PAQUETAGES STANDARDS

Paquetages standards

- `java.lang` : bases
- `java.io` : entrée-sorties
- `java.util` : utilitaires, structures de données
- `java.text` : internationalisation
- `java.awt` : graphisme (Abstract Window Toolkit)
- `java.applet` : applications
- `java.beans` : composants
- `java.rmi` : objets distribués (Remote Method Invocation)
- `java.net` : réseau
- `java.math` : précision arbitraire, ...
- `java.sql` : bases de données (JDBC)
- `java.security` : cryptage, authentification, ⋯
- ...

Paquetage `java.lang`

- des classes de base : `String, StringBuffer, System, Thread, Object` ...
- des classes permettant d'encapsuler les types élémentaires dans des objets (classes `Integer`, `Float`, ...)
- la librairie mathématique (classe `Math`)

Note : ce paquetage est "importé" automatiquement

Classe `String`

- constructeurs : `String()`, `String(String)`, ...
- longueur : `length()`
- accès à un caractère : `char charAt(int)`
- recherche dans la chaine :
  `int indexOf(char), int indexOf(String)`
- comparaison de contenu (ordre lexicographique) :
  `int compareTo(String)`

Classe `String` (2)

- test égalité de contenu :
  `boolean equals(Object)`
- test égalité partielle :
  `boolean regionMatches(int start, String other, int oStart, int len)`
- test de début/fin de la chaîne :
  `boolean startsWith(String)`,
  + `boolean endsWith(String)`,...
- concaténation : `String concat(String)`
  (renvoie nouvelle chaîne avec le résultat)

Classe `String` (3)

- sous-chaîne : `String substring(int deb, int fin)`
  `void getChars(int deb, int fin,char[] dest, int destDeb)`
- suppression des blancs de début et fin :
  `String trim()`
- changement de casse :
  `String toLowerCase(),...`
- substitution de caractère :
  `String replace(char old, char new)`
- conversion en chaîne des types de base : méthodes statiques
  `String.valueOf(int), String.valueOf(float),...`

Classe `StringBuffer`

- constructeurs : `StringBuffer()`, `StringBuffer(String)`, `StringBuffer(int)`, ...
- concaténation : `StringBuffer append(String)`, `StringBuffer append(int)`, ...
- modification de caractère : `StringBuffer setCharAt(int index, char c)`
- insertion : `StringBuffer insert(int index, char c)`, ...
- suppression : `deleteCharAt(int index)`, `delete(int debut, int fin)`

Classe `StringBuffer` (2)

- troncature : `void setLength(int)`
- miroir : `StringBuffer reverse()`
- conversion en String : `String toString()`
- capacité : `int capacity(), ...`
- comme String : `charAt(int)`, `substring()`, `getChars()`

Classe Racine (`Object`)

- ancêtre de toutes les classes
- comparaison : `public boolean equals(Object obj)`
- conversion en chaîne :
  `public String toString()`
- duplication : `protected Object Clone()`
- code de hachage : `public int hashCode()`

Classe Racine (`Object`) (2)

- destruction : `void finalize()`
- synchronisation threads :
  `notify(), notifyAll(), wait()`
- classe instanciée :
  `public Class getClass()`

Classe `Class`

- représentation des classes et interfaces (et tableaux)
- obtention via une instance (`o.getClass()`), ou par le nom : `Class.forName("NomTotalClasse")`
- littéraux : `NomClasse.class`
  (ex: `String.class` désigne la classe `String`)
- instantiation : `Object newInstance()`

Classe `Class` (2)

- récupération des membres de la classe
  - `Method[] getDeclaredMethods()`
  - méthodes publiques : `Method[] getMethods()`
  - `Field[] getDeclaredFields(), getFields()`
  - `Constructors[] getDeclaredConstructors(), getConstructors()`
  - `Class[] getInterfaces()`
  - `Class getSuperClass()`
  - classes internes : `Class[] getClasses()`
- recherche d'un membre donné :
  `Method getMethod(String),`
  `Field getField(String),...`

Classe `Class` (3)

- informations sur la classe :
  - nom **complet** : `String getName()`
  - `toString()` : idem, précédé de "class" ou "interface" selon les cas
  - `boolean isInterface()`
  - `boolean isArray()`
  - `boolean isPrimitive()`
  - `int getModifiers()` : masque de la forme `Modifier.PUBLIC|Modifier.STATIC`...
  - type des éléments (pour tableaux seulement) : `Class getComponentType()`
- tester l'appartenance d'un objet à la classe (`instanceof` dynamique) : `boolean isInstance(Object)`

Classe `Field`

- représentation des attributs
- implémente l'interface `Member` :
  - accès au nom : `String getName()`
  - nature : `int getModifiers()`
  - `Class getDeclaringClass()`
- type : `Class getType()`
  (renvoie instances spéciales int.class, float.class, boolean.class,... si type primitif)
- accès/modification valeur :
  - `void set(Object instance,Object valeur)`
  - `void Object get(Object instance)`

Classe `Method`

- représentation des méthodes des classes
- implémente `Member` (cf. `Field`)
- type des paramètres :
  `Class[] getParameterType()`
- type de retour :
  `Class getReturnType()`
- invocation :
  `Object invoke(Object instance, Object[] args)`

Classes `Integer`, `Float`, ...

- pour chaque type élémentaire (`boolean, char, int, float`, ...), il existe une classe "enveloppe" correspondante (`Boolean, Character, Integer, Float`, ...) pour pouvoir manipuler si nécessaire des valeurs élémentaires comme des objets
- elles ont toutes :
  - un constructeur à partir d'une valeur du type primitif correspondant
  - un constructeur à partir de `String`, et une méthode statique équivalente `NomClasse.valueOf(String)`
  - une méthode `String toString()`
  - un attribut `TYPE` avec l'objet `Class` correspondant

Classe `Number` et classes filles

- classe abstraite mère des classes `Byte, Integer, Float`, …
- dans chaque classe fille :
  - valeurs min/max du type : constantes de classe `MIN_VALUE` et `MAX_VALUE`
  - méthodes de conversion : `byte byteValue(), int intValue(), float floatValue()`, …
  - conversion en chaîne du type élémentaire : méthode statique `String toString(`*`type`*`)`

Classes `Integer`, `Byte`, `Short`, `Long`

- presque totalement similaires
- évaluer un entier écrit sous forme de chaîne de caractères : `int Integer.parseInt(String), byte Byte.parseByte(String)`, …
- idem en précisant une base : `int Integer.parseInt(String s,int base)`
- écriture en base 2 à 36 : `String Integer.toString(int val,int base)`
- ...

Classes `Float` et `Double`

- totalement similaires
- infini :
  `float Float.POSITIVE_INFINITY,`
  `float Float.NEGATIVE_INFINITY,`
  `boolean Float.isInfinite(float)`
- indéterminé :
  `boolean Float.isNaN(float)`

Classe `Character`

- ne contient quasiment que des méthodes statiques
- `boolean Character.isLetter(char),`
  `Character.isDigit(char),`
  `Character.isWhiteSpace(char),` …
- type de caractère :
  `int Character.getType(char ch)` renvoie un identifiant parmi : `Character.LOWERCASE_LETTER,`
  `Character.DECIMAL_DIGIT_NUMBER,` …
- changement de casse :
  `char Character.toLowerCase(char),` …

Classes `Throwable`, `Exception`, ...

- classe mère `Throwable` :
  - `Throwable(String)` : constructeur avec message explicatif sur la cause de l'exception
  - `String getMessage()` : renvoie le message explicatif en question
  - `void printStackTrace()` : affiche la pile d'appel jusqu'au point de lancement de l'exception
- rien de plus dans les classes filles `Exception`, `RuntimeException`, ... mais toutes ont un constructeur de la forme :
  - `NomException(String)` : constructeur avec message explicatif sur la cause de l'exception

Classe `System`

- entrée-sorties standards :
  `InputStream System.in`,
  `PrintStream System.out` et `System.err`
- redirection des I/O : `System.setIn(InputStream)`, ...
- lancement forcé du GC : `System.gc()`
- fin d'exécution : `System.exit(int status)`
- accès aux variables d'environnement :

  `String System.getProperty(Sring name)`...

