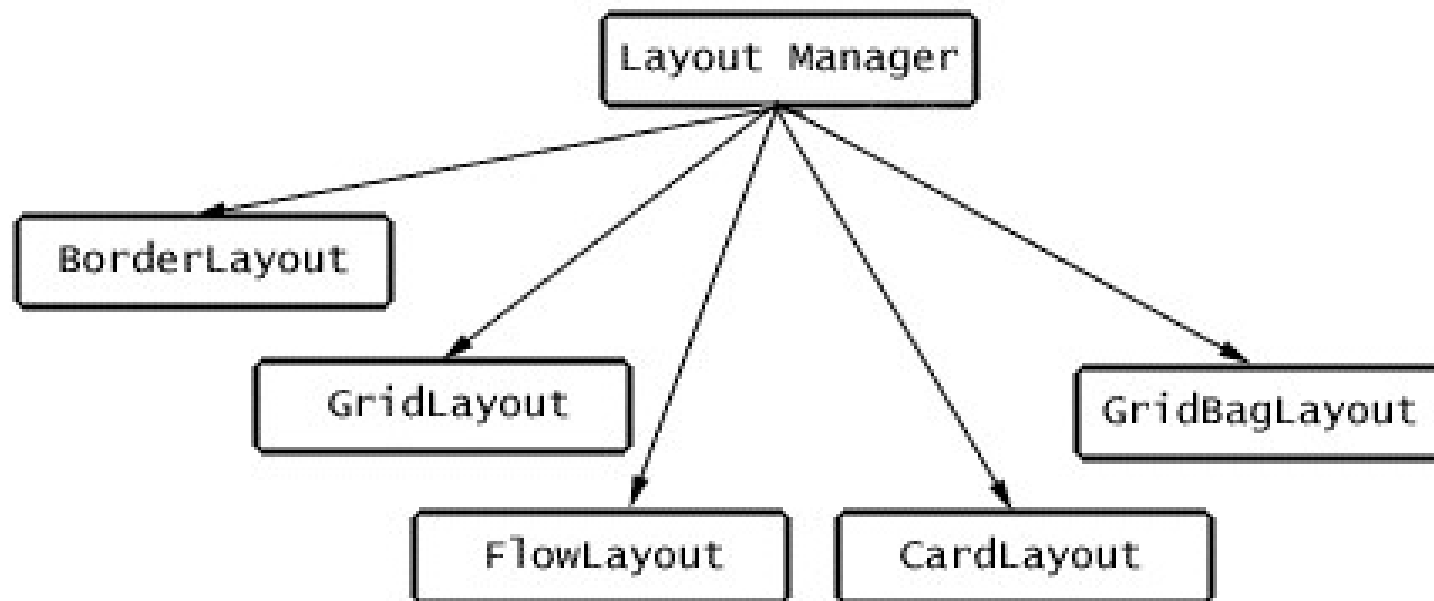
- **JInternalFrame** :

# Concevoir une interface

1. Dessiner l'interface avec tous les composants
2. Faire une classe fille de **JFrame**
3. Définir , si nécessaire, une barre de menu
4. Ajouter, cette barre de menu à la fenêtre avec la méthode **setMenuBar(JMenuBar)**
5. Récupérer le **contenant** associé à la fenêtre avec la méthode **getContentPane**. Ce **contenant** est de classe **JPanel**
6. Choisir un **objet de placement** et l'associer au **contenant** avec la méthode **setLayout**
7. Avec la méthode **add** du **contenant**, placer les **composants de l'interface**. Ceci seulement pour les **composants** qui occupent seuls une zone. Il faudra mettre un **contenant** dans les zones où placer plusieurs **composants**. Ce contenant est de classe **JPanel**, **JScrollPane**, **JLayeredPane**, **JSplitPane**, **JTabbedPane** ou **JInternalFrame**
8. Dans ce cas, ajouter ce **contenant** à son propre **contenant** avec la méthode **add**. Puis, pour chacun de ces **contenants**, faire les étapes 6 et 7 pour placer les **composants** ou les **contenants**
9. Continuer jusqu'à ce que tous les **composants** soient placés.



# Les objets de placement



# Objets de placement (FlowLayout)

- Les composants ajoutés avec **FlowLayout** se disposent sous forme de **liste**. La liste est horizontale, de gauche à droite, et on peut choisir l'espace entre chaque composant

- `FlowLayout(int,int,int)`

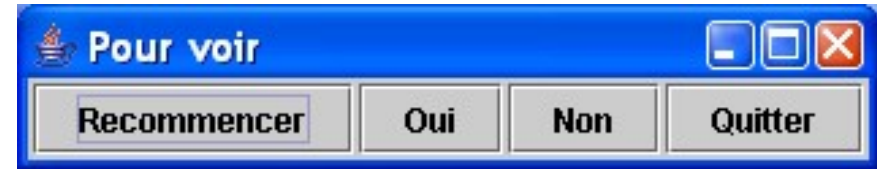


`FlowLayout.CENTER` ou `FlowLayout.LEFT` ou `FlowLayout.RIGHT`

Espacement horizontal

Espacement vertical

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



```
class TestFlow extends JFrame { // Classe de la fenêtre interface  
  
    private JButton recommencer,oui,non,quitter; // les objets d'interfaces utilisés  
  
    public TestFlow (String nom) { // constructeur de la fenêtre  
        super(nom); // Construction de JFrame  
        FlowLayout placement=new FlowLayout(FlowLayout.LEFT,2,2);  
        getContentPane().setLayout(placement); // choix du mode de placement  
        recommencer=new JButton("Recommencer");  
        getContentPane().add(recommencer); // ajout du 1er bouton  
        oui=new JButton("Oui");  
        getContentPane().add(oui); // ajout du 2ème bouton  
        non=new JButton("Non");  
        getContentPane().add(non); // ajout du 3ème bouton  
        quitter=new JButton("Quitter");  
        getContentPane().add(quitter); // ajout du 4ème bouton  
        pack(); // Taille de la fenêtre calculée par java  
        setVisible(true); // rendre la fenêtre visible  
    }  
  
    static public void main(String argv[]) { // programme qui crée la fenêtre  
        TestFlow f=new TestFlow ("Pour voir"); // création avec titre  
    }  
}
```

Création de  
cette interface

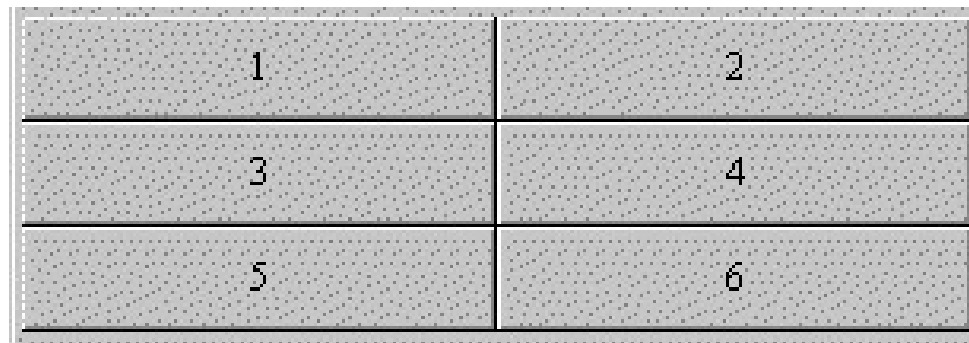
# Objets de placement (BorderLayout)

- Les composants ajoutés avec **BorderLayout** se disposent dans cinq zones: Nord, Sud, Est, Ouest et Centre.
- Pour placer les composants on utilise la méthode **add** avec deux paramètres, le second est un entier qui indique la position du composant :
  - BorderLayout.NORTH,
  - BorderLayout.SOUTH,
  - BorderLayout.EAST,
  - BorderLayout.WEST
  - BorderLayout.CENTER



# Objets de placement (GridLayout)

- L'objet de placement se crée avec un nombre de lignes et de colonnes et les Composants vont dans les cellules du tableau ainsi défini.
- `GridLayout(int,int,int,int)` lignes, colonnes, esp. horiz., esp. vert.

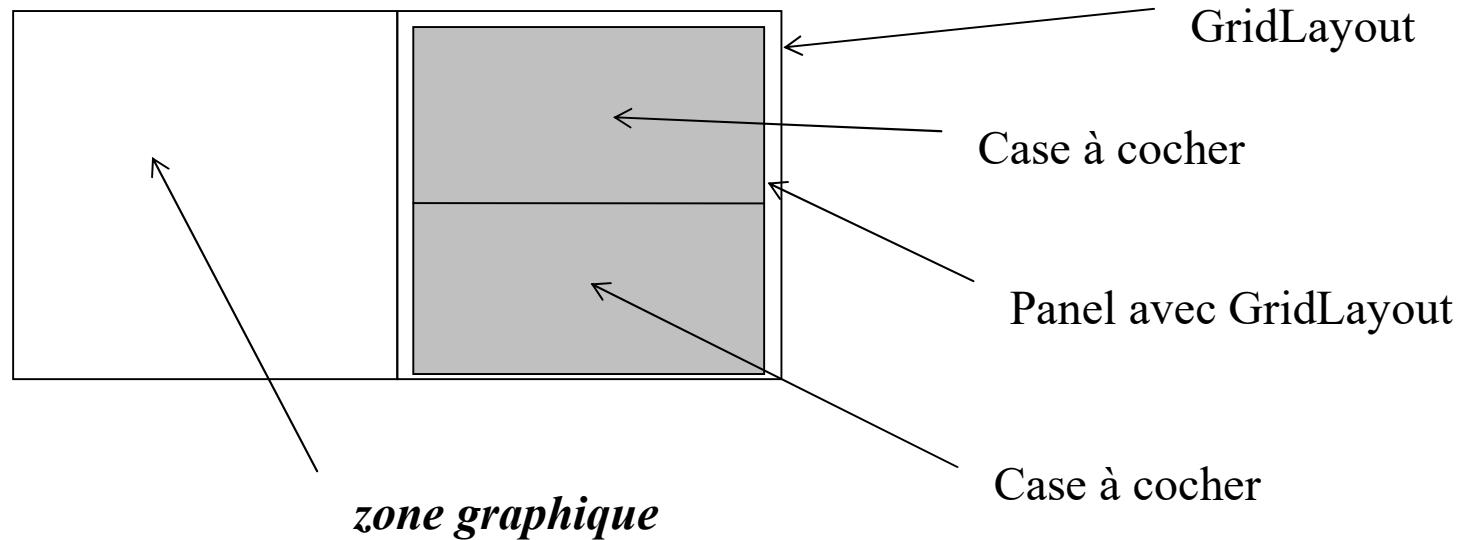


1	2
3	4
5	6

# Réaliser une interface



Réalisable avec 2  
GridLayout



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

## 2<sup>ème</sup> niveau

```
public class Langue extends JFrame {
```

```
    private JCheckBox francais, espagnol;
    private JLabel drapeau;
    private ImageIcon draF, draC, draAct;
```

```
    public Langue () {
        super("Choix de langue");
```

```
        draF=new ImageIcon("Fr.jpg");
        draC=new ImageIcon("Es.jpg");
        draAct = draC;
        drapeau =new JLabel(draAct);
        drapeau.setPreferredSize(90,50);
        GridLayout placement=new GridLayout(1,2,5,5);
        getContentPane().setLayout(placement);
        getContentPane().add(drapeau);
        JPanel droite=new JPanel();
        getContentPane().add(droite);
```

