Programmation JAVA

Sun service formation

Projet : Programmation JAVA

Copyright Sun Service Formation

Réf. Sun : SL210

Révision : E-beta

Date : 24/12/99

Sun Microsystems

Programmation JAVA

Sun service formation

Projet : Programmation JAVA

Copyright Sun Service Formation

Réf. Sun : SL210

Révision : E-beta

Date : 24/12/99

Sun Microsystems

Programmation JAVA

Sun service formation

Projet : Programmation JAVA

Copyright Sun Service Formation

Réf. Sun : SL210

Révision : E-beta

Date : 24/12/99

Sun Microsystems

Programmation JAVA

Sun service formation

Projet : Programmation JAVA

Copyright Sun Service Formation

Réf. Sun : SL210

Révision : E-beta

Date : 24/12/99

Sun Microsystems

Programmation JAVA

Sun service formation

Projet : Programmation JAVA

Copyright Sun Service Formation

Réf. Sun : SL210

Révision : E-beta

Date : 24/12/99

Sun Microsystems

Programmation JAVA

Sun service formation

Projet :	Programmation JAVA
Copyright	Sun Service Formation
Réf. Sun :	SL210
Révision :	E-beta
Date :	24/12/99

Sun Microsystems France

Programmation JAVA

Sun service formation

Projet :	Programmation JAVA
Copyright	Sun Service Formation
Réf. Sun :	SL210
Révision :	E-beta
Date :	24/12/99

Sun Microsystems France

Programmation JAVA

Sun service formation

Projet :	Programmation JAVA
Copyright	Sun Service Formation
Réf. Sun :	SL210
Révision :	E-beta
Date :	24/12/99

Sun Microsystems France

Programmation JAVA

Sun service formation

Sun Microsystems France S.A.
Service Formation
143 bis, avenue de Verdun
92442 ISSY LES MOULINEAUX Cedex
Tel 01 41 33 17 17
Fax 01 41 33 17 20

Intitulé Cours :	Programmation JAVA
Copyright	Sun Service Formation
Réf. Sun :	SL210
Révision :	E-beta
Date :	24 / 12 / 99