Classe `System` (2)

- réglage sécurité :
  `System.setSecurityManager(SecurityManager)`
- heure (en ms depuis 1/1/70) :
  `long System.currentTimeMillis()`
- copie rapide de tableau
  `System.arraycopy(src,deb,dest,debD,len)`

Classe `Runtime`

- classe instanciée 1 fois (et une seule) dans chaque programme Java s'exécutant
- récupération de l'instance courante : `Runtime`
  `Runtime.getRuntime()`
- lancement de programme externe (dans un process séparé) :
  `Process exec(String commande)`…
  exemple :
  `Runtime.getRuntime().exec("ls -l *.c");`
- bilan mémoire :
  `long freeMemory()` et `totalMemory()`

Classe `Process`

- classe abstraite permettant de gérer les programmes externes lancés en process séparés par `exec()` de `Runtime`
- attente de fin et récupération du status :
  `int waitFor()`
- suppression du process (kill) : `destroy()`
- connexion avec les I/O standard :
  - `InputStream getInputStream()` récupère la sortie standard pour lire dessus
  - `OutputStream getOutputStream()` permet de se connecter à l'entrée standard du process pour écrire des choses dessus

Classe `Thread`

- classe pour gérer les "sous-processus" internes à un programme Java
- constructeurs : `Thread(String)`, `Thread(Runnable)`
- thread courant : `Thread.currentThread()`
- démarrage : `start()`
- envoyer message : `interrupt()` => `isInterrupted()` devient `true` (à exploiter dans méthode `run`)
- état : `boolean isAlive() == true` depuis `start()` jusqu'à fin d'exécution

Classe `Thread`

- attente de la fin du thread t : `t.join()`
- contrôle du thread actif courant (méth. statiques) :
  - mise en sommeil : `Thread.sleep(long millisec)`
  - rendre la main : `Thread.yield()`
- infos: `getName(),getPriority(),getThreadGroup()`
- priorité : `setPriority(int p)`, p entre `Thread.MIN_PRIORITY` et `Thread.MAX_PRIORITY`

Classe `Math`

- constantes `Math.PI`, `Math.E`
- fonctions mathématiques usuelles (toutes static, avec doubles en entrée et en sortie)

```
sin(), cos(),tan(), acos(),asin(), atan()
sqrt(), exp(), log(), pow( , )
ceil(), floor(), rint()
```

Classe `Math` (2)

- autres fonctions :
  - `int round(float), long round(double)`: arrondis à entier le plus proche
  - `abs(a), min(a,b), max(a,b)` pour a et b : `int`, `long`, `float` et `double`
  - `double random()` : nombre entre 0. inclus et 1. exclus, tiré d'une série pseudo-aléatoire nouvelle à chaque exécution du programme

ENTREES-SORTIES : paquetage `java.io`

- lecture/écriture séquentielle (flux)
- gestion fichiers

Les 4 catégories de flux

- flux d'octets : classes abstraites `InputStream` et `OutputStream`
- flux de caractères : classes abstraites `Reader` et `Writer`
- principales méthodes de lecture :
  - `int read()` : prochain élément du flux (ou -1 si "fin de flux")
  - `int read(byte[] buf)` pour `InputStream` et `int read(char[] buf)` pour `Reader`
  - `long skip(long nb)` : saute nb éléments
  - `void close()`

Les 4 catégories de flux (2)

- principales méthodes d'écriture :
  - `void write(int c)`
  - `void write (byte[] / char[] buf)`
  - pour `Writer : void write(String s)`
  - `void flush()`
  - `void close()`

Les flux "concrets"

- lecture/écriture (séquentielle) dans fichiers :

  `FileInputStream/FileOutputStream`,
  `FileReader/FileWriter`

  nom du fichier spécifié dans le constructeur :

  `new FileReader(filename)`

- lecture/écriture dans tableau en mémoire:

  `ByteArrayInput(/Output)Stream`
  `CharArrayReader/CharArrayWriter`

  tableau donné dans le constructeur :
  `new CharArrayReader(tab)`,

  récupéré par `toByteArray()` (resp. `toCharArray()`)

Les flux "concrets " (2)

- lecture/écriture dans une chaîne :

  `StringReader/StringWriter`

  chaîne donnée dans le constructeur (`new StringReader(ch)`), ou récupérée par `toString()`

- enchaînements de flux ("pipes") :

  `PipedInputStream/PipedOutputstream`,
  `PipedReader/PipedWriter`

Conversions pour flux binaire

- pour lire/écrire autre chose que octets ou caractères, on utilise des classes avec autres méthodes et qui font conversion avec flux "concrets" bas niveau :
- flux données binaires (types primitifs) :

`DataInputStream/DataOutputStream`

  – couche au-dessus d'un flux d'octets (`new DataOutputStream(outStream)`, ...)
  – méthodes lecture : `readFloat()`, `readInt()`,... et écriture : `writeFloat(x)`, `writeInt(i)`,...

Conversions pour flux binaire (2)

- flux données binaires (objets persistants) : `ObjectInputStream / ObjectOutputStream`
  – couche au-dessus d'un flux d'octets
  – mêmes méthodes que `DataXXXStream` + `Object readObject()` (resp. `writeObject(o)`)
  – objets doivent être de classe implémentant l'interface marqueur `Serializable`

Autres conversions de flux

- flux "avec tampon" (pour éviter accès disque ou réseau à chaque lecture/écriture) :
  `BufferedInputStream / BufferedOutputStream, BufferedReader / BufferredWriter`
- flux pour écriture type sortie standard : classe `PrintWriter`
  méthodes `print(), println(), ...` (comme sur `System.out`)
- flux compressés : paquetage `java.util.zip`
  (classes `ZipInputStream, ZipOutputStream`,...)
- conversion flux octets / flux caractères :
  `InputStreamReader/OutputStreamWriter`

Classes utilitaires pour entrées-sorties

- `StreamTokenizer` : pour découpage flux de caractères en "jetons" ("tokens") de type mot, nombre, blanc ou caractère
  - constructeur : `StreamTokenizer(Reader r)`
  - lecture jeton suivant : `int nextToken()`
    (renvoie, et met dans champ ttype le type du jeton parmi `TT_WORD, TT_NUMBER, TT_EOL, TT_EOF`, et met nombre dans champ `nval`, ou chaîne dans champ `sval` le cas échéant)

Classes utilitaires pour entrées-sorties

- `File` : pour manipuler fichiers/répertoires
  - constructeurs : `File(String name)`, `File(String path, String name)`, ...
  - méthodes :
    - `boolean exists()`, `long length()`, `File getParent()`
    - `boolean isDirectory()`, `String[] list()`
    - `void delete()`, `void mkdir()`,
    - `boolean renameTo(File f)`, ...
- `RandomAccessFile` : pour lecture/écriture sur fichier sans utiliser flux (permet accès direct non-séquentiel)

Paquetage `java.util` : COLLECTIONS, DATES, ...