## 1<sup>er</sup> niveau

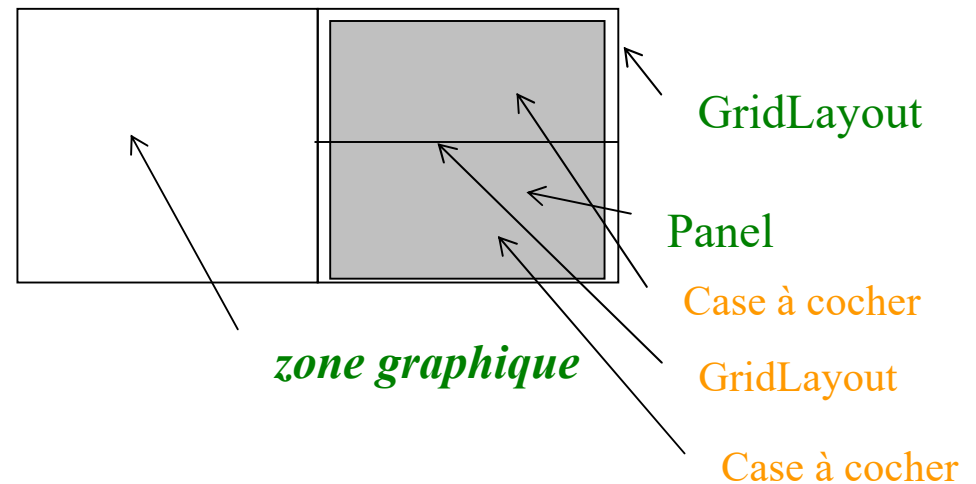
```
        GridLayout placement2=new GridLayout(2,1,3,3);
        droite.setLayout(placement2);
        francais=new JCheckBox("Français",true);
        droite.add(francais);
```

```
        espagnol=new JCheckBox("Castellano",false);
        droite.add(espagnol);
```

```
        pack();
        setVisible(true);
    }
```

```
    static public void main(String argv[]) {
        Langue f=new Langue ();
    }
```

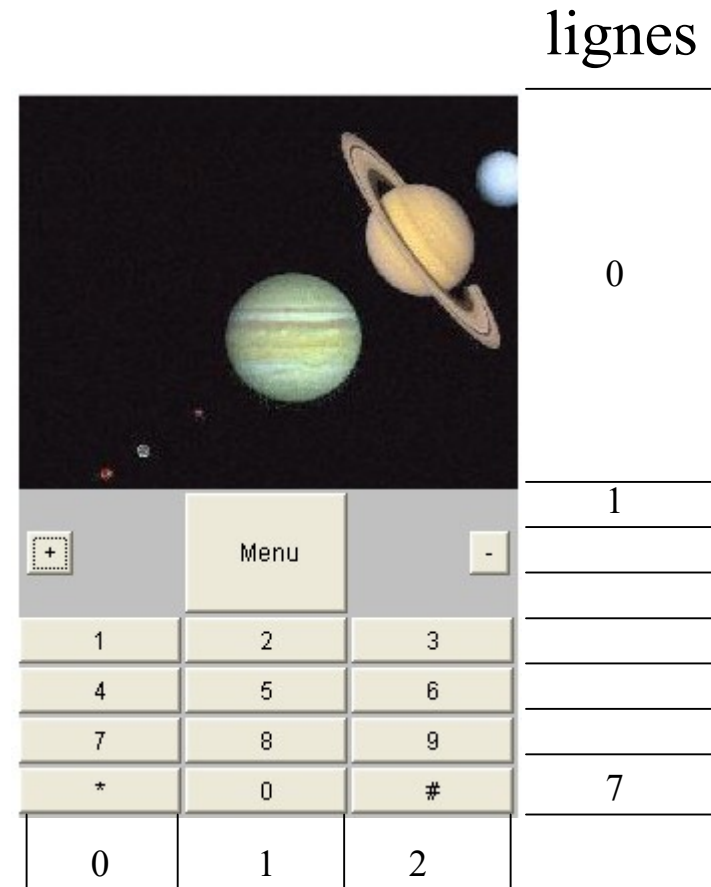
```
}
```



# Objets de placement (GridBagLayout)

- Identique à GridLayout, avec la différence que les lignes et les colonnes n'ont pas nécessairement la même taille.
- Un composant peut occuper plusieurs cases
- Les caractéristiques sont regroupées dans un objet de classe **GridBagConstraints**
- On associe ces caractéristiques à chaque composant par :  
**setConstraints(Component, GridBagConstraints)**

colonnes





# Contraintes de placement dans un GridBagLayout : GridBagConstraints

- gridx et gridy *position*
- gridwidth et gridheight *tailles en nombre de lignes et de colonnes*
- weightx et weighty *poids de redimensionnement en x et y*
- fill *remplissage des cases*  
GridBagConstraints.HORIZONTAL , VERTICAL , BOTH , NONE
- anchor *position dans les cases*  
GridBagConstraints.CENTER , NORTH, NORTHEAST, EAST, SOUTHEAST, SOUTH, SOUTHWEST, WEST, NORTHWEST
- ipadx et ipady *marges externes en x et y*
- Insets *marges internes*  
new Insets(haut, gauche, bas, droite)

```

public class Placement extends JFrame {
    private JButton soumettre;
    private JTextField formule, resultat;

    public Placement() {
        super("Exemple de placement");
        GridBagLayout placement=new GridBagLayout(); // objet de placement
        GridBagConstraints regles=new GridBagConstraints(); // définition des contraintes
        getContentPane(). setLayout(placement); // utiliser cet objet de placement pour la fenêtre
        regles.fill=GridBagConstraints.NONE; // remplissage complet des cases
        regles.anchor=GridBagConstraints.CENTER; // placement des composants
        regles.weightx=0; regles.weighty=0;
        regles.insets=new Insets(3,3,0,0); // marges externes
        regles.ipadx=2; regles.ipady=2; // marges internes

        // Placement des composants
        soumettre=new JButton("Soumettre");
        regles.gridx=1; regles.gridy=0; // coordonnees
        regles.gridwidth=1; regles.gridheight=1; // 1 case
        placement.setConstraints(soumettre, regles);
        getContentPane(). add(soumettre); // placement du bouton
        formule=new JTextField("Taper ici le texte à envoyer");
        regles.gridx=0; regles.gridy=0; // coordonnees
        regles.gridwidth=1; regles.gridheight=1; // 1 case
        placement.setConstraints(formule, regles);
        getContentPane(). add(formule);
        resultat=new JTextField("Lire ici la réponse");
        regles.gridx=0; regles.gridy=1; // coordonnees
        regles.gridwidth=2; regles.gridheight=1; // 2 cases
        placement.setConstraints(resultat, regles);
        getContentPane(). add(resultat);
        pack();
        setVisible(true);
    }
}

```



Création de  
cette interface

```

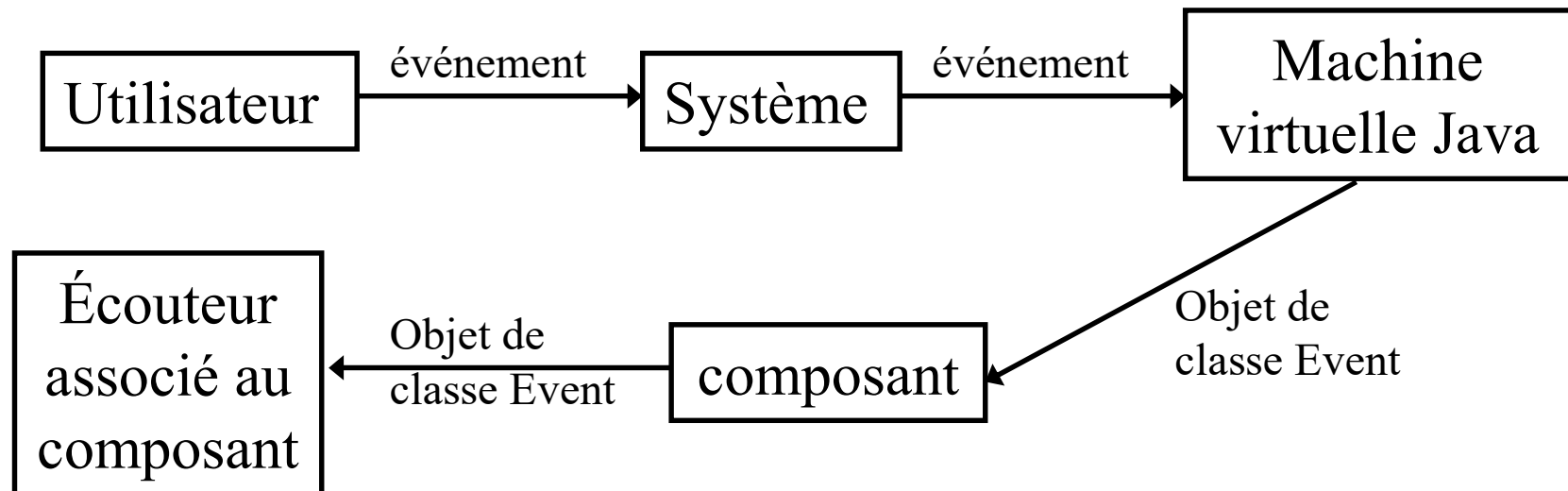
static public void main(String argv[]) {
    Placement interf=new Placement(); // création de l'interface
}
}

```

# Objets de placement (CardLayout)

- S'utilise quand on veut superposer des zones de composants
- **add** avec deux paramètres, le second est une chaîne de caractères (String) qui indique la position du composant (nom du contenant).
- Pas d'onglets on change de plan par programme  
void **show**(Container, String)

# Événements



- Quand un utilisateur interagit avec l'application, la machine virtuelle Java crée un **objet événement** (la classe dépend du type d'événement)
- Elle transmet cet objet au **composant**
- Le **composant** le transmet à son **écouteur d'événements**.
- L'**écouteur d'événements** exécute la méthode correspondante.

# Écouteur d'événements

C'est un objet qui **hérite** de la **classe modèle** (**xxxAdapter**) proposée par java pour traiter cette **classe d'événements** (**xxxEvent**) OU qui **implémente** l'**interface modèle** (**xxxListener**) proposée par java pour traiter cette **classe d'événements**.

Il est **associé au composant** par la méthode appropriée de ce composant (*add....Listener*)

Il contient des **méthodes** qui seront automatiquement appelées lorsque surviendra l'événement. Ces **méthodes** recevront en paramètre un objet de la **classe d'événements** appropriée.

# Événements élémentaires

Type d'événement	événement	classe modèle interface modèle	classe d'événement	nom de méthode associée
Visibilité	Prend le focus	<b>FocusAdapter</b>	<b>FocusEvent</b>	<b>focusGained</b>
	Perd le focus	<b>FocusListener</b>		<b>focusLost</b>
Clavier	touche appuyée	<b>KeyAdapter</b> <b>KeyListener</b>	<b>KeyEvent</b>	<b>keyPressed</b>
	touche lâchée			<b>keyReleased</b>
	touche tapée			<b>keyTyped</b>
Souris	clic	<b>MouseAdapter</b> <b>MouseListener</b>	<b>MouseEvent</b>	<b>mouseClicked</b>
	le curseur rentre dans le composant			<b>mouseEntered</b>
	le curseur sort du composant			<b>mouseExited</b>
	un bouton de la souris appuyé			<b>mousePressed</b>
	un bouton de la souris lâché			<b>mouseReleased</b>
	souris déplacée avec un bouton appuyé	<b>MouseMotionAdapter</b> <b>MouseMotionListener</b>	<b>MouseMotionEvent</b>	<b>mouseDragged</b>
	souris déplacée			<b>mouseMoved</b>