Sun Microsystems France


Protections Juridiques

AVERTISSEMENT

Table des Matières

Présentation de Java 1

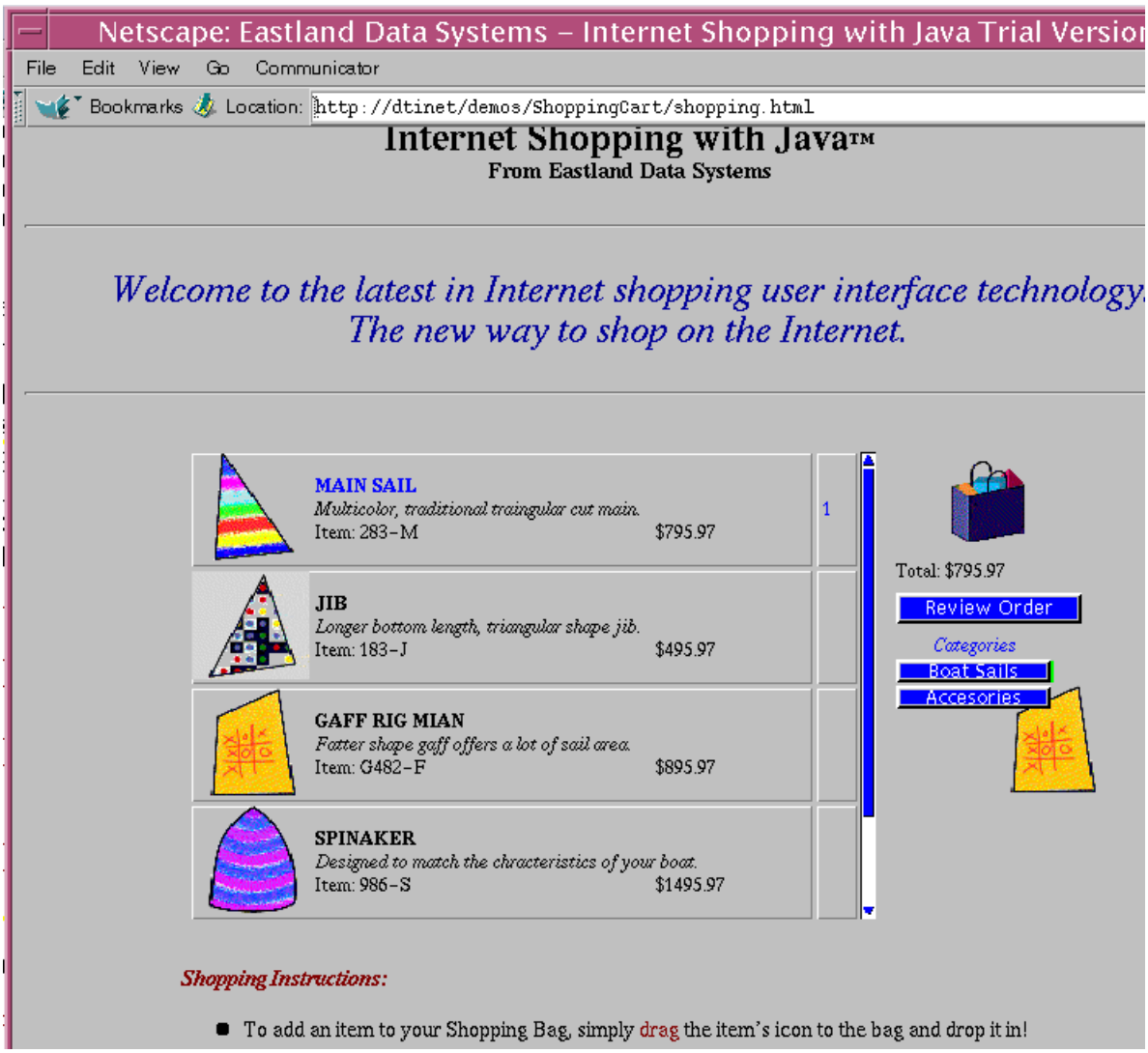
Points essentiels

Une introduction aux mécanismes fondamentaux de la technologie Java:

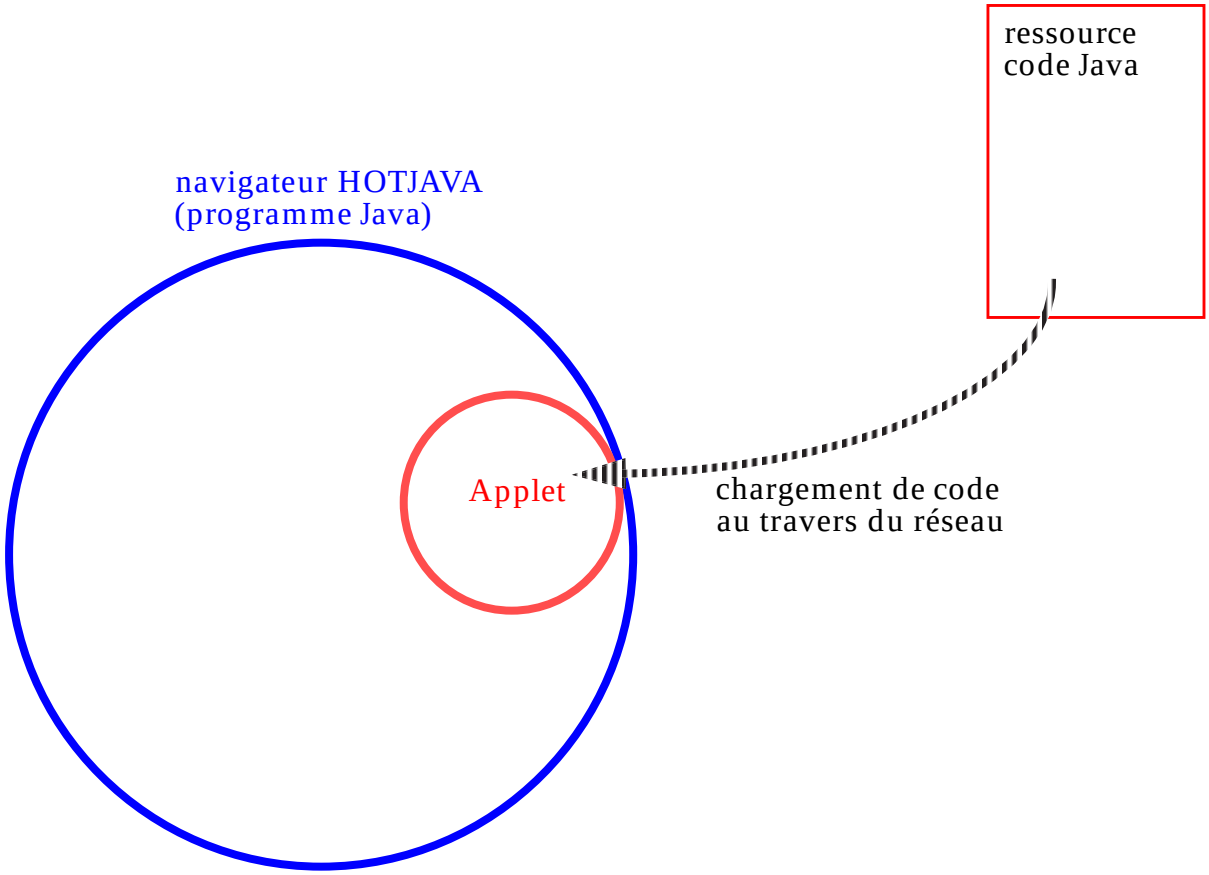
- Caractéristiques principales de la technologie Java.
- Code source, code exécutable
- La machine virtuelle
- Mécanismes d'exécution, mécanismes de contrôle
- Un premier programme Java.