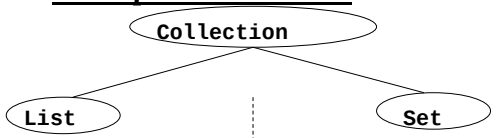
- Ensembles, Listes, Piles, Tables de hachage,...
- interface d'Itérateurs
- gestion des dates/heures
- internationalisation

Les collections

- une collection = un objet regroupant plusieurs éléments (ex. : tableau, liste, …)
- Java 1.2 propose, en plus des tableaux, deux grands types de collections :
  - ensemble : aucun élément dupliqué, pas d'ordre intrinsèque
  - liste : collection ordonnée avec répétition possible du même élément

Les collections Java

Principales interfaces :



Classes implémentant ces interfaces :

`Vector`
`ArrayList`
`LinkedList`
`TreeSet`
`HashSet`

L'interface `Collection`

- l'interface `Collection` permet de manipuler toutes les collection de façon unifiée, et regroupe les méthodes communes à toutes les collections :
  - taille : `int size()`
  - test si vide : `boolean isEmpty()`
  - ajout : `boolean add(Object element)`
  - suppression : `boolean remove(Object elt)`
  - recherche : `boolean contains(Object elmt)`
  - parcours : `Iterator iterator()`
  - …

L'interface `Iterator`

- permet de parcourir n'importe quelle collection
- contient les méthodes :
  - `boolean hasNext()`
  - `Object next()`
  - `void remove()`

Les listes (interface `List`)

- identique à `Collection`, avec en plus des méthodes pour l'accès par position, et un itérateur spécifique :
  - `Object get(int index)`
  - `Object set(int index, Object element)`
  - `void add(int index, Object element)`
  - `Object remove(int index)`
  - `int indexOf(Object elt)`
  - `ListIterator listIterator()`
  - …

Les listes (2)

- `ListIterator` : comme `Iterator`, mais permet de parcourir dans les 2 sens (ajout des méthodes `Object previous()`, `boolean hasPrevious()`, …)
- trois implémentations de l'interface `List` :
  - classe `Vector`
  - classe `ArrayList`
  - classe `LinkedList`

Les ensembles (interface `Set`)

- exactement les même méthodes que `Collection`, mais impossible de mettre 2 fois le même élément
- deux implémentations :
  - classe `TreeSet`
  - classe `HashSet`

Classe `Vector`

- Vecteur (tableau à taille modifiable) : classe `Vector`
  - ajout d'élément : void `addElement(Object elt)`, `void insertElementAt(Object o, int index)`
    [en Java1.2, similaires : `boolean add(Object)` et void `add(int index, Object o)`]
  - modification d'élément :
    `void setElementAt(Object o, int index)`
    [Java1.2 : `Object set(Object o, int index)`]
  - taille : `int size()`, `boolean isEmpty()`

Classe `Vector` (2)

- accès aux éléments :
  `Object elementAt(int index)`,
  [Java1.2 : `Object get(int index)`]
  `boolean contains(Object o)`,
  `int indexOf(Object o)`
- suppression d'élément : `void removeElementAt(int index)`, `boolean removeElement(Object o)`
  [Java1.2 : `Object remove(int)` et boolean `remove(Object)`]

Classe `Stack`

- Pile : classe `Stack` dérivée de `Vector`, avec des méthodes supplémentaires
  - empiler : `Object push(Object)`
  - sommet : `Object peek()`
  - dépiler : `Object pop()`

Enumeration

- interface d'itérateur : `Enumeration`
  - élément suivant : `Object nextElement()`
  - test de fin : `boolean hasMoreElements()`
  - utilisée par diverses classes avant Java1.2 (exemple : `v.elements()` sur un `Vector`)
  - a priori remplacée maintenant par `Iterator`

Hashtable

- table de hachage (ensemble d'éléments rangés en fonction de leur "clef") : classe `Hashtable`, qui définit les méthodes :
  - ajout dans table : `Object put(Object clef, Object o)`
  - recherche dans table : `Object get(Object clef)`
  - suppression : `Object remove(Object clef)`
  - taille : `int size(), boolean isEmpty()`
  - liste des éléments : `Enumeration elements()`
  - liste des clef : `Enumeration keys()`

BitSet

- "champ de bits" : classe `BitSet`
  - constructeur : `BitSet(int taille)`
  - mise à vrai : `void set(int position)`
  - mise à faux : `void clear(int position)`
  - opérations : `void and(BitSet autre), void or(BitSet autre), void xor(BitSet autre)`
  - taille courante : `int size()`

Divers : `Calendar, Locale`

- date et heure : classe `Calendar`
  - maintenant : `Calendar now=Calendar.getInstance();`
  - extraire année, mois, ... : méthode `int get(int champ)`, avec champ valant `Calendar.YEAR, Calendar.MONTH`,...
  - modifier : `void set(int quoi, int valeur)`, ou `void set(annee, mois, jour)`, ...
  - comparer : `boolean after(Calendar c2)`, ...
  - chaîne affichable : `Date getTime()`
- internationalisation : classe `Locale`

INTERFACES HOMME-MACHINE (IHM) GRAPHIQUES

IHM graphiques

- Java inclut diverses classes facilitant la création d'Interfaces Homme-Machine graphiques PORTABLES
- cette IHM peut être :
- soit une applet intégrée dans une page Web,
- soit une application indépendante
- paquetages :
  - `java.awt` : bases communes
  - `java.applet`
  - `javax.swing` : composants basés sur `awt`, de plus haut niveau, et plus efficaces

Applet v.s. application

- pour les applets téléchargées, il y a de nombreuses restrictions pour la sécurité :
  - accès aux fichiers locaux restreints
  - pas de connexion réseau avec des sites autres que le site d'origine
  - pas le droit de lancer d'autres programmes
  - interdiction de contenir des méthodes "natives", ou de charger des librairies
- ces restrictions ne s'appliquent pas :
  - aux applets locales exécutées avec appletviewer
  - aux applets téléchargées mais définies comme "de confiance" par l'utilisateur, et authentifiées par une signature cryptée

tag HTML <APPLET>

- pour insérer une applet dans une page Web, celle-ci doit contenir :

```
<APPLET code="NomApplet.class"
   width=w height=h>
...
</APPLET>
```

où `w` et `h` sont les dimensions réservées à l'applet (en pixels)

- on peut insérer (entre le tag ouvrant et le tag fermant) des tags de la forme :

```
<PARAM name=Nom  value=Val>
```

pour passer des paramètres récupérables dans l'applet

Ecrire une Applet

- pas de `main()`, mais une sous-classe de `Applet` ou `JApplet`, redéfinissant tout ou partie des méthodes suivantes (appelées par le browser, ou par appletviewer) :
  - `void init()` : exécutée au chargement de l'applet
  - `void start()` : exécutée au démarrage de l'applet (doit rendre la main => lancer des threads si besoin de boucle continue)
  - `void paint(Graphics)` : dessine l'applet (est normalement appelée automatiquement par `update()` quand nécessaire)
  - `void update(Graphics)` : appelée automatiquement quand il faut redessiner l'applet
    (par défaut : efface, puis appelle `void paint()`)

Ecrire une Applet (2)

  - `void print(Graphics)` : pour imprimer, si différent de affichage
  - `void stop()` : exécutée quand l'applet devient temporairement invisible (=> suspendre les threads, ...)
  - `void destroy()` : exécutée au déchargement de l'applet (=> libérer les ressources)

- méthodes essentielles souvent utilisées :
  - `void repaint()` : demande de ré-affichage (exécutée par le système dès que possible)
  - `Image getImage(URL base, String name)` : chargement en mémoire d'un fichier image

Ecrire une Applet (3)

- `AudioClip getAudioClip(Url base, String name)` : chargement d'un fichier son
- `URL getDocumentBase()` : récupération de l'URL du document web incluant l'applet
- `URL getCodeBase()` : récupération de l'URL du répertoire de l'applet
- `String getParameter(String)` : pour récupérer la valeur d'un des <PARAM>

Ecrire une Applet (4)

- affichage d'une image : `g.drawImage(img, x,y,this);` (dans la méthode `void paint(Graphics g)`)
- utilisation d'un son `Audioclip ac` : `ac.play(), ac.loop(), ac.stop()`

Une applet élémentaire

- Applet affichant "Bonjour !"

```
import java.applet.*; // Pour classe Applet
import java.awt.*;    // Pour classe Graphics

public class BonjourApplet
                    extends Applet {
  /** Redéfinition de paint(), appelée
        quand il faut par le browser
  */
  public void paint(Graphics g) {
    g.drawString("Bonjour !", 25, 50);
  }
}
```


Une applet avec image et son

- Affichage d'une image à une position modifiée à chaque redémarrage, et son joué durant l'affichage

```
import java.applet.*;
import java.awt.*;
import java.net.*; // pour la classe URL
public class ImageSonApplet extends Applet{
  private Image img;
  private int xIm, yIm;
  private AudioClip son;
  /** Chargement image et son */
  public void init() {
    URL base = getDocumentBase();
    img = getImage(base, "im.gif");
```