# Événements de fenêtres

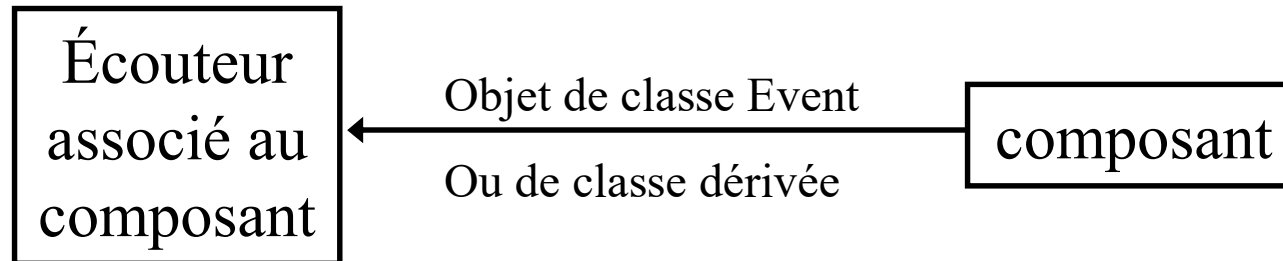
Fenêtre	événement	classe modèle interface modèle	classe d'événement	nom de méthode associée
normales : <b>JFrame</b> , <b>JDialog</b> et <b>JApplet</b>	La fenêtre devient active (elle recevra les saisies au clavier)	<b>WindowAdapter</b> <b>WindowListener</b>	<b>WindowEvent</b>	<b>windowActivated</b>
	La fenêtre devient inactive			<b>windowDeactivated</b>
	La fenêtre est fermée			<b>windowClosed</b>
	La fenêtre est en cours de fermeture (pas encore fermée)			<b>windowClosing</b>
	La fenêtre est mise ne icône			<b>windowIconified</b>
	La fenêtre est restaurée			<b>windowDeiconified</b>
	La fenêtre est visible pour la première fois			<b>windowOpened</b>
internes : <b>JInternalFrame</b>	La fenêtre devient active (elle recevra les saisies au clavier)	<b>InternalFrameAdapter</b> <b>InternalFrameListener</b>	<b>InternalFrameEvent</b>	<b>InternalFrameActivated</b>
	La fenêtre devient inactive			<b>InternalFrameDeactivated</b>
	La fenêtre est fermée			<b>InternalFrameClosed</b>
	La fenêtre est en cours de fermeture (pas encore fermée)			<b>InternalFrameClosing</b>
	La fenêtre est mise ne icône			<b>InternalFrameIconified</b>
	La fenêtre est restaurée			<b>InternalFrameDeiconified</b>
	La fenêtre est visible pour la première fois			<b>InternalFrameOpened</b>

# Événements de composants

Type d'événement	événement	interface modèle	<i>classe d'événement</i>	nom de méthode associée
Action	activation de bouton, case cochée, choix dans un menu ou choix d'un fichier, saisie de texte	<b>ActionListener</b>	<b>ActionEvent</b>	<b>actionPerformed</b>
Ajustement	modification de la position de l'ascenseur	<b>AdjustmentListener</b>	<b>AdjustmentEvent</b>	<b>adjustmentValueChanged</b>
Changement	modification de la position d'un curseur ou d'une barre de progression	<b>ChangeListener</b>	<b>ChangeEvent</b>	<b>stateChanged</b>
Elément	sélection d'un élément	<b>ItemListener</b>	<b>ItemEvent</b>	<b>itemStateChanged</b>
Document	modification, insertion ou suppression de texte	<b>DocumentListener</b>	<b>DocumentEvent</b>	<b>changedUpdate removeUpdate insertUpdate</b>
Curseur	déplacement du curseur d'insertion	<b>CaretListener</b>	<b>CaretEvent</b>	<b>caretUpdate</b>
Sélection	sélection d'un ou plusieurs éléments	<b>ListSelectionListener</b>	<b>ListSelectionEvent</b>	<b>valueChanged</b>



# Écouteurs d'événements



- L'écouteur d'événements est un objet qui **hérite** d'une **classe modèle** (**WindowAdapter, MouseAdapter, ...**) proposée par java pour traiter cette **classe d'événements** (**WindowEvent, ActionEvent, ...**)
- OU qui **implémente** l'**interface modèle** (**ActionListener, AdjustmentListener, ...**) proposée par java pour traiter cette **classe d'événements**.
- Il est **associé au composant** par la méthode appropriée de ce composant (**addActionListener, addAdjustmentListener, ...**)
  - Il contient des **méthodes** (**windowClosing, actionPerformed, ...**) qui seront automatiquement appelées lorsque surviendra l'événement. Ces **méthodes** recevront en paramètre un objet de la **classe d'événements** appropriée.

# Comment écrire un écouteur

Pour chaque élément dont on veut traiter les événements il faut :

1. Créer une classe fille de la **classe modèle** (**WindowAdapter**, **MouseAdapter**, ...)
- OU une classe implémentant l'**interface modèle** (**ActionListener**, **AdjustmentListener**, ...) correspondant à la **classe d'événements** (**WindowEvent**, **ActionEvent**, ...) que l'on veut traiter.
1. Écrire les actions associées aux événements dans les **méthodes** correspondantes (**windowClosing**, **actionPerformed**, ...).
2. **Associer un objet de cette classe au composant** avec la méthode **addxxxListener** (**addActionListener**, **addAdjustmentListener**, ...) du composant.

Par exemple, on implémentera **ActionListener**, on surchargera **actionPerformed** et on l'associera à un bouton par **addActionListener**

ou on héritera de **MouseAdapter**, on surchargera **mouseClicked** et **mouseEntered** puis on l'associera à une étiquette par **addMouseListener**

# Exemple d'écouteur

Par exemple, pour définir la classe associée aux actions sur une case à cocher :

```
private class ActionSurLaCase implements ActionListener {  
    public synchronized void actionPerformed(ActionEvent e) {  
        // action associée à l'activation de la case à cocher  
        // le paramètre permet de savoir quel est l'événement  
    }  
}
```

Pour associer ceci à la case à cocher :

```
maCase=new JCheckBox("une case à cocher",true)  
maCase.addActionListener(new ActionSurLaCase()).
```

# Actions dans l'exemple

Ce que l'on veut :



- **Case** cochée => drapeau correspondant
- Une et une seule **case** peut être case cochée à la fois
- Fermeture de la **fenêtre** => fin du programme
- Bouton souris appuyé sur **drapeau** => affichage d'une carte du pays
- Bouton souris lâché sur **drapeau** => retour au drapeau

⇒ Ecouteurs sur **case**, **fenêtre** et **drapeau**

**Case** (JCheckbox) : **ActionListener**

**Fenetre** (JFrame) : **WindowAdapter** ou **WindowListener**

**Drapeau** (JLabel) : **MouseListener** ou **MouseAdapter**

# Actions dans l'exemple

## Modifications de l'exemple :

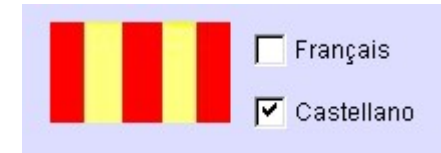
```
import java.awt.event.*;  
import javax.swing.event.*;
```

.....

```
public class Langue extends JFrame {  
    super("Choix de langue");
```

....

```
    addWindowListener(new GestionFenetre()); // évts de fenêtre  
    francais.addActionListener(new ChoixFrancais()); // évts des cases  
    espagnol.addActionListener(new ChoixEspagnol());  
    drapeau.addMouseListener(new ClicDrapeau()); // évts drapeau  
}
```



# Actions des cases à cocher

Règles à respecter :

- Quand on coche une case :

- Soit l'autre était cochée => il faut la dé-cocher
- Soit l'autre n'était pas cochée : impossible car ça signifie que les 2 cases étaient non cochées avant

- Si on décoche une case :

- Soit l'autre n'était pas cochée => il faut refuser de décocher cette case
- Soit l'autre était cochée : impossible car ça signifie que les 2 cases étaient cochées avant

Méthode à suivre quand on modifie une case (événement) :

- Si elle devient cochée on décoche l'autre
- Sinon on la re-coche
- On adapte l'image du drapeau à la case cochée

# Actions des cases à cocher

```
private class ChoixFrancais implements ActionListener {  
    public synchronized void actionPerformed(ActionEvent e) {  
        if (francais.isSelected()) {  
            espagnol.setSelected(false); // une seule case cochée  
            draAct = draF;  
            drapeau.setIcon(draAct); // changement du drapeau  
        }  
        else francais.setSelected(true); // une seule case cochée  
    }  
}
```

```
private class ChoixEspagnol implements ActionListener {  
    public synchronized void actionPerformed(ActionEvent e) {  
        if (espagnol.isSelected()) {  
            francais.setSelected(false); // une seule case cochée  
            draAct = draC;  
            drapeau.setIcon(draAct); // changement du drapeau  
        }  
        else espagnol.setSelected(true); // une seule case cochée  
    }  
}
```

# Actions de fenêtre

```
private class GestionFenetre extends WindowAdapter {  
    public synchronized void windowClosing(WindowEvent e) {  
        // On peut faire quelque chose avant de terminer le programme  
        System.exit(0); // terminer le programme  
    }  
}
```

Remarque : Si la fermeture de la fenêtre termine le programme on peut se passer d'un écouteur d'événements et mettre dans le constructeur de la fenêtre:

```
setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
```



# Actions sur le drapeau

```
private class ClicDrapeau extends MouseAdapter {  
    public synchronized void mousePressed(MouseEvent e) { // Afficher une carte  
        if (français.isSelected())  
            drapeau.setIcon(new ImageIcon("carteFR.jpg");  
        else  
            drapeau.setIcon(new ImageIcon("carteES.jpg");  
    }  
  
    public synchronized void mouseReleased(MouseEvent e) { // Retour au drapeau  
        drapeau.setIcon(draAct); // on remet l'image du drapeau  
    }  
}
```

Remarque : on a choisi d'utiliser l'héritage (`extends`) pour ne pas avoir à écrire les méthodes non utilisées (`mouseEntered`, `mouseExited`, `mouseClicked`),

# Evénements : problèmes

- Pour certains composants (JComboBox par exemple), quand un événement se produit l'écouteur est exécuté que l'événement vienne :
  - De l'utilisateur
  - Du programme
- Donc si, lors d'une action (par exemple sur un bouton) le programme sélectionne un élément, l'écouteur sera exécuté comme si l'utilisateur avait sélectionné cet élément.
- C'est par exemple le cas quand on ajoute un élément à une JComboBox car l'élément ajouté est automatiquement sélectionné ou quand on en enlève un car un autre peut être sélectionné

Selon ce que fait l'écouteur ça peut poser problème.

# Événements : problèmes

- Solution :

- Définir une propriété : `private boolean garde`; et l'initialiser à *false*
- Dans le programme (écouteur du bouton par exemple) avant de sélectionner, d'insérer ou d'enlever un élément dans la JComboBox faire : `garde = true`;
- Dans l'écouteur de la JComboBox mettre :

```
if(garde) { // modifié par programme ⇒ ne pas traiter
    garde = false; // réarmer la garde pour le prochain événement
    return;
} else { // modifié par l'utilisateur ⇒ traiter
    // Traitement normal
}
```

- Remarque :

`garde` est modifiée par plusieurs écouteurs en concurrence dans ce cas il faut la déclarer **volatile** : `private volatile boolean garde`;

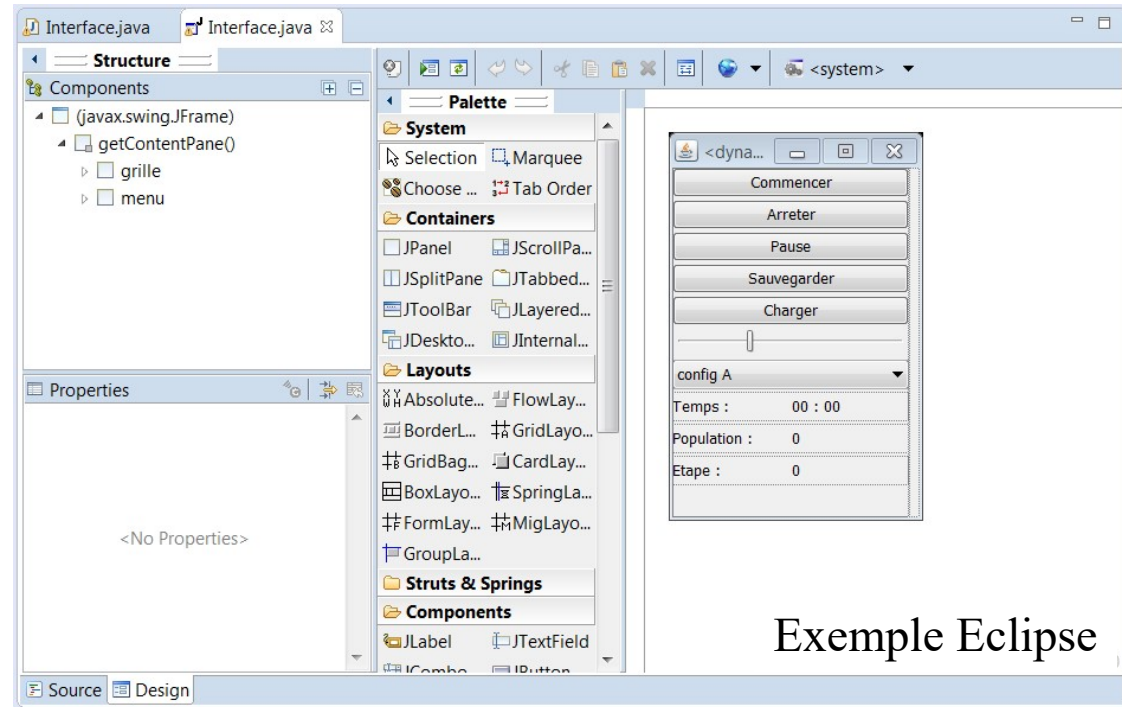
# Récapitulatif (création d'interfaces)