Une applet avec image et son (2)

```
 son = getAudioClip(base, "s.au");
 xIm=10;
 yIm=20; }
/** Jouer le son */
public void start() { son.loop(); }
/** Afficher image */
public void paint(Graphics g) {
  g.drawImage(img,xIm,yIm,this); }
/** Arrêter son, modifier position */
public void stop() {
  son.stop();
  xIm+=20; yIm+=20; }
}
```


Une applet animée

- affiche un disque de diamètre variable

```
import java.applet.*;
import java.awt.*;
public class AnimApplet extends Applet
                implements Runnable {
 private int x=50,y=50,dt=100,r=5,dr=1;
 private final int MAX_DIM=40;
 private Thread anim;
 private boolean suspendAnim=false;
 public void init(){anim=new Thread(this);}
 public void start() {
   if (!anim.isAlive()) anim.start();
   else suspendAnim=false; }
```

Une applet animée (2)

```
 public void paint(Graphics g) {
    g.fillOval(x,y,2*r,2*r); }
 /** Corps du thread d'animation */
 public void run() {
    while (true) {
      if (!suspendAnim) {
         if (r>MAX_DIM||r<1) dr*=-1;
         r += dr; }
      try{Thread.sleep(dt);}
        catch(InterruptedException e){}
      repaint();}

  }
  public void stop() {suspendAnim=true;}

}
```


Paquetages `java.awt` et `javax.swing`

- composants graphiques élémentaires (boutons, menus, ...)
- conteneurs (fenêtres, ...)
- primitives graphiques pour dessiner
- gestion des images et des sons
- modèle d'événements

De `java.awt` à `javax.swing`

- changement du modèle de gestion des événements entre le `jdk 1.0` et le `jdk 1.1`
- utilisation des composants graphiques natifs des différents systèmes de multifenêtrage des différents systèmes d'exploitation dans `java.awt`
  - manque d'efficacité, grosse consommation mémoire
  - problèmes de portabilité
- utilisation généralisée de composants légers indépendants du système de multifenêtrage natif dans `javax.swing`
  - permet de choisir le « look and feel » indépendamment de l'environnement graphique à l'exécution

Composants graphiques élémentaires

- Briques de base à assembler pour faire l'IHM ; classes abstraites `Component` de `awt`, `JComponent` de `swing` et leurs sous-classes :
  - `Button` et `JButton` : bouton
  - `Label` et `JLabel` : texte affiché par le programme
  - `TextField`, `TextArea`, `JTextField` et `JTextArea` : pour entrer du texte
  - `Checkbox`, `CheckboxGroup` et `JCheckBox` : case à cocher/décocher (éventuellement choix parmi un ensemble mutuellement exclusif)
  - `Choice` : menu déroulant permettant un choix exclusif entre les valeurs proposées

Composants graphiques élémentaires (2)

- `List` et `JList` : ensemble de choix cumulables, présentés dans une fenêtre avec ascenseur (scrollable)
- `MenuBar, Menu, MenuItem, JMenuBar, Jmenu` et `JMenuItem` : pour faire menus déroulants usuels
- `PopupMenu` et `JPopupMenu` : menu "contextuel" lié à un autre composant au-dessus duquel il peut apparaître
- `ScrollBar` et `JScrollBar` : barre de défilement
- `Canvas` : zone pour dessiner, ou base pour créer de nouveaux composants par dérivation
- `Container` (classe abstraite) : pour grouper des composants selon un certain positionnement

Types de Conteneurs `awt`

- classe abstraite `Container` (sous-classe de `Component`), avec sous-classes :
  - `Panel` : conteneur de base, pour grouper des composants
  - `ScrollPane` : conteneur avec barres de défilement, mais pour un seul composant
  - `Window` : fenêtre (sans bordure, ni titre), pour création fenêtres personalisées par dérivation
  - `Frame` : Fenêtre "usuelle" avec bandeau en haut
  - `Dialog` : fenêtre secondaire, associée à une fenêtre maîtresse (notamment pour pop-up de dialogue)
  - `FileDialog` : fenêtre de sélection de fichier

NOTE : la classe `Applet` dérive de `Panel`

Types de Conteneurs `swing`

- Classes dérivées de `java.awt.Container` :
  - `JComponent` et toutes ses classes dérivées, `JPanel`, `JScrollPane`, `JFrame`, `JDialog`
  - `Box` : conteneur léger qui permet d'ajouter facilement des composants de droite à gauche ou de haut en bas

Placement dans les conteneurs

- à chaque conteneur est associé un `LayoutManager` qui définit la façon de positionner ses composants,
- l'interface `LayoutManager` et l'interface dérivée `LayoutManager2` sont implémentées par diverses classes :
  - `BoxLayout` : en ligne ou en colonne, utilisé par la classe `Box`
  - `FlowLayout` : de gauche à droite et de haut en bas, dans ordre d'ajout (ou en fonction de position demandée)
  - `GridLayout` : nombre de lignes et colonnes définis à création, puis idem `FlowLayout`, sauf : nombre de colonnes reste fixe lors d 'un redimensionnement.(=> composants retaillés en conséquence)

Placement dans les conteneurs (2)

- `BorderLayout` : 5 composants seulement, placés en haut, bas, droite, gauche ou centre (cf. constantes de classe `NORTH, SOUTH, EAST, WEST, CENTER`)
- `CardLayout` : superposé
- `GridBagLayout` : basé grille, mais très paramétrable via `GridBagConstraints`

Fonctionnement des conteneurs

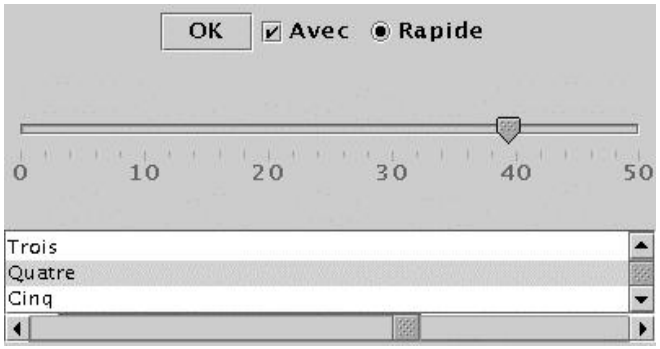
- gestion des composants :
  - `Component add(Component c)` : ajout en première place vide (au centre pour `BorderLayout`)
  - `void add(Component c, Object contrainte)` : ajout en précisant une contrainte (`BorderLayout.NORTH` pour container en mode `BorderLayout`, ...)
  - `Component add(Component c, int pos)` : ajout à la position pos entre 0 et la place du dernier composant +1 (ne marche pas pour BorderLayout)

Fonctionnement des conteneurs (2)

- `int getComponentCount()` : nombre de composants
- `Component [] getComponents()` : tableau des composants
- `Component getComponent(int pos)` : composant situé à la position `pos`
- `void remove(int pos)` : retrait du composant situé à position pos
- `void removeAll()`

- gestion du type de placement :
  - `void setLayout(LayoutManager)`
  - `LayoutManager getLayout()`

Composants élémentaires : boutons, … (1)



Composants élémentaires : boutons, … (2)

```
import java.awt.*;
import javax.swing.*;
public class ExCom  {
    public static void main(String s[]) {
        JFrame frame = new JFrame();
        frame.getContentPane().setLayout(new
  GridLayout(3,1));
        JPanel bouPan=new JPanel();
        JButton bou=new JButton("OK");
        bouPan.add(bou);
        JCheckBox co=new JCheckBox("Avec");
        bouPan.add(co);
        JRadioButton ra=new JRadioButton("Rapide");
        bouPan.add(ra);
```


Composants élémentaires : boutons, … (3)

```
        frame.getContentPane().add(bouPan,0);
        JSlider sli= new
   JSlider(JSlider.HORIZONTAL,0,50,25);
        sli.setMinorTickSpacing(2);
        sli.setMajorTickSpacing(10);
        sli.setPaintTicks(true);
        sli.setPaintLabels(true);
        frame.getContentPane().add(sli,1);
        JPanel pan=new JPanel();
        pan.setLayout(new BorderLayout(5,5));
        String
   label[]={"Un","Deux","Trois","Quatre","Cinq"};
        JList li=new JList(label);
        JScrollPane liAsc=new JScrollPane(li);
        liAsc.setPreferredSize(new Dimension(400,50));
```

Composants élémentaires : boutons, … (4)

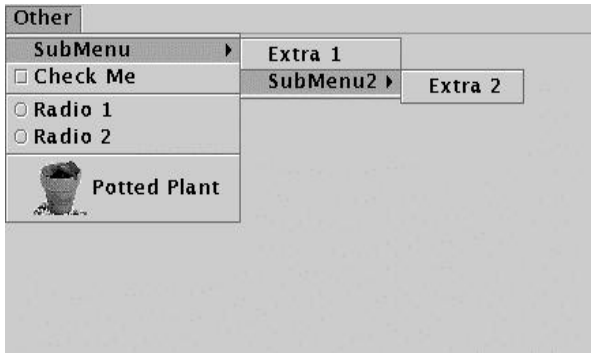
```
        pan.add(liAsc,BorderLayout.NORTH);
        JScrollBar bar=new
  JScrollBar(JScrollBar.HORIZONTAL,30,20,0,600);
        bar.setUnitIncrement(2);
        bar.setSize(400,20);
        pan.add(bar,BorderLayout.SOUTH);
        frame.getContentPane().add(pan,2);
        frame.setSize(400,200);
        frame.setVisible(true);
    }
}
```


Les Menus

- créer une `JMenuBar mb`
- l'insérer dans un `JFrame f` avec `f.setJMenuBar(mb);`
- créer les menus `JMenu m` et les ajouter à la `JMenuBar` avec `mb.add(m);`
- créer les `JCheckboxMenuItem, JCheckBoxMenuItem` et/ou `JMenuItem` et les ajouter aux menus avec `JMenuItem i; m.add(i);`

  NOTE : `JMenu` dérive de `JMenuItem`
  => on fait des sous-menus en ajoutant un menu comme item dans un autre

Exemple de menu



Exemple de menu (2)

```
import java.awt.*;
import javax.swing.*;
public class ExMenu extends JMenuBar  {
    public ExMenu() {
        JMenu otherMenu = new JMenu("Other");
        JMenu subMenu = new JMenu("SubMenu");
        JMenu subMenu2 = new JMenu("SubMenu2");
        //  Assemble the submenus of the Other Menu
        JMenuItem item;
        subMenu2.add(item = new JMenuItem("Extra 2"));
        subMenu.add(item = new JMenuItem("Extra 1"));
        subMenu.add(subMenu2);
```

Exemple de menu (3)

```
    //  Assemble the Other Menu itself
        otherMenu.add(subMenu);
        otherMenu.add(item = new JCheckBoxMenuItem("Check
Me"));
        otherMenu.addSeparator();
        ButtonGroup buttonGroup = new ButtonGroup();
        otherMenu.add(item = new
JRadioButtonMenuItem("Radio 1"));
        buttonGroup.add(item);
        otherMenu.add(item = new
JRadioButtonMenuItem("Radio 2"));
        buttonGroup.add(item);
        otherMenu.addSeparator();
        otherMenu.add(item = new JMenuItem("Potted Plant",
new ImageIcon("image.gif")));
```


Exemple de menu (4)

```
        //  Finally, add all the menus to the menubar
            add(otherMenu);
        }
        public static void main(String s[]) {
            JFrame frame = new JFrame("Simple Menu Example");
            frame.setJMenuBar(new ExMenu());
            frame.setSize(400,200);
            frame.setVisible(true);
        }
    }
```

Dessiner

- pour dessiner sur un composant, il faut un "contexte graphique" (objet `Graphics`) associé
- on obtient une copie du contexte graphique courant d'un composant par un appel à sa méthode `Graphics getGraphics()`

[possible a priori que pour les conteneurs et les composants de type `Canvas`]

Méthodes pour dessiner

- méthodes de `Graphics` pour dessiner :
  - `void drawString(String texte, int x, int y)`
  - `void drawLine(int x1, int y1, int x2, int y2)`
  - `void drawRect(int x, int y, int l, int h)`
  - `void drawOval(int x, int y, int l, int h)`
  - `void drawArc(int x, int y, int l, int h, int angleDeb, int angleArc)`
  - `void drawPolygone(int[] tabX, int[] tabY, int nbPoints)` et `void drawPolygon(Polygone)`
  - `void drawPolyline(int[] tabX, int[] tabY, int nbPoints)`
  - + méthodes de "remplissage" associées : `fillRect()`, `fillArc()`, ...

Couleurs

- les dessins sur un `Graphics` se font avec la "couleur courante", consultable et modifiable par ses méthodes respectives :
  - `Color getColor()`
  - `void setColor(Color)`
- les couleurs d'avant-plan et d'arrière-plan d'un composant sont aussi modifiables via les méthodes de `Component` :
  - `Color getForeground()`
    `void setForeground(Color)`
  - `Color getBackground()`
    `void setBackground(Color)`

Couleurs (2)

- classe `Color` :
  - constructeur
    `Color(int rouge, int vert, int bleu)`
  - couleurs usuelles : constantes de classe `Color.black`, `Color.red`, `Color.blue`, `...`

Polices de caractères

- chaque `Graphics` et `Component` a une " police courante" consultable/modifiable par ses méthodes :
  - `Font getFont()`
  - `setFont(Font)`
- polices existant sur toutes plates-formes :
  Serif, SansSerif, Monospaced, Dialog et DialogInput
- liste complète locale par appel à :
  `String[] Toolkit.getDefaultToolkit().getFontList()`

Polices de caractères (2)

- classe `Font` :
  - constructeur `Font(String nom, int style, int taille)`
    - `nom` est par exemple "Helvetica"
    - `style` est une des constantes `Font.BOLD`, `Font.ITALIC`, `Font.PLAIN` (ou une combinaison par |)
    - `taille` est la dimension en "points "

La saisie de texte

- sous paquetage de swing particulier : `javax.swing.text`


```
import java.awt.*;
import javax.swing.*;
public class ExTex  {
  public static void main(String s[]) {
```


La saisie de texte (2)

```
  JFrame frame = new JFrame();
       frame.getContentPane().setLayout(new
GridLayout(2,1));
       JPanel pan1= new JPanel();
       frame.getContentPane().add(pan1,0);
       pan1.setLayout(new GridLayout(2,2,2,2));
       JLabel la1=new JLabel("Nom");
       JTextField t1= new JTextField("Invité");
       JLabel la2=new JLabel("Mot de passe");
       JPasswordField t2=new JPasswordField();
       pan1.add(la1);
       pan1.add(t1);
```

La saisie de texte (3)

```
        pan1.add(la2);
        pan1.add(t2);
        JPanel pan2= new JPanel();
        JTextArea ta=new JTextArea();
        ta.setFont(new
Font("Helvetica",Font.BOLD,12));
        ta.append("Bienvenue.\n");
        ta.append("Ceci est un exemple de\n");
        ta.append("JTextArea.\n");
        pan2.add(ta);
        frame.getContentPane().add(pan2,1);
        frame.setSize(400,100);
        frame.setVisible(true);} }
```


Images

- chargement par les méthodes de `JApplet`
  - `Image getImage(URL url_base,String name)`
  - `Image getImage(URL url)`
- ou bien, hors des applets, par les méthodes de la classe `Toolkit`
  - `Image getImage(String filename)`
  - `Image getImage(URL url)`
- la classe `Toolkit` est la superclasse commune à toutes les implémentations de systèmes de fenêtres
- on obtient le `Toolkit` associé à un composant par `Toolkit getToolkit()`, ou bien par `Toolkit.getDefaultToolkit()`

Images (2)

- affichage dans un `Graphics` :
  ```
  boolean drawImage(Image i,int x,int y,
                    ImageObserver ob)
  ```
  - `ImageObserver` est une interface implémentée par les composants, `ob` désigne le composant sur lequel on veut afficher l 'image
- méthodes de la classe `Image` :
  - `int getHeight(ImageObserver)`
  - `int getWidth(ImageObserver)`
  - `Image getScaledInstance(int l, int h, int method)`
  - `Graphics getGraphics()` pour pouvoir dessiner dessus
  - ...