1. Faire une classe fille de **JFrame**.
2. Dans son constructeur récupérer le **contenant** associé à la fenêtre (méthode **getContentPane** de la fenêtre).
3. Lui associer un **objet de placement** (méthode **setLayout** du **contenant**)
4. Placer les **composants de l'interface** qui occupent seuls une zone (méthode **add** du **contenant**).
5. Placer des **contenants** dans les zones à redécouper (méthode **add** du **contenant** supérieur).
6. Pour chacun de ces **contenants**, faire les étapes 3 à 5 pour y placer des **composants de l'interface** ou des **contenants** jusqu'à ce que l'interface soit complète.
7. Ajouter un écouteur d'événements aux **composants de l'interface** qui en ont besoin (méthode **addxxxListener** du **composant de l'interface**) .
8. Pour chacun de ces écouteurs écrire une classe (qui hérite de **xxxAdapter** ou implémente **xxxListener**) en surchargeant les méthodes appropriées.
9. Tester le plus souvent possible ...

# Environnement de développement (Exemple d'Eclipse)

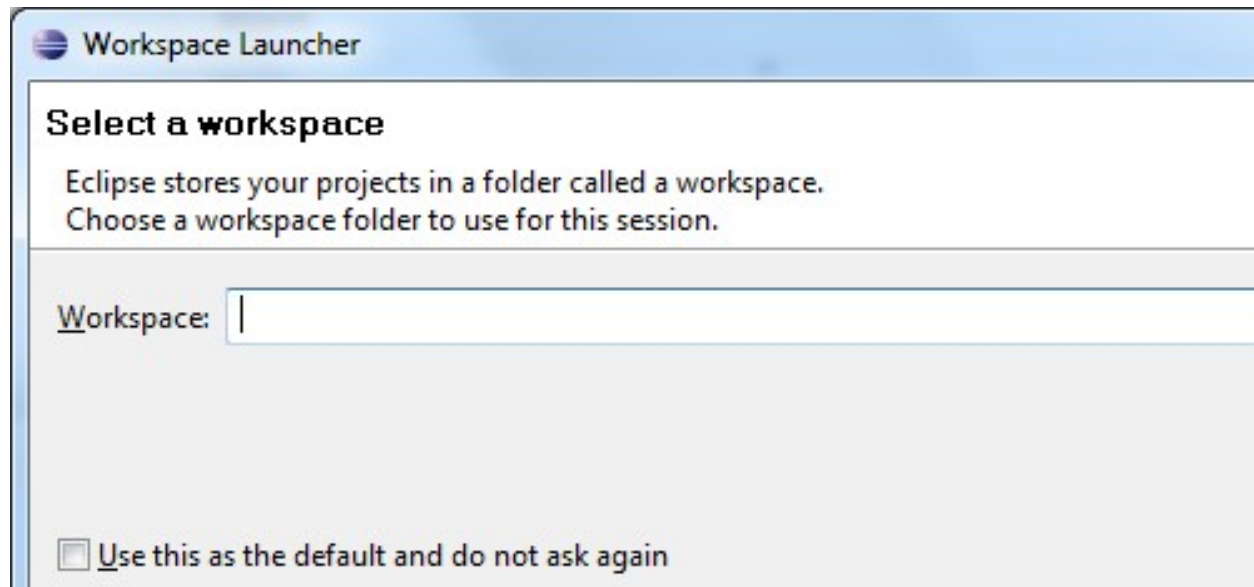
# Générateurs d'interface

- Graphiques
- Génèrent le code java



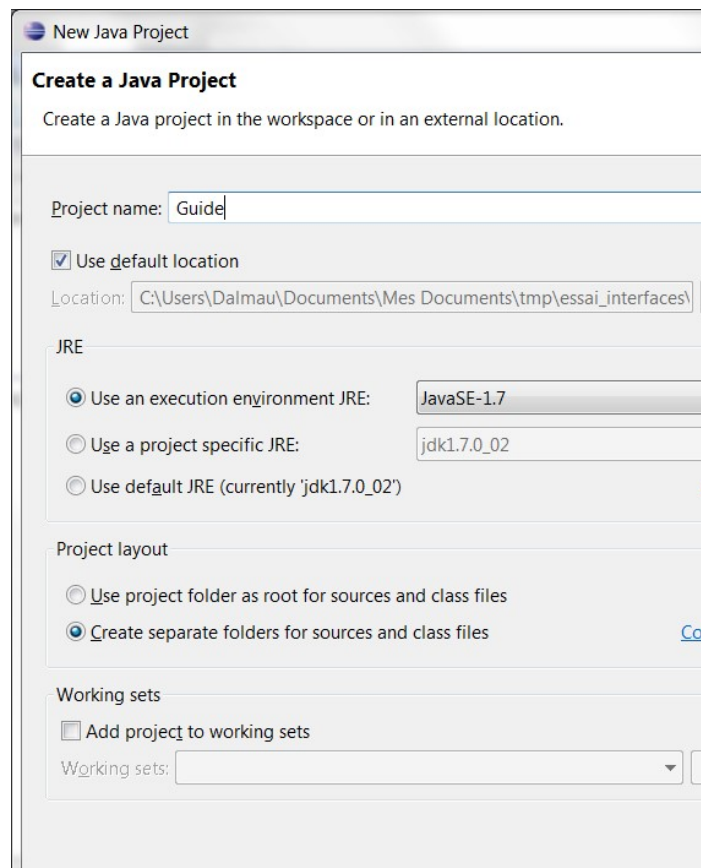
- Pas toujours faciles à utiliser pour des interfaces complexes
- Nécessité d'intervenir manuellement sur le code généré dans certains cas
- **Conclusion** : on peut s'en servir mais il vaut mieux savoir lire/modifier le code généré

# Répertoire de travail avec Eclipse



- Eclipse ne s'installe pas (simple copie) <https://eclipse.org/downloads/>
- Les paramètres d'Eclipse sont liés au répertoire de travail  
=> Possibilité d'utiliser des configurations différentes

# Créer un projet Eclipse



- L'espace de travail contient un ou plusieurs projets partageant les mêmes paramétrage.
- Il peut y avoir des liens entre projets



# Ajouter une classe au projet

**New Java Class**

**Java Class**  
⚠ The use of the default package is discouraged.

Source folder: Guide/src

Package: (default)

☐ Enclosing type:

Name: Fenetre

Modifiers: ☒ public ☐ default ☐ private ☐ protected  
☐ abstract ☐ final ☐ static

Superclass: java.lang.Object

Interfaces:

Which method stubs would you like to create?

☐ public static void main(String[] args)

☐ Constructors from superclass

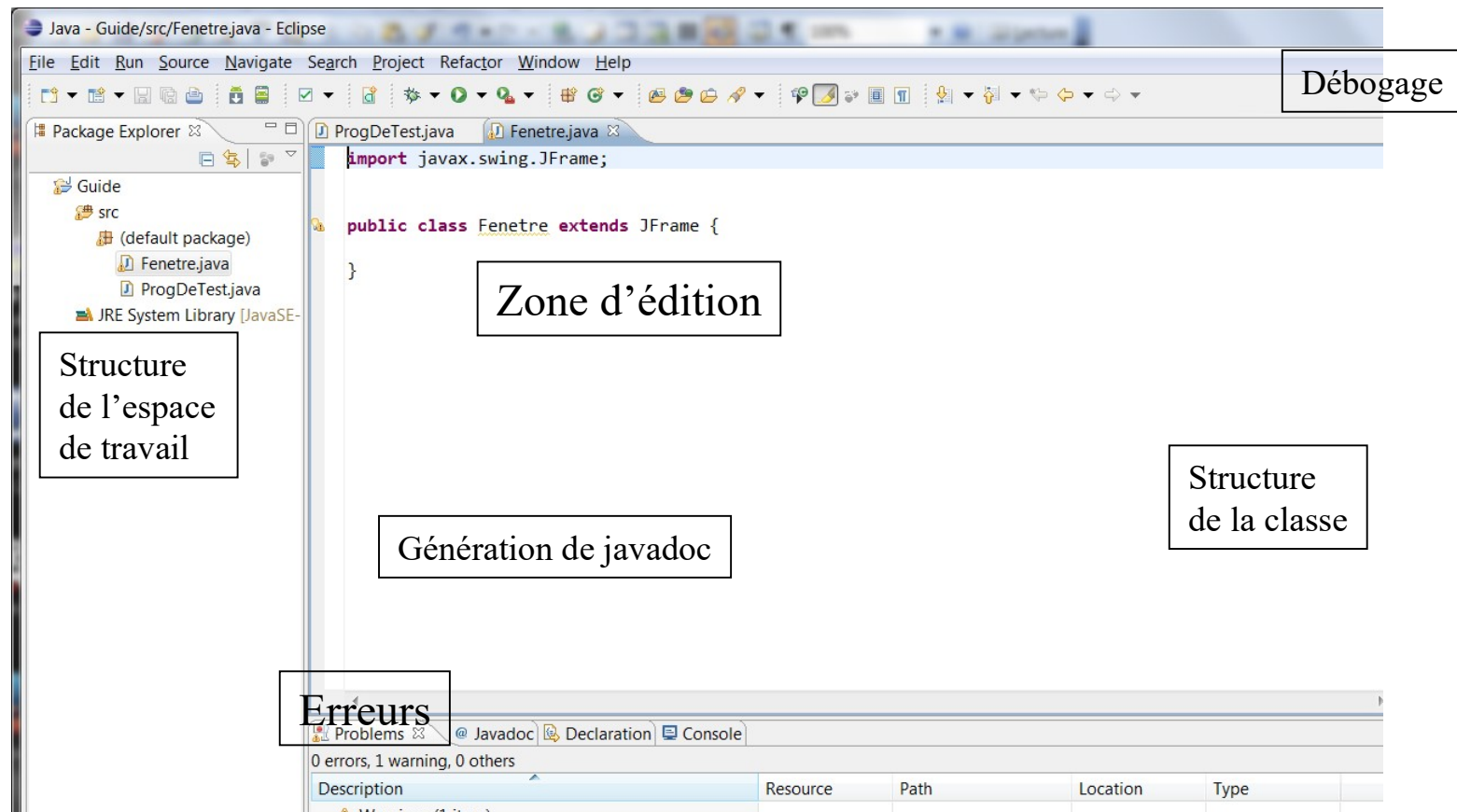
☒ Inherited abstract methods

Do you want to add comments? (Configure templates and default value [here](#))

☐ Generate comments

- Chaque classe est dans un paquetage (cas de *protected*)
- Si la classe hérite de JFrame on pourra l'ouvrir dans l'éditeur graphique d'interfaces

# Ecran principal d'Eclipse



# Flots et sérialisation

# Les flots

- Les entrées/sorties en java se font par des **flots**.
  - entrées/sorties classiques clavier et écran
  - entrées/sorties sur fichier ou sur Internet
  - entrées sorties sur réseau (sockets)
  - entrées sorties en mémoire (tableaux, buffers)
- Flots d'entrée
- Flots de sortie
- De nombreuses classes selon les accès possibles (octets, informations typées, objets ...)

# Flots d'entrée

- La classe **InputStream** n'offre que la méthode **read** permettant de lire un ou plusieurs octets.
- La classe **DataInputStream** offre des méthodes adaptées aux types primitifs comme **readInt**, **readChar**, **readLong** etc.