Création d'images

- dans certains cas (double buffering pour animation, ...) on veut dessiner dans un objet `Image` non visible avant de l'afficher. Pour faire cela :
  - créer une image par
    ```
    Image img=createImage(largeur,hauteur);
    // méthode de Component
    ```
  - récupérer un `Graphics` associé :
    ```
    Graphics g = img.getGraphics();
    ```
  - dessiner sur `g`
  - quand on le souhaite, afficher l'image dans le composant voulu par appel de `drawImage()` sur un `Graphics` associé au composant (et non à l'image)

Autres méthodes de `Graphics`

- `void clearRect(x,y,l,h)` : remet à la couleur de fond la zone rectangulaire spécifiée
- `void copyArea(x,y,l,h,deltaX, deltaY)` : recopie d'une zone rectangulaire avec translation de (`deltaX`,`deltaY`)
- `void setClip(x,y,l,h)` : modifie la zone où les choses sont effectivement dessinées

Position et dimension des composants

- position (coin en haut à gauche du rectangle englobant le composant) :
  - `Point getLocation()`, où `Point` regroupe dans attributs publics `x` et `y` la position relative au composant "père"
  - `void setLocation(x,y), void setLocation(Point p)`
- dimension (du rectangle englobant) :
  - `Dimension getSize()`, où `Dimension` regroupe dans attrib. publics `width` et `height`
  - `void setSize(w,h), void setSize(Dimension d)`

Position et dimension des composants (2)

- bornes (position + dimension) :
  - `Rectangle getBounds()`, où `Rectangle` regroupe `x, y, width, height` et diverses méthodes
- test d'inclusion d'un point dans bornes :
  - `boolean contains(x,y), boolean contains(Point p)`

Curseur

- classe `Cursor` :
  - constructeur `Cursor(int type)` où `type` parmi :
    - `Cursor.DEFAULT_CURSOR`
    - `Cursor.CROSSHAIR_CURSOR`
    - `Cursor.HAND_CURSOR`
    - `Cursor.TEXT_CURSOR`
    - `Cursor.WAIT_CURSOR`
    - `Cursor.MOVE_CURSOR`
    - `Cursor.`*`XX`*`_RESIZE_CURSOR` où XX est N, S, E, W, NE, NW, SE, ou SW

Curseur (2)

- méthodes statiques :
  ```
  static Cursor Cursor.getDefaultCursor()
  static Cursor Cursor.getPredefinedCursor(int)
  static Cursor
  Cursor.getSystemCustomCursor(String name)
  ```
- méthodes :
  ```
  int getType()
  String getName()
  String toString()
  ```

- chaque composant a un curseur associé :
  - `Cursor getCursor()`
  - `void setCursor(Cursor cur)`

Ecrire une IHM hors applet

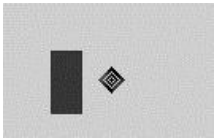
- écrire une sous-classe de `JFrame` adaptée à l'application
- dans son constructeur, créer les sous-conteneurs et composants élémentaires, et installer chacun à sa place dans son conteneur (méthode `add()`)
- redéfinir éventuellement la méthode `void paint(Graphics)` pour y faire tout ce qui est dessin
- dans le `main` :
  - créer une instance `f` de la sous-classe de `JFrame` en question
  - afficher la fenêtre : `f.setVisible(true)`

Exemple d'application graphique

```
import java.awt.*;
import javax.swing.*;
class Gra extends JFrame {
 private Image img;
 private JPanel zoneImg;
 public Gra() {
   img=getToolkit().getImage("3DDiamond.gif");
   zoneImg = new JPanel();
   getContentPane().add(zoneImg,
  BorderLayout.CENTER);
 }
 public void paint(Graphics g) {
   super.paint(g);
```


Exemple d'application graphique (2)

```
    Graphics ig=zoneImg.getGraphics();
    ig.drawImage(img,60,40,this);
    ig.setColor(Color.red);
    ig.fillRect(30,30,20,40);
  }
  public static void main(String[] args){
    Gra a = new Gra();
    a.setSize(100,100);
    a.setVisible(true);
  }
}
```



Evénements

- c'est le mécanisme permettant l'interaction avec l'utilisateur via l'interface graphique
- fonctionnement totalement changé entre Java 1.0 et 1.1
- en Java 1.1 (et ultérieurs), modèle émetteur/récepteur, avec séparation claire entre les éléments d'interface qui émettent les événements, et des objets récepteurs qui "écoutent" les événements et agissent en conséquence
- classes dans le paquetage `java.awt.event` et `javax.swing`

Modèle d'événement depuis Java 1.1

- chaque composant peut générer des événements (classe abstraite `AWTEvent` et ses sous-classes `MouseEvent, ActionEvent`, ...)
- tout objet `o` qui doit réagir quand un type d'événement se produit dans un certain composant `c` doit :
  - implémenter l'interface adéquate (`MouseListener, ActionListener`, ...)
  - être enregistré dans la liste des objets intéressés par ce type d'événements issus de ce composant (`c.addMouseListener(o)`, `c.addActionListener(o)`, ...)

Modèle d'événement (2)

- quand un événement se produit sur le composant `c`, il est transmis à tous les récepteurs enregistrés chez lui pour ce type d'événement, ceci par appel de sa méthode correspondante
  - pour appui sur bouton souris, `o.mousePressed(evt)` pour tous les `o` concernés, et où `evt` est l'événement

Principaux types d'événements

- `WindowEvent` : apparition, iconification, (dé-)masquage, fermeture, ...
- `ComponentEvent` : redimensionnement, déplacement, ...
- `FocusEvent` : début/fin de sélection comme destinataire des inputs (clavier et souris)
- `KeyEvent` : clavier (touche appuyée, ...)
- `MouseEvent` : souris (boutons, déplacement, ...)

Principaux types d'événements (2)

- `ActionEvent` : appui sur `Button`, `JButton`, double-clic sur item de `List`, `JList`, return dans un `TextField`, `JTextField`, choix d'un `MenuItem`, `JMenuItem`, ...
- `ItemEvent` : (dé-)sélection d'une `Checkbox`, `JCheckBox`, d'un item de `List`, `JList` ou de `Choice`, passage sur un `MenuItem`, `JMenuItem`, ...
- `TextEvent` : modification de texte
- `ContainerEvent` : ajout/suppression de composant
- `AdjustementEvent` : défilement de `Scrollbar`, `JScrollBar`

Sources des événements

tous les `Component`	`ComponentEvent, FocusEvent, KeyEvent, MouseEvent`
toutes les `Window`	`WindowEvent`
`Button, JButton, MenuItem, JMenuItem,`	`ActionEvent`
`List, Jlist`	`ActionEvent, ItemEvent`
`Checkbox, JCheckBox, CheckboxMenuItem, JCheckBoxMenuItem, Choice`	`ItemEvent`
`TextField JtextField`	`ActionEvent, TextEvent`
`TextArea JTextArea` et autres `(J)TextComponent`	`TextEvent`
tous les `Container`	`ContainerEvent`
`Scrollbar, JScrollBar`	`AdjustmentEvent`

Interfaces récepteurs

- à chaque classe *Xxx*`Event` est associée une interface *Xxx*`Listener`, qui regroupe une ou plusieurs méthodes (lesquelles passent toutes l'événement en paramètre) :