Il est possible de construire un objet de classe **DataInputStream** à partir d'un objet de classe **InputStream** en faisant :

```
DataInputStream lecture = new DataInputStream(objet_de_classe InputStream);
```

- La classe **BufferedReader** offre la méthode **readLine** pour les chaînes de caractères.

Il est possible de construire un objet de classe **BufferedReader** à partir d'un objet de classe **InputStream** en faisant :

```
BufferedReader lectChaine = new BufferedReader(new InputStreamReader  
                                                    (objet_de_classe InputStream));
```

Remarque : toutes les méthodes permettant la lecture sur un flot peuvent lever une exception de classe **IOException** en cas d'erreur et **EOFException** s'il n'y a pas assez d'entrées pour terminer la lecture.

# Flots de sortie

- La classe **OutputStream** n'offre que la méthode **write** permettant d'écrire un ou plusieurs octets.
- La classe **PrintWriter** offre en plus les méthodes **print** et **println** qui connaissent les types primitifs et les chaînes de caractères.

Il est possible de construire un objet de classe **PrintWriter** à partir d'un objet de classe **OutputStream** en faisant :

**PrintWriter** ecriture = new **PrintWriter**(objet de classe **OutputStream**);

# URL et flots

Les objets de classe **URL** ont une méthode **openStream()** qui établit la connexion à cette URL et retourne un flot de classe **InputStream** permettant de récupérer l'information qu'elle contient.

En cas d'impossibilité une exception de classe **IOException** est levée.

Remarque : on ne peut **que lire** par le biais d'une URL

# Objets et flots

On peut utiliser des flots pour envoyer et recevoir des **objets** via des fichiers ou le réseau.

Pour cela on utilise les classes **ObjectOutputStream** et **ObjectInputStream**.

La création de tels objets à partir d'un flot existant se fait par leurs constructeurs :

**new ObjectOutputStream(OutputStream)**

**new ObjectInputStream(InputStream)**

Le transfert des objets fait alors appel aux méthodes :

- **Objet readObject()** pour la classe **ObjectInputStream**.  
Cette méthode peut lever les exceptions : **ClassNotFoundException**, **InvalidClassException**, **OptionalDataException** et **IOException**
- **void writeObject(Objet)** pour la classe **ObjectOutputStream**.  
Cette méthode peut lever l'exception **NotSerializableException** et **IOException**.



# Objets pouvant être lus/écrits dans des flots

- Il doivent implémenter l'interface **Serializable**
- *Presque toutes* les classes des bibliothèques de java implémentent **Serializable**
- Création d'une classe d'objets sérialisables :  
Class ObjetEnvoye implements **Serializable** {  
    private static final long serialVersionUID = 642400L;  
    ...  
}

Le numéro de série (**serialVersionUID**) permet de gérer des versions : erreur si n° de série différent

# Exemple de classe sérialisable

Class Envoi implements **Serializable** {

private static final long serialVersionUID = 1L; // n° de version

private int val; // propriété envoyée

private **transient** final int taille = 10; // propriété non envoyée

private Color coul; // propriété envoyée

public Envoi(int v, Color c) { // construction

val = v;

coul = c;

}

.....

}

# Ecrire des objets dans un flot

```
// on suppose disposer d'un flux de sortie par exemple connecté à un fichier  
// ce flux est de classe OutputStream et s'appelle flotVersFichier
```

```
ObjectOutputStream sortie = null; // création du flot pour écriture d'objets  
try { sortie = new ObjectOutputStream(flotVersFichier); }  
catch (IOException e1) { System.out.println("création du flux impossible"); }
```

```
Envoi m = new Envoi(5, new Color(0,100,200));  
try { sortie.writeObject(m); /* écriture d'un objet m */ }  
catch (NotSerializableException e2) { System.out.println("objet non sérialisable"); }  
catch (IOException e3) { System.out.println("écriture de l'objet impossible"); }
```

```
try {  
    sortie.flush(); // vidage du flot  
    sortie.close(); // fermeture du flot  
}  
catch (IOException e4) { System.out.println("fermeture du flux impossible"); }
```

# Lire des objets dans un flot

```
// on suppose disposer d'un flux d'entrée par exemple connecté à un fichier
// ce flux est de classe InputStream et s'appelle flotDuFichier
```

```
ObjectInputStream entree = null; // création du flot pour lecture d'objets
try { entree = new ObjectInputStream(flotDuFichier); }
catch (IOException e1) { System.out.println("création du flux impossible"); }
```

```
Envoi lu = null;
try { lu = (Envoi)entree.readObject(); /* lecture d'un objet et coercion */ }
catch (ClassNotFoundException e2) { System.out.println("classe de l'objet inconnue"); }
catch (InvalidClassException e3) { System.out.println("classe de l'objet incorrecte"); }
catch (OptionalDataException e4) { System.out.println("l'objet ne correspond pas à la classe"); }
catch (IOException e5) { System.out.println("lecture de l'objet impossible"); }
```

```
try { entree.close(); // fermeture du flot
}
catch (IOException e6) { System.out.println("fermeture du flux impossible"); }
```

# Parallélisme

# Threads

Java permet de lancer des processus en parallèle (threads)

- C'est le système d'exploitation qui gère l'enchaînement de ces processus
- Tout objet peut devenir un processus :
  - Soit en héritant de la classe **Thread**
  - Soit en implémentant l'interface **Runnable**
- Un processus peut être lancé à tout moment : il implémente une méthode **run** qui est exécutée lorsqu'il est lancé
- Un processus ne peut pas être arrêté : il doit se terminer en terminant sa méthode **run**

# Threads

## Création d'un Thread

```
class MonThread extends Thread {  
    public MonThread(...) { ... }  
    public void run() { ... }  
}
```

ou

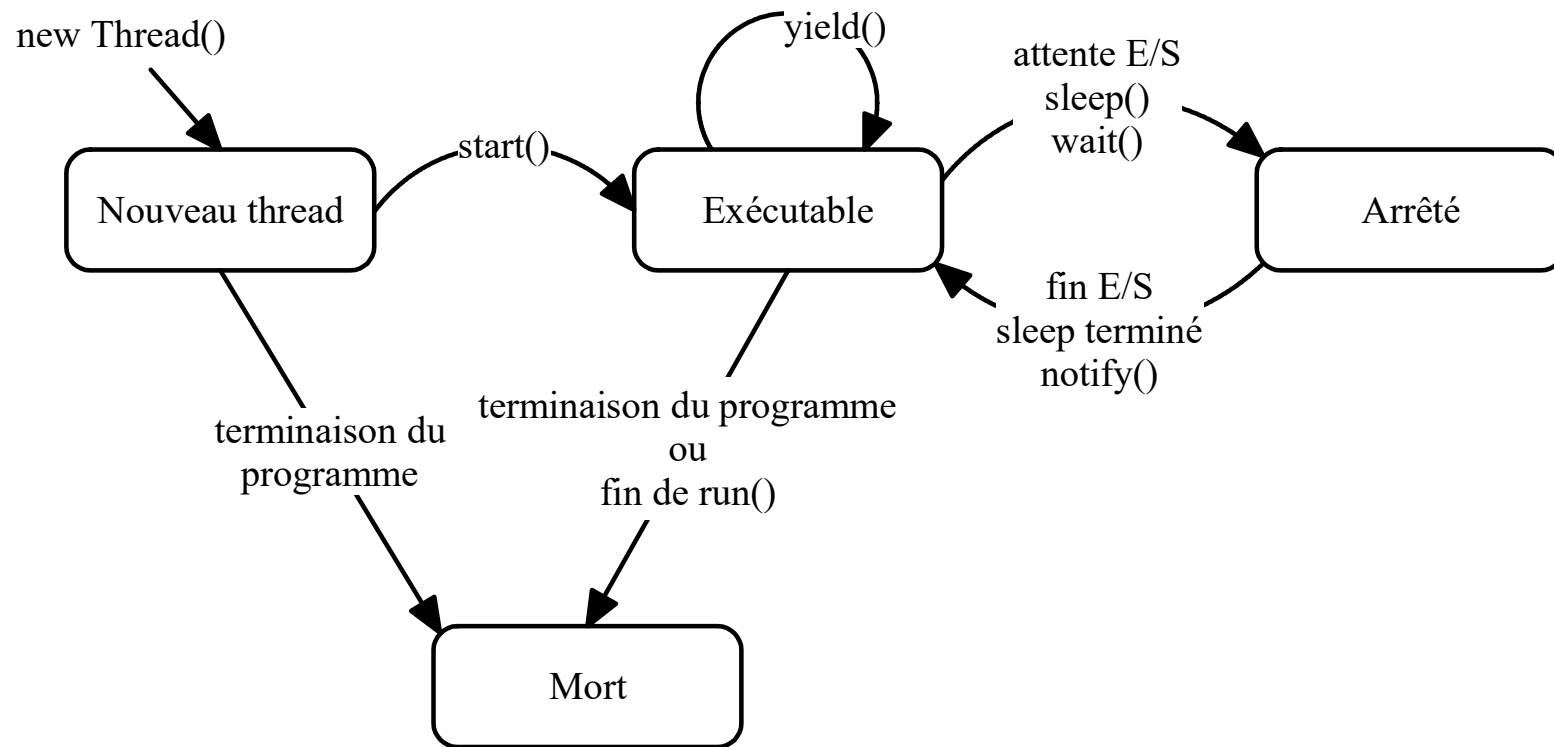
```
public class MonThread implements Runnable {  
    public MonThread(...) { ... }  
    public void run() { ... }  
}
```

## Démarrage d'un Thread

Créer le thread : *MonThread monTravail* = new *MonThread* (paramètres);

- Si le thread est fils de **Thread** il peut démarrer seul : *monTravail.start()*;
- S'il implémente **Runnable** il doit être inclus dans un thread de Java :
  1. Inclusion : *Thread celuiDeJava* = new *Thread*(*monTravail*);
  2. Démarrage du thread : *celuiDeJava.start()*;

# Etats d'un thread





# Bouton temporisé (thread)

```
import java.awt.*;

class Temporisateur extends JButton implements Runnable // Affiche un décompte du temps
{ // Affiche un décompte de temps
    private int temps;

    public Temporisateur(int t) { // construction du bouton avec une temporisation
        super(String.valueOf(t));
        temps=t;
    }

    public void reinitialiser(int t) { // Relancer la temporisation
        temps=t;
        setLabel(String.valueOf(temps));
    }

    public void arreter() { // Arrêter la temporisation => fin du thread
        reinitialiser(0);
    }

    public void run() { // Effectue le décompte de temps et l'affiche sur le bouton
        while (temps > 0) {
            try { Thread.sleep(1000); } catch (InterruptedException e) {};
            temps--;
            setLabel(String.valueOf(temps));
        }
        setEnabled(false); // le temps est passé => le bouton ne marche plus
    }
}
```

# Utilisation dans une application

```
public class Tempo extends JFrame {  
  
    private Temporisateur compteur;  
  
    public Tempo () {  
        super("Test de bouton temporisé");  
  
        FlowLayout placement=new FlowLayout(FlowLayout.LEFT,2,2);  
        getContentPane().setLayout(placement);  
  
        compteur =new Temporisateur (10);  
        getContentPane().add(compteur);  
  
        pack();  
        setVisible(true);  
  
        Thread lancer=new Thread(compteur);  
        lancer.start();  
    }  
  
    static public void main(String argv[]) {  
        new Tempo ();  
    }  
}
```

# Démarrage automatique

Il est parfois intéressant qu'un thread démarre dès sa création  
Dans l'exemple du bouton on le crée puis on le démarre.

Dans le constructeur de Temporisateur on met :

```
public Temporisateur(int t) {  
    super(String.valueOf(t));  
    temps=t;  
    new Thread(this).run(); // démarrage du bouton en tant que thread  
}
```

Dans l'application qui l'utilise on enlève le lancement

```
compteur =new Temporisateur (10); // il est créé et il démarre  
getContentPane().add(compteur);
```

```
pack();  
setVisible(true);
```

```
Thread lancer=new Thread(compteur);  
lancer.start();
```

# Arrêt d'un Thread

- Un thread se termine lorsque `run` se termine. Mais il arrive que l'on veuille écrire des threads qui ne se terminent pas (animation par exemple)
- La méthode `run` contient :  
`while(true) { ... }`
- Pour l'arrêter il faut le faire sortir de ce while
- Solution :  
`private volatile boolean marche; // volatile car modifié de l'extérieur`  
  
`void arret() { marche = false; } // arrêt du thread`  
  
`public void run() {  
 marche = true; // Au départ la boucle s'exécute  
 while(marche) { .... } // La boucle se termine si marche devient false  
}`
- L'appel de `arret()` terminera le thread dès la fin de la boucle

# Arrêt d'un Thread suspendu

- Si la boucle infinie du `run` contient des attentes (`sleep`) lorsque le booléen sera mis à faux le thread ne s'arrêtera qu'à la fin de l'attente
- Pour éviter cela et l'arrêter de suite :
  - Utiliser `interrupt()` qui lève une exception de classe `InterruptedException`
- Méthode d'arrêt :

```
void arret() {  
    marche = false;  
    interrupt();  
}
```
- Boucle du run :

```
while (marche) {  
    ...  
    try { sleep(x); }  
    catch (InterruptedException ie) { /* le sleep a été interrompu : fin du thread */ }  
    ...  
}
```

# Liens entre threads

- Communication entre threads  
Par le **réseau**, par des **flux** ou par des **objets partagés**
- Concurrency et synchronisation
  - Mot clé **synchronized** dans l'entête d'une méthode  
Tant qu'un thread exécute une méthode **synchronized** d'une classe A, **aucun autre thread** ne peut exécuter de méthode **synchronized** de cette classe A (la même méthode ou une autre). Les thread qui veulent le faire se mettent dans une **file d'attente associée à cette classe**.
  - **wait()** → **InterruptedException**  
**wait** doit se trouver dans une méthode **synchronized**. Il **suspend l'exécution du thread** qui se met dans la file d'attente associée à cette classe. Les méthodes **synchronized** de cette classe **sont à nouveau accessibles**.
  - **notify()** **Libère l'un des threads suspendus** par **wait** dans la file d'attente associée à cette classe
  - **notifyAll()** **Libère tous les threads suspendus** par **wait** dans la file d'attente associée à cette classe

# Taches retardées ou répétitives

- La classe **Timer** de java permet de lancer une tâche après un délai ou de façon répétitive.
- Création d'un timer :  
`Timer t = new Timer();`
- Lancement d'une tâche :
  1. Dans un délai  
`t.schedule(TimerTask, long);`
    - 2<sup>nd</sup> paramètre = délai en ms
  2. Répétitive  
`t.scheduleAtFixedRate(TimerTask, long, long)`
    - 2<sup>ème</sup> paramètre = délai de 1<sup>er</sup> lancement en ms
    - 3<sup>ème</sup> paramètre = période de répétition en ms

# Taches retardées ou répétitives

- La tâche :

```
class MaTache extends TimerTask {  
    ...  
    public void run() {  
        ...  
    }  
}
```

- Lancement :

```
t.schedule(new MaTache(), 1000);
```

Ou

```
t.scheduleAtFixedRate(new MaTache(), 1000, 1000);
```

- Arrêter un timer :

```
t.cancel();
```

Un timer arrêté ne peut pas être relancé : il faut le recréer (new)



# Communication entre programmes

# Communication par événements

Similaire aux événements d'interface et écouteurs mais les événements sont levés par des objets et pas par l'utilisateur :

- Un objet envoie des événements à d'autres objets par exemple lors de certains de ses changements d'état
- Principe :
  - Objets **observable** qui signalent des événements à des objets observateurs
  - Objets **observateur** qui reçoivent des événements d'objets observés
- Réalisation par :
  - Une classe (**Observable**)
  - Une interface (**Observer**)

# Exemple

Dans un jeu de rôle :

- Chaque objet qui gère un ennemi (Thread) est observable
- Lorsqu'il meurt il envoie un événement contenant ses caractéristiques et ses possessions puis disparaît.
- Les objets qui gèrent le score, l'expérience et les possessions du joueur sont observateurs de tous les ennemis
- Lors de la réception d'un événement (mort d'un ennemi) :
  - Le score et l'expérience augmentent selon les caractéristiques de l'ennemi
  - Les possessions s'enrichissent des celles de l'ennemi

# Classe Observable

- L'objet qui signale des événements appartient à une classe qui hérite de **Observable** donc des méthodes suivantes :
  - Gestion des observateurs :
    - void **addObserver**(Observer)
    - void **deleteObserver**(Observer)
    - void **deleteObservers**()
    - int **countObservers**()
  - Pour signaler un événement il utilise :
    - void **notifyObservers**(Object evt)

# Interface Observer

- Les objets qui reçoivent des événements doivent implémenter l'interface **Observer**
- Ils surchargent la méthode :
  - void **update**(Observable o, Object evt)
- Qui sera exécutée chaque fois qu'un événement leur arrive :
  - Le 1<sup>er</sup> paramètre est l'objet qui a envoyé l'événement
  - Le second paramètre est l'événement envoyé
- Pour recevoir les événements ils s'inscrivent auprès de l'objet observable par sa méthode **addObserver**
- Ils peuvent se désinscrire par sa méthode **deleteObserver**

# Ressources par URL

# URLs (Uniform Resource Locator)

- Forme écrite:

**protocole**://**machine**[:**port**]/**path**[#**partie**][**paramètres**]

**http**://**onesearch.sun.com**/**search/onesearch/index.jsp**?**qt=jmf&x=6&y=10**

- La classe **URL** :

URL(String)

URL(String, String, String)

String toString()

String getFile()

String.getHost()

int getPort()

String getProtocol()

}

**MalformedURLException**

```
URL chemin=null;
try {
    chemin = new
        URL("http://docs.oracle.com/javase/1.5.0/docs/api/");
    // utiliser l'URL chemin
}
catch (MalformedURLException mue) {
    // ce que l'on fait si chemin n'est pas une URL
}
```

# URL et flots

Les objets de classe **URL** ont une méthode **openStream()** qui établit la connexion à cette URL et retourne un flot de classe **InputStream** permettant de récupérer l'information qu'elle contient.

En cas d'impossibilité une exception de classe **IOException** est levée.

Remarque : on ne peut **que lire** par le biais d'une URL



# Ouvrir un flot sur une URL

```
// ouverture de l'URL à traiter (page HTML)
URL chemin;
try {
    chemin=new URL("http://www.ac-toulouse.fr/histgeo/branchem.htm");
}
catch (MalformedURLException mfe) {
    chemin=null; System.out.println("URL incorrecte");
}
// création d'un flot pour lire la page html
if (chemin != null) {
    BufferedReader lecture;
    try {
        lecture=new BufferedReader(new InputStreamReader(chemin.openStream()));
    }
    catch (IOException lec) {
        lecture=null;
        System.out.println("Ouverture du flux impossible");
    }
}
```

# Lire sur ce flot

```
if (lecture != null) {  
    // on va lire une page html ligne par ligne  
    String ligne; // une ligne lue dans le fichier  
    try {  
        while((ligne = lecture.readLine()) != null) {  
            System.out.println(ligne);  
        }  
    }  
    catch (IOException aff) {  
        System.out.println("Lecture du flot impossible");  
    }  
}
```

# Communication HTTP

# Classes pour Http

- La classe **URLConnection** :
  - void connect() throws IOException
  - InputStream getInputStream() throws IOException
  - OutputStream getOutputStream() throws IOException
  - URL getURL()
- La classe **HttpURLConnection** hérite de **URLConnection**
- Création
  - new URL(String) throws MalformedURLException
  - HttpURLConnection connecteur = (HttpURLConnection)  
url.openConnection() throws IOException

# *Protocole HTTP*

## *Requête HTTP*

2 types de messages :

- requêtes
- réponses

forme : <commande><entêtes>[<données>]

- **La commande** est de la forme : méthode<esp>URL<esp>version<rc>  
méthode = GET ou HEAD ou POST  
URL désigne la ressource  
version = version du protocole HTTP
- **Les entêtes** contiennent des champs et leur valeur comme :  
Connexion : close  
User-Agent : mozilla/4.0  
etc
- **Les données** n'existent que pour POST.

# *Exemples d'applications de l'Internet*

## Réponse HTTP

forme : <état><entêtes><données>

- **L'état** est de la forme : version<esp>code<esp>message<rc>  
version = version du protocole HTTP  
code = code d'erreur (200=OK, 404=not found ...)  
message = détail de l'erreur
- **Les entêtes** contiennent des champs et leur valeur comme :  
Content-Type : **type mime** (détaillé pas la suite)  
Content-Length : taille des données  
etc
- **Les données** contiennent par exemple une page HTML, une image etc. selon le type mime précisé.

# Classe **URLConnection**

- Accès aux informations réponse HTTP :
  - int getLength()
  - String getHeaderField(int n)
  - int getResponseCode() throws IOException
  - String getResponseMessage() throws IOException
- Accès au contenu par flux :

```
BufferedReader lecture = new BufferedReader(new  
InputStreamReader(connecteur.getInputStream()));  
String lecture.readLine();
```

# Communication client / serveur



# Communication par réseau

- Communication par sockets (client/serveur)
  - Sockets pour serveur et pour client : **Socket** et **ServerSocket**
  - Liées à TCP/IP
  - Les sockets sont liées à des flux d'entrée et de sortie (**InputStream** et **OutputStream**)
- Sérialisation d'objets  
**ObjectInputStream** et **ObjectOutputStream** créés à partir de ceux de la socket

# Dialogue par sockets

- Il existe 2 classes **Socket** et **ServerSocket**, la première est utilisée par le client, la seconde par le serveur.
- Le serveur crée une **ServerSocket** et attend les connexions des clients
- Le client crée une **Socket** et se connecte au serveur.
- Quand une connexion est acceptée le serveur crée une **Socket** pour dialoguer avec le client
- Le dialogue s'établit par des flots qui peuvent être des flots d'objets

# ServerSocket

- `ServerSocket(int, int)` : numéro de port et taille de la file d'attente associée à cette socket d'écoute. Peut lever une exception de classe `IOException` (erreur de création) et `SecurityException` (création non autorisée).
- `void close()` : fermeture de la socket.
- `Socket accept()` : attend une demande de connexion. Lorsqu'elle arrive, elle est acceptée et une nouvelle `Socket` est créée qui permet le dialogue avec le client (*par un thread* => multi-client).  
Peut lever une exception de classe `IOException` ou `SecurityException`.

# Socket

- `Socket(String, int)` : nom de machine ou adresse IP et numéro de port.

Peut lever une exception de classe `IOException` (erreur de création) et `UnknownHostException` (connexion à l'hôte impossible).

- `void close()` : fermeture de la socket.
- `InputStream getInputStream()` : retourne un flux pour lire sur la socket. Peut lever une exception de classe `IOException` si le flux ne peut pas être créé.
- `OutputStream getOutputStream()` : retourne un flux pour écrire sur la socket. Peut lever une exception de classe `IOException` si le flux ne peut pas être créé.