`KeyEvent`	`KeyListener`	`keyPressed()`
		`keyReleased()`
		`keyTyped()`

`MouseEvent`	`MouseListener`	`mouseClicked()`
		`mouseEntered()`
		`mouseExited()`
		`mousePressed()`
		`mouseReleased()`

	`MouseMotionListener`	`mouseDragged()` `mouseMoved()`

`ComponentEvent`	`ComponentListener`	`componentHidden()`
		`componentMoved()`
		`ComponentResized()`
		`componentShown()`

`FocusEvent`	`FocusListener`	`focusGained()` `focusLost()`

Interfaces récepteurs (2)

`WindowEvent`	`WindowListener`	`windowActivated()`
		`windowClosed()`
		`windowClosing()`
		`windowDeactivated()`
		`windowDeiconified()`
		`windowIconified()`
		`windowOpened()`

`ActionEvent`	`ActionListener`	`actionPerformed()`
`ItemEvent`	`ItemListener`	`itemStateChanged()`
`TextEvent`	`TextListener`	`textValueChanged()`
`AdjustmentEvent`	`AdjustmentListener`	`adjustmentValueChanged()`
`ContainerEvent`	`ContainerListener`	`componentAdded()` `componentRemoved()`

Classes Adaptateurs

- pour chaque interface *Xxx*`Listener` est prédéfinie aussi une classe *Xxx*`Adapter` qui implémente l'interface avec des méthodes toutes vides
- ainsi, un objet intéressé par un seul sous-type d'événement (e.g. clic souris), peut être défini comme héritant de la classe adaptateur, et du coup n'avoir à redéfinir que la méthode souhaitée :
  - hériter de la classe `MouseAdapter` en redéfinissant uniquement `mouseClicked()`

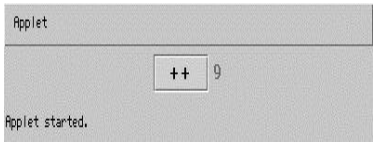
Ecrire une méthode de gestion d'événement

- se servir des méthodes des événements pour avoir détails sur ce qui s'est passé :
  - méthode commune : `int getID()` donne la nature précise de l'événement
  - `ComponentEvent` : `getComponent()`
  - `InputEvent` :
    `long getWhen(), int getModifiers(),`
    `void consume(), boolean isConsumed()`
  - `MouseEvent` coordonnées (relatives à source) :
    `int getX(), int getY()`
  - `KeyEvent` :
    `char getKeyChar()` ou `int getKeyCode()`

Ecrire une méthode de gestion d'événement (2)

- `WindowEvent : getWindow()`
- `ActionEvent : String getActionCommand()`
- `ItemEvent :`
  `Object getItem()` et `int getStateChange()`
- `ContainerEvent : Component getChild()`

Exemple de gestion d'événements




```
import java.applet.*;
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
public class EventApplet extends Applet
implements ActionListener {
  private JLabel lab;
  private int k=0;
  public void init() {
    JButton b = new JButton("++");
// enregistrement de l'applet comme
// récepteur du bouton
    b.addActionListener(this);
    add(b);
```

Exemple de gestion d'événements (2)

```
     lab=new JLabel(String.valueOf(k));
    add(lab);
// création gestionnaire des clics souris
    MouseListener ml = new MouseAdapter() {
        public void mouseClicked(MouseEvent e){
          k = e.getX();
          lab.setText(String.valueOf(k));
        } }    ;
      addMouseListener(ml);}
/** Incrémente la label quand on a cliqué sur le
  bouton */
    public void actionPerformed(ActionEvent e) {
      lab.setText(String.valueOf(++k));
      lab.setSize(lab.getMinimumSize());} }
```


Changement de « look and feel »

```
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
public class LnFListener implements ActionListener {
 Frame frame;
 public LnFListener(Frame f) {
  frame = f;
 }
 public void actionPerformed(ActionEvent e) {
  String lnfName = null;
  if (e.getActionCommand().equals("Metal")) {
   lnfName = "javax.swing.plaf.metal.MetalLookAndFeel";
  } else if (e.getActionCommand().equals("Motif")) {
   lnfName = "com.sun.java.swing.plaf.motif.MotifLookAndFeel";
  } else {
   lnfName = "com.sun.java.swing.plaf.windows.WindowsLookAndFeel";
  }
```

Changement de « look and feel » (2)

```
try {
        UIManager.setLookAndFeel(lnfName);
        SwingUtilities.updateComponentTreeUI(frame);
      }
      catch (UnsupportedLookAndFeelException ex1) {
        System.err.println("Unsupported LookAndFeel: " + lnfName);
      }
      catch (ClassNotFoundException ex2) {
        System.err.println("LookAndFeel class not found: " + lnfName);
      }
      catch (InstantiationException ex3) {
        System.err.println("Could not load LookAndFeel: " + lnfName);
      }
      catch (IllegalAccessException ex4) {
        System.err.println("Cannot use LookAndFeel: " + lnfName);
      }
    }
  }
```


Changement de « look and feel » (3)

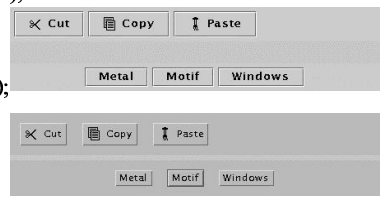
```
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
public class ToolbarFrame3 extends Frame implements ActionListener {
  // This time, let's use JButtons!
  JButton cutButton, copyButton, pasteButton;
  JButton winButton, javaButton, motifButton;
  public ToolbarFrame3() {
    super("Toolbar Example (Swing)");
    setSize(450, 100);
    addWindowListener(new BasicWindowMonitor());
    JPanel toolbar = new JPanel();
    toolbar.setLayout(new FlowLayout(FlowLayout.LEFT));
    cutButton = new JButton("Cut", new ImageIcon("cut.gif"));
    cutButton.addActionListener(this);
```

Changement de « look and feel » (4)

```
toolbar.add(cutButton);
 copyButton = new JButton("Copy", new ImageIcon("copy.gif"));
 copyButton.addActionListener(this);
 toolbar.add(copyButton);
 pasteButton = new JButton("Paste", new ImageIcon("paste.gif"));
 pasteButton.addActionListener(this);
 toolbar.add(pasteButton);
 add(toolbar, BorderLayout.NORTH);
 // Add the look and feel controls using regular AWT buttons
 JPanel lnfPanel = new JPanel();
 LnFListener lnfListener = new LnFListener(this);
 javaButton = new JButton("Metal");
 javaButton.addActionListener(lnfListener);
 lnfPanel.add(javaButton);
```


Changement de « look and feel » (5)

```
  motifButton = new JButton("Motif");
  motifButton.addActionListener(lnfListener);
  lnfPanel.add(motifButton);
  winButton = new JButton("Windows");
  winButton.addActionListener(lnfListener);
  lnfPanel.add(winButton);
  add(lnfPanel, BorderLayout.SOUTH);
 }
public void actionPerformed(ActionEvent ae) {
  System.out.println(ae.getActionCommand());
 }
 public static void main(String args[]) {
  ToolbarFrame3 tf3 = new ToolbarFrame3();
  tf3.setVisible(true);
 } }
```



PROGRAMMATION RESEAU : paquetage `java.net`

- manipulation des URL
- accès aux protocoles standards de l'Internet (http, ftp, mail, ...)
- communication inter-process

Accès aux protocoles Internet

- classe `URL` (protocoles http, ftp, ...)
  - constructeurs : `URL(String nom)`, `URL(URL base, String nom)`, ou `URL(protocole,host,port,file)`
  - ouverture de flux en lecture : `InputStream openStream()`
  - manipulation plus fine, infos sur le contenu : `URLConnection openConnection()`, puis méthodes de `URLConnection` :
    - `connect()`,
    - `getContentType()`, `getContentLength()`, ...
    - `getInputStream()`, `getOutputStream()`

Accès aux protocoles Internet (2)