# Serveur : squelette

```
public class Serveur {  
  
    private final int PORT = 45678; // numéro de port  
    protected ServerSocket socketServeur = null;  
    protected Socket socketClient = null;  
  
    public Serveur() {  
        try {  
            // crée une socket de type socketServeur rattachée au port,  
            socketServeur = new ServerSocket(PORT);  
        }  
        catch(IOException e) {  
            System.err.println("Erreur de création de socket d'écoute : " + e);  
            e.printStackTrace();  
            System.exit(1);  
        }  
        System.out.println("SocketServeur: a l'ecoute sur le port " + socketServeur.getLocalPort());  
        service();  
    }  
}
```

# Serveur : appel et création des flux

```
void service() {
    try {
        while(true) { // il ne reste qu'à attendre que quelqu'un veuille bien nous parler
            socketClient = socketServeur.accept();
            System.out.println("Serveur: connexion etablie avec le client : " + socketClient.getInetAddress());

            // création des flux pour les échanges de données avec le client
            ObjectInputStream lire=null;
            ObjectOutputStream ecrire=null;
            try { // associe un flot d'objets en entrée et en sortie à la socket
                lire = new ObjectInputStream(socketClient.getInputStream());
                ecrire = new ObjectOutputStream(socketClient.getOutputStream());
                // Traitement de la demande du client
            }
            catch(IOException e) {
                System.err.println("Erreur de création de flux : " + e);
                try {socketClient.close();} catch(IOException e2) {}
            }
        }
        catch(IOException e) {
            System.err.println("Erreur serveur : " + e);
            e.printStackTrace();
        }
        try {socketClient.close();} catch(IOException e) {}
        System.out.println("Serveur: connexion terminée");
    }
}
```

# Serveur : traitement

```
try {  
    // Attente de la demande sur le flot d'entrée (String)  
    Question quest = (Question )lire.readObject(); // Lire la question  
    // calculer la réponse  
    ecrire.writeObject(new Reponse(texte)); // Répondre au client  
    ecrire.flush();  
}  
catch(ClassNotFoundException cnfe) {  
    System.err.println("Erreur de service (classe) : " + cnfe);  
}  
catch(IOException e) {  
    System.err.println("Erreur de service : " + e);  
}  
try {  
    lire.close(); ecrire.close();  
    socketClient.close();  
}  
catch(IOException e) { System.err.println("Erreur de fermeture de socket : " + e) }
```

# Client : connexion

```
Socket socketClient = null;
try {
    socketClient = new Socket(HOST, PORT);
    // Communiquer avec le serveur
}
catch (UnknownHostException uhe) {
    System.err.println("Hote inconnu : "+uhe);
}
catch (IOException io1) {
    System.err.println("Erreur de création de socket client : "+io1);
}
System.out.println("Client: connexion établie sur " +
socketClient.getInetAddress() + "(port = " + socketClient.getPort() + ")");
```



# Client : échange avec le serveur

```
ObjectInputStream lire=null;
ObjectOutputStream ecrire=null;
try {    // associe un flot d'objets en entrée et en sortie à la socket
    ecrire = new ObjectOutputStream(socketClient.getOutputStream());
    lire = new ObjectInputStream(socketClient.getInputStream());
}
catch(IOException e) {
    System.err.println("Erreur de création de flux : " + e);
    try {socketClient.close();}
    catch(IOException e2) { System.err.println("Erreur de fermeture de connexion"); }
}
try {
    ecrire.writeObject(new Question(...)); // Envoi de la demande au serveur
    ecrire.flush();
    Reponse reponse = (Reponse)lire.readObject(); // Attente de la réponse fabriqué par le serveur
    // Traiter la réponse
}
catch(ClassNotFoundException cnfe) {
    System.err.println("Erreur de service (classe non trouvée) : " + cnfe);
}
catch(IOException e) {
    System.err.println("Erreur de service : " + e);
}
try{// Connexion terminée
    lire.close(); ecrire.close();
    socketClient.close();
} catch(IOException e){ System.err.println("Erreur de fermeture de connexion"); }
```

# Serveur multi clients

- Pour qu'un serveur puisse accepter plusieurs clients simultanés il suffit que le traitement de la demande du client soit fait dans un Thread
- Quand il a fait le `accept()`, le serveur crée un Thread auquel il passe la `Socket` renvoyée par le `accept()`
- Ce Thread assure la totalité de l'échange avec le client puis ferme cette `Socket`
- Pour éviter que des Threads ne restent en attente indéfiniment si le client n'envoie rien on peut utiliser un `Timer` pour couper la connexion après un délai

# RMI (Remote method Invocation)

Idée :

- Sur la machine M1 on a un objet  $a$  de classe  $A$  qui a besoin d'appeler une méthode  $m$  d'un objet de classe  $B$ .
- Normalement il crée un objet de classe  $B$  et appelle la méthode  $m$
- Mais si on n'a pas le code de la classe  $B$  sur la machine M1 **on ne peut pas le faire**
- RMI permet à l'objet  $a$  de M1 (client) d'appeler la méthode  $m$  d'un objet  $b$  de classe  $B$  **qui a été créé sur une autre machine** M2 (serveur).
- Cet objet  $b$  a été créé sur M2 dans ce but (service distant)

# RMI (principe)

- Sur la machine M1 on n'a pas la classe *B* mais une interface *iB* qui contient les entêtes des méthodes de *B* appelables à distance (dont *m*)
- L'objet *a* utilise donc cette interface *iB*
- Sur M1 RMI transforme l'appel de *m* en :
  - Connexion sur la machine M2
  - Envoi des paramètres de *m* par sérialisation
  - Récupération du résultat de *m* par sérialisation
  - Renvoi de ce résultat à l'objet *a* en retour de l'appel de *m*
- Sur M2 RMI :
  - Récupère les paramètres envoyés par sérialisation par M1
  - Exécute la méthode *m* de l'objet *b*
  - Renvoie le résultat à M1 par sérialisation

# RMI localisation du service

- Pour que tout ce mécanisme marche il faut que M1 sache que l'objet recherché *b* est sur M2 et puisse l'identifier.
- Pour identifier l'objet recherché RMI utilise un **annuaire** d'enregistrement de service créé sur M2. **Chaque service a un nom unique sur M2.**
- L'annuaire est créé sur M2 par le serveur
- Quand *b* est créé sur M2 il est enregistré (avec un nom) sur cet annuaire (nom de service)
- Quand M1 recherche l'objet *b* elle se connecte à M2 pour consulter cet **annuaire** et, **avec ce nom**, récupérer l'objet *b*

# Code java

- Serveur

// création de l'annuaire

Registry annuaire = LocateRegistry.createRegistry(PORT);

// création de l'objet qui assure le service

B serv = new B(...);

// Enregistrement du service dans l'annuaire

annuaire.bind(serv, "Nom");

- Client

// connexion à l'annuaire

Registry annuaire = LocateRegistry.getRegistry(nom\_hote, PORT);

// récupération de l'objet qui assure le service

iB serv = (iB) annuaire.lookup("Nom ");

// Le client utilise serv comme s'il l'avait créé lui-même

# Services

- RMI permet la mise en place de services
- Un service est un objet qui offre une ou plusieurs méthodes
- Si cet objet a des propriétés (état) ces propriétés sont conservées puisqu'il est sur le serveur :
  - Un autre client utilisant ce service va le trouver dans l'état où le client précédant l'a laissé (**danger**)
- Soit on crée des services sans état
- Soit on fait en sorte que l'état soit inaccessible ou inutilisable (par exemple crypté) ou sans intérêt
- Soit le service offre une méthode **d'effacement de son état** que le client appelle avant de partir (sauf s'il ne le fait pas ! **danger**)

# Applications sous contrôle



# Les Applets

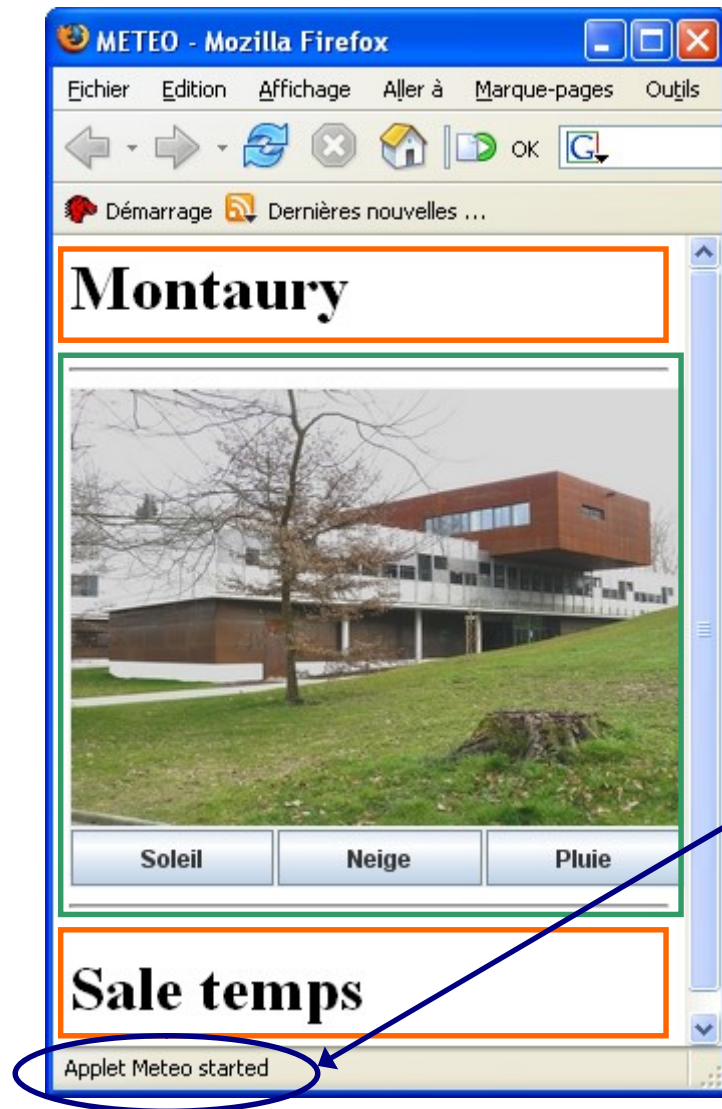
Petite application accessible sur un serveur Internet, qui se transporte sur le réseau, s'installe automatiquement et s'exécute in situ comme partie d'un document web.

Zones réservées au navigateur

Zone réservée à l'applet

L'applet est lancée

Démo



# ***STRUCTURE D'UNE PAGE HTML***

- Les étiquettes (ordres ou directives) peuvent être ouvertes ou fermées. Fermées, elles ont un début et une fin.  
Par exemple: *<CENTER> Titre de ma page </CENTER>*
- Une étiquette peut avoir des paramètres.  
Par exemple: l'étiquette *<BODY BGCOLOR="#FF00FF"> </BODY>* indique la couleur du fond.
- La structure d'une page est la suivante:  
*<HTML>*  
*<HEAD>* Entête de page  
*<TITLE>* Titre *</TITLE>*  
*</HEAD>*  
*<BODY>*  
Corps de la page. Ce que l'on voit dans le navigateur.  
*</BODY>*  
*</HTML>*

# La Balise Applet

Mettre une applet dans une page HTML

```
<HTML>
<HEAD>
  <TITLE> essai d'applet </TITLE>
</HEAD>
<BODY>
  ...
```

```
  <APPLET CODE="Lange.class" CODEBASE="applets"
    WIDTH="100%" HEIGHT=200 >
```

```
    <PARAM NAME="Couleur" VALUE="rouge">
    <PARAM NAME="Vitesse" VALUE="50">
```

```
  </APPLET>
```

```
  ...
</BODY>
</HTML>
```



Paramètres de l'applet.

# La Balise Object

Depuis la version 4 de HTML il est préférable d'utiliser la balise **object** plutôt que la balise APPLET. Ceci se fait comme suit:

**<object**

codetype="application/java-archive"

classid="java:Langue.class"

codebase="applets"

width="100%" height="200" align="middle"

standby="texte affiché pendant que l'applet se charge"

>

<PARAM NAME="Couleur" VALUE="rouge">

<PARAM NAME="Vitesse" VALUE="50">

Paramètres de l'applet.