- conversion chaîne au format url (pour envoi de donnée à programme CGI) :
  `String URLEncoder.encode(String)`
- manipulation adresse internet : classe `InetAddress` (instances obtenues par `static InetAddress InetAddress.getLocalHost()` ou `static InetAddress InetAddress.getByName(String host)`)

Exemple d'utilisation de la classe URL

```
import java.net.*;
import java.io.*;
class VisuHTML {
 public static void main(String[] arg) {
  try{
   URL url = new URL(arg[0]);
   URLConnection c = url.openConnection();
   c.connect();
   String type=c.getContentType();
   if (type.startsWith("text")){
     Reader in;
     in=new InputStreamReader(c.getInputStream());
     BufferedReader bin=new BufferedReader(in);
     String ligne;
```

Exemple d'utilisation de la classe URL (2)

```
      while ( (ligne=bin.readLine()) != null)
        System.out.println(ligne);
      bin.close();
    }
    else {
      System.out.print("Fichier de type : ");
      System.out.println(c.getContentType());
    }
   }
   catch(Exception e){System.out.println(e);}
  }
}
```


Connection réseau bas niveau

- communication entre 2 programmes : via une connection passant par un "port" (logique) d'une machine "hôte"
- un "socket" ("prise") = extrêmité d'une connection
- écrire un client : classe `Socket`
  - constructeur `Socket(String host,int port)`
  - ouverture flux I/O : `getInputStream()`, `getOutputStream()`
  - fermeture connection : `void close()`

Connection réseau bas niveau (2)

- écrire un serveur : classe `ServerSocket`
  - constructeur : `ServerSocket(int port)`
  - méthode pour attendre connexion de clients : `Socket accept()`
  - fermeture serveur : `void close()`
- manipulation directe de paquets : classes `DatagramPacket` et `DatagramSocket`

Exemple de Client

```
import java.net.*;
import java.io.*;
class ClientMiroir {
 public static void main(String[] arg){
  try {
   // ouverture socket (port 9999, host arg[0]
   Socket client = new Socket(arg[0],9999);
   // ouverture flux écriture sur socket
   OutputStream out = client.getOutputStream();
   // ouverture flux lecture sur socket
   InputStream in = client.getInputStream();
    for(int i=0;i<10;i++)  {
          String s = "Client Miroir "+i ;
```

Exemple de Client (2)

```
    // recopie sur socket
    out.write(s.getBytes()); out.write('\n');
    out.flush();
    // lecture sur socket
    byte [] buf = new byte[1000];
    in.read(buf); s = new String(buf);
    System.out.println(s);
   }
  } catch (IOException e) {
     System.out.println(e);}
 }
}
```


Exemple de Serveur

```
import java.util.*;
import java.net.*;
import java.io.*;
class ServeurMiroir {
 public static void main(String[] arg){
  try {
   // mise en route du serveur sur port 9999
   ServerSocket serveur = new ServerSocket(9999);
   // attente connexion client
   Socket client = serveur.accept();
   // ouverture flux lecture sur socket
   InputStream is=client.getInputStream();
   // ouverture flux écriture sur socket
   OutputStream out = client.getOutputStream();
```

Exemple de Serveur (2)

```
// fonctionnement du serveur
   StringBuffer buf = new StringBuffer();
   do {
    int car=is.read();
    if (car==-1) break;
    else if (car=='\n'){
      buf.reverse();
      out.write(buf.toString().getBytes());
      out.flush(); buf = new StringBuffer();
    }
    else buf.append((char)car);
   } while (true);
  }catch (IOException e) {
    System.err.println(e); }
  }}
```


ACCES BASES DE DONNEES : `java.sql`

- SQL (Standard Query Langage) = langage standard d'accès aux Bases de Données
- paquetage `java.sql` contient des classes permettant de se connecter à une base de données, puis de lui envoyer des instructions SQL (et de récupérer le résultat des requêtes)
- paquetage aussi connu sous le nom de JDBC (Java DataBase Connectivity)

Principe de programmation

- 1/ on charge le driver de BD par `Class.forName(nomClasseDriver)`
- 2/ on se connecte : `Connection c = DriverManager.getConnection(String url)` où `url` est de la forme `jdbc:nom_driver:nom_base`
- 3/ on crée un objet `Statement` à partir de la connection : `Statement s = c.createStatement();`

Principe de programmation (2)

- 4/ on peut ensuite passer directement les instructions SQL à l'objet `Statement` :
  - modifications : `s.executeUpdate(String commandeSQL)`
  - requête : `ResultSet rs=s.executeQuery(String commandeSQL)`
  - exploitation du `ResultSet` :
    - ligne suivante : `rs.next()`
    - lecture champ : `rs.getString(nom)`, `rs.getFloat(n)`

PROGRAMMATION DISTRIBUEE : `java.rmi`

- rmi = Remote Method Invocation
  (appel de méthode à distance)
- sorte de mini CORBA, mais spécifique à Java
- facilite la programmation client-serveur en permettant la manipulation d'objets distants à travers réseau

Principe

- une interface, connue du serveur et des clients, qui définit les méthodes appelables à distance, et doit dériver de l'interface **`Remote`**
- une classe pour l'objet distant qui implémente **`Remote`** et dérive de la classe **`UnicastRemoteObject`**
- un programme principal qui crée des objets distants et les enregistre auprès du service de nommage par : **`Naming.rebind(nom,obj)`**
- les programmes clients obtiennent une référence aux objets distants par : **`Naming.lookup(nom_url)`** où `nom_url` est de type `rmi://host/nom`

Utilisation

- compiler les sources Java de l'interface, de la classe de l'objet distant, des programmes serveur et client
- appliquer **`rmic`** à la classe des objets distants (pour créer les classes "associées") :
  `rmic MaClasseDistante`
- lancer le serveur de nom : `rmiregistry &`
- lancer le programme serveur, puis les clients

Exemple

Interface

```
import java.rmi.*;
public interface Compte extends Remote {
  void deposer(int x) throws RemoteException;
  void retirer(int x) throws RemoteException;
  int lireSolde() throws RemoteException; }
```

Objets distants et serveur RMI

```
import java.rmi.*;
import java.rmi.server.*; // pour UnicastRemoteObject
public class CompteDist extends UnicastRemoteObject
                                implements Compte {
 private int solde=0;
 public CompteDistant() throws RemoteException{}
 public int lireSolde() throws RemoteException {
   return solde;}
 public void deposer(int x) throws RemoteException{
   solde+=x;}
 public void retirer(int x) throws RemoteException{
   solde-=x;}
 /** programme serveur créant les comptes */
 public static void main(String[] arg) {
   CompteDistant cd1,cd2;
   try {
    // création des comptes
```

Exemple (suite)

Client RMI

```
import java.rmi.*;
public class AccesCompte {
 public static void main(String[] arg) {
  try {
   // URL du compte distant
   String url="rmi://vander/"+arg[0];
   // pour gestion sécurité
   System.setSecurityManager(
              new RMISecurityManager());
   // récupération de l'objet distant
   Compte c=(Compte)Naming.lookup(url);
   // appels de méthode
   if (arg[1].equals("solde")) {
      System.out.print("solde "+arg[0]+" = ");
       System.out.println(c.lireSolde()); }
   else if (arg[1].equals("depot")) {
      c.deposer(Integer.parseInt(arg[2]));
      System.out.print("depôt de "+arg[2]);
       System.out.println(" effectué"); }
```


LES JAVABEANS : `java.beans`

- Beans : composants logiciels réutilisables
- conçus pour être assemblés aisément (et le plus souvent visuellement) à l'aide d'outils de développement intéractifs
- un bean se définit par :
  - ses propriétés (attributs persistants)
  - les événements qu'il reconnait
  - ses méthodes de traitement des événements

JavaBeans (suite)

- l'outil de développement doit pouvoir reconnaître automatiquement les propriétés paramétrables du bean
- possible grâce à l'introspection Java (cf. classe **`Class`**, ...) et au respect de règles de conception ("design patterns") :
  - pour chaque propriété, méthode
    `<TypeProp> get<NomProp>()` et
    `void set<NomProp>(<TypeProp> val)`
  - …