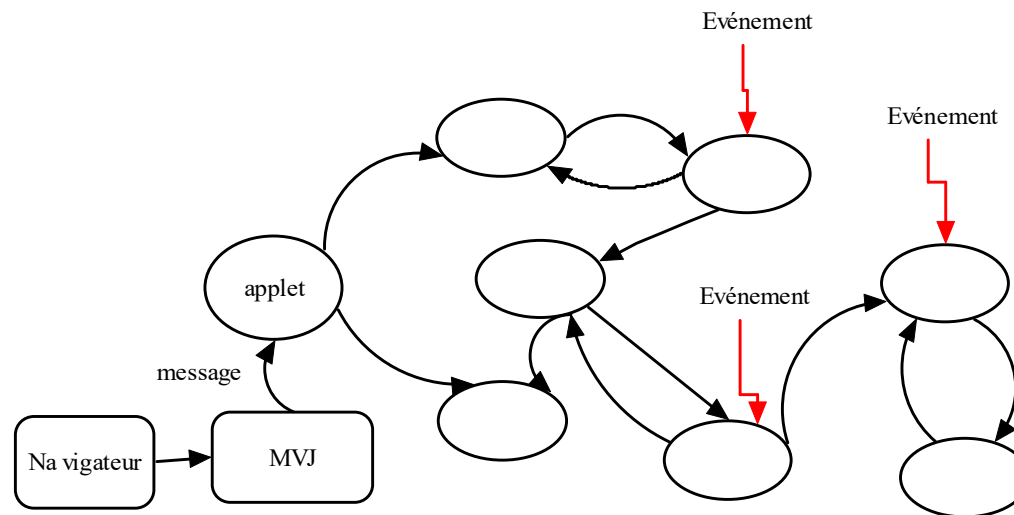
On peut mettre ici un texte qui sera affiché si l'applet ne démarre pas

**</object>**



# Exécution d'une Applet

- Une **applet** : s'exécute sous le contrôle d'un **navigateur**. Le navigateur appelle, selon ses besoins, des méthodes de l'applet.
- Les événements sont directement transmis aux objets de l'interface (écouteurs d'événements)



JApplet

# Sécurité

- Une applet ne peut pas accéder aux fichiers locaux
- Une applet ne peut faire aucune connexion réseau hormis sur l'URL à laquelle elle a été trouvée (ressources distantes)
- L'applet est arrêtée dès que la page qui la contient est fermée
- Mais une applet peut charger le processeur (threads)

# Écrire une applet

- La bibliothèque `javax.swing` contient une classe **JApplet**
- La classe JApplet sert de **classe mère** pour toutes les applets
- La classe JApplet a des **méthodes que le navigateur connaît et appelle**
- Écrire une applet c'est écrire une **classe fille de JApplet** dans laquelle on **surcharge** certaines méthodes
- Cette classe doit être créée avec le modificateur **public** pour que le navigateur puisse l'utiliser
- Le fichier **.class** doit être **accessible** par le navigateur client (droits).

# Cycle de vie d'une applet

*Une applet s'exécute dans une page web. Cet environnement va déterminer le cycle de vie de l'applet:*

- Quand il se crée une instance de la classe qui contrôle l'applet  
L'applet s'initialise : **void init()**
- Quand on quitte la page, par exemple, pour aller à la suivante, l'applet arrête de s'exécuter : **void stop()** .
- Quand on revient à la page qui contient l'applet, elle se ré-exécute :  
**void start()** .
- Quand le navigateur doit afficher l'applet il appelle la méthode :  
**void paint(Graphics)**
- Finalement, quand se termine l'exécution du navigateur, ou de l'application qui visualise l'applet, l'exécution de l'applet se termine et toute la mémoire et les ressources occupées par l'applet sont libérées avant de quitter le navigateur : **void destroy()** .



# Relations d'une applet avec l'environnement

## -1- Apparence

- void paint(Graphics)
- void repaint()
- void setForeground(Color)
- void setBackground(Color)
- int getWidth (). et int getHeight().
- *interface avec des composants d'interface*

# Relations d'une applet avec l'environnement

## -2- Navigateur et Réseau

- URL `getDocumentBase()` retourne l'URL de la page
- URL `getCodeBase()` retourne l'URL du répertoire de l'applet
- String `getParameter(String)` retourne le contenu (String) du paramètre de l'applet (balise `<PARAM ...>`) dont le nom est contenu dans le paramètre de cette méthode.
- AppletContext `getAppletContext()` retourne le contexte de l'applet.

`getAppletContext().showDocument(URL, String)`

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

// Acces à une page externe (ici la doc en ligne de java)
public class AfficherPageWeb extends JApplet {

    public void init() {
        URL page; // url où on veut aller (page externe)
        try { page=new URL("http://docs.oracle.com/javase/1.5.0/docs/api/");
            getAppletContext().showDocument(page, "docJava"); // dans une autre fenêtre
        }
        catch (MalformedURLException mue) {
            page=null;
            System.out.println("L'URL n'est pas correcte"); // affichage dans la console java
        }
    }
}
```

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

**// Acces à une page locale dans la même fenêtre**

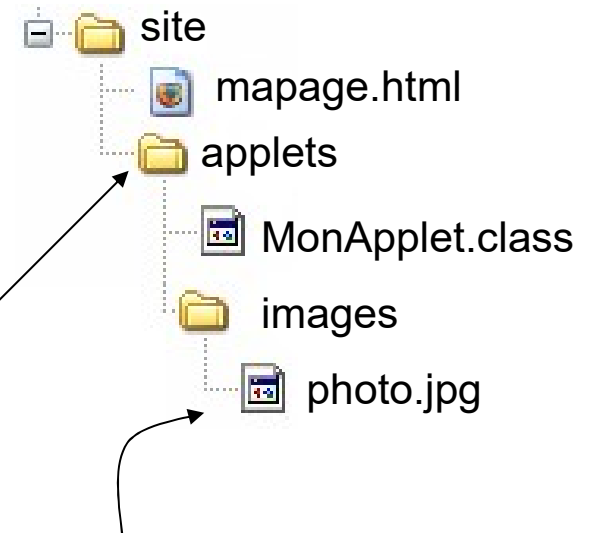
```
public class RemplacerPage extends JApplet {  
  
    public void init() {  
        try { Thread.sleep(5000); } // petit délai pour avoir le temps de voir quelque chose  
        catch (InterruptedException ie) {}  
        // Changement de page (remplace la page actuelle)  
        URL page; // url où on veut aller (page locale au serveur)  
        URL cheminDeLApplet = getCodeBase();  
        try {    page=new URL(cheminDeLApplet, "../autrepage.html");  
                getAppletContext().showDocument(page, "_self"); // remplace la page actuelle  
            }  
        catch (MalformedURLException mue) {  
            page=null;  
            System.out.println("L'URL n'est pas correcte"); // affichage dans la console java  
        }  
    }  
}
```

# Relations d'une applet avec l'environnement

## - 3- images et sons

Récupérer une image :

```
ImageIcon img;  
URL ressourceImage = null;  
try {  
    ressourceImage = new URL(getCodeBase(), "images/photo.jpg");  
    img = new ImageIcon(ressourceImage);  
}  
catch (MalformedURLException mfe) {  
    img = null;  
}
```



# Relations d'une applet avec l'environnement

## - 3- images et sons

- Récupérer un son :

```
AudioClip son;
```

```
son = getAudioClip(getCodeBase(), "sons/bruit.wav");
```

on jouera le son par : `son.play()`

- Jouer directement un son :

```
play(getCodeBase(), "sons/bruit.wav")
```

# Placement des composants d'interface



Comme avec swing

# Traitement des événements

Quand l'applet est créée il ne reste plus qu'à traiter les **événements** de l'utilisateur sur les divers composants de l'interface.

Pour cela on utilisera des **écouteurs d'événements** associés à ces composants d'interface comme avec swing





# Introspection (classe)

Possibilité pour un programme d'explorer une classe :

- Savoir de quoi elle hérite (classe et interfaces)
- Accéder à ses méthodes (paramètres / retour / invocation)
- Accéder à ses constructeurs (paramètres)
- Accéder à ses propriétés (récupérer / modifier)

Utilise la classe **Class**

- Obtention d'un objet de classe **Class** à partir d'un objet x par :  
`x.getClass()` ou `x.class`
- Obtention d'un objet de classe **Class** à partir d'un type primitif x par :  
`x.class`
- Obtention d'un objet de classe **Class** à partir d'un nom de classe par :  
`Class.forName(String nom)` throws **ClassNotFoundException**

# Introspection (héritage)

Méthodes de la classe **Class** :

- Récupération du nom de la classe :

`String getName()`

- Récupération des interfaces qu'implémente la classe :

`Class[] getInterfaces()`

- Récupération de la classe dont hérite la classe :

`Class getSuperClass()`

- Savoir si un objet de la classe donnée en paramètre peut être assigné à la classe (càd est de même classe ou de classe fille) :

`boolean isAssignableFrom(Class c)`

# Introspection (méthodes)

Méthodes de la classe **Class** :

- Récupération des constructeurs de la classe :

`Constructor[] getConstructors()`

- Récupération du constructeur de la classe acceptant les classes de paramètres donnés :

`Constructor getConstructor(Class p1, ... , Class pn) throws  
NoSuchMethodException`

- Récupération des méthodes de la classe :

`Method[] getMethods()`

- Récupération de la méthode de la classe correspondant au nom donné en 1<sup>er</sup> paramètre et aux classes de paramètres donnés :

`Method getMethod(String nom, Class p1, ... , Class pn) throws  
NoSuchMethodException`

# Introspection (constructeur)

- Méthodes de la classe **Constructor**
  - Récupération des classes des paramètres du constructeur :  
`Class[] getParameterTypes ()`
  - Récupération des classes des exceptions que peut lever le constructeur :  
`Class[] getExceptionTypes ()`
  - Création d'une instance d'objet en utilisant ce constructeur avec passage des paramètres de construction :  
`Object newInstance(Object p1, ... Object pn)` throws **InstantiationException**,  
**IllegalAccessException**, **IllegalArgumentException**, **InvocationTargetException**
    - **InstantiationException** – si la classe est abstraite.
    - **IllegalAccessException** – si le constructeur est inaccessible.
    - **IllegalArgumentException** – si le nombre ou les types des paramètres ne correspondent pas.
    - **InvocationTargetException** – si le constructeur lève une exception.

# Introspection (méthodes)

- Méthodes de la classe **Method**
  - Récupération des classes des paramètres de la méthode :  
`Class[] getParameterTypes ()`
  - Récupération de la classe de la valeur retournée par la méthode ou **Void.TYPE** :  
`Class getReturnType()`
  - Récupération des classes des exceptions que peut lever la méthode :  
`Class[] getExceptionTypes ()`
  - Appel d'une méthode, le 1<sup>er</sup> paramètre est l'objet dont on appelle la méthode, les suivants sont les paramètres passés à cette méthode (objets ou types primitifs) :  
`Object invoke(Object obj, Object p1, ... Object pn) throws IllegalAccessException,  
IllegalArgumentException, InvocationTargetException`
    - **IllegalAccessException** – si la méthode est inaccessible.
    - **IllegalArgumentException** – si le nombre ou les types des paramètres ne correspondent pas.
    - **InvocationTargetException** – si la méthode lève une exception.

# Introspection (propriétés)

Méthodes de la classe **Class** :

- Récupération des propriétés de la classe :

`Field[] getDeclaredFields()`

- Récupération de la propriété de la classe correspondant au nom donné en paramètre :

`Field getDeclaredField(String) throws NoSuchFieldException`

# Introspection (propriétés)

- Méthodes de la classe **Field**
  - Récupération du nom de la propriété :  
`String getName()`
  - Récupération du contenu de la propriété (le paramètre est l'objet dont on récupère la propriété) :
    - Si c'est un objet :  
`Object get(Object obj)`
    - Si c'est in type primitif :  
`getBoolean(Object obj), getInt(Object obj) ...`

Ces méthodes peuvent lever `throws IllegalAccessException, IllegalArgumentException`

- Modification du contenu de la propriété (le 1<sup>er</sup> paramètre est l'objet dont on récupère la propriété, le second est la valeur qu'on lui affecte) :
  - Si c'est un objet :  
`set(Object obj, Object value)`
  - Si c'est in type primitif :  
`setBoolean(Object obj, boolean v), setInt(Object v, int i) ...`

Ces méthodes peuvent lever `throws IllegalAccessException, IllegalArgumentException`



# Introspection

Méthodes de la classe **Class** :

- Récupération de l'objet ayant servi à charger la classe en mémoire :  
`ClassLoader getClassLoader()`

*Remarque : ClassLoader permet de charger du code provenant d'un fichier ou du réseau, c'est ce qu'utilise la MV java.*

- Création d'une instance d'objet de la classe en utilisant le constructeur par défaut :

`Object newInstance()` throws **InstantiationException**, **IllegalAccessException**

**IllegalAccessException** – s'il n'y a pas de constructeur sans paramètres

**InstantiationException** – si la classe est abstraite.