Applications, Applets, ...: à la découverte de Java

Une Applet est un programme qui s'exécute au sein d'une page HTML Un tel programme est chargé dynamiquement par le navigateur qui trouve sa référence dans le code HTML et qui demande le chargement de ce code depuis le serveur HTTP.



Applications, Applets,...: à la découverte de Java



Considérons la situation suivante:

- Un navigateur HotJava est un programme autonome réalisé entièrement à l'aide du langage de programmation Java.
- Au travers d'une page du Web il télécharge un "code" Java (une Applet) depuis un serveur et exécute ce code

Quelles caractéristiques importantes du langage Java dévoile ce comportement?

A la découverte de Java

L'exemple précédent révèle quelques caractéristiques essentielles de Java:

- Le serveur HTTP envoie le même code à tous ses clients, (il ne sait pas s'il s'agit d'un PC, d'un macintosh, d'une station unix, etc.)
  Le code exécutable Java est un **code portable** : il est capable de s'exécuter sur des machines ayant des architectures et des systèmes d'exploitation différents.

- Le code est exécuté dynamiquement: l'application Java (HotJava) est capable, dans le cadre d'une autre tâche, de découvrir une instruction l'enjoignant de charger un autre code Java (a priori inconnu!) et de déclencher son exécution.
  Cette **exécution dynamique** a un certain nombre de corollaires:

  - Il faut que le code importé ne puisse pas réaliser des opérations non souhaitées sur le site local. Un **mécanisme de sécurité** est intégré au langage.

  - La coopération dynamique entre différents codes suppose une modularité du code très bien étudiée. Java est un **langage à Objets** particulièrement modulaire (ex. pas de variables globales, pas de fonctions isolées, etc...).

  - L'exécution de l'Applet se déroule pendant que le programme principal continue à s'exécuter (y compris pour lancer d'autres Applets). Java permet d'accéder à des mécanismes d'exécution en parallèle (**processus légers** `-threads-`).

  - ...

Le langage de programmation Java

Conçu à l'origine comme environnement de développement pour des applications portables destinées à de l'électronique grand-public, Java a connu une forte expansion avec l'explosion du Web.

La communauté des développeurs a rapidement adopté ce langage pour sa clarté, sa puissance d'expression, son organisation de langage à objets, sa portabilité,...

Lorsqu'on parle du langage Java on fait référence à :

- Un langage de programmation (pour une définition plus formelle voir JLS : Java Langage Specification)
- Un environnement de développement
- Un environnement d'exécution
- Un environnement de déploiement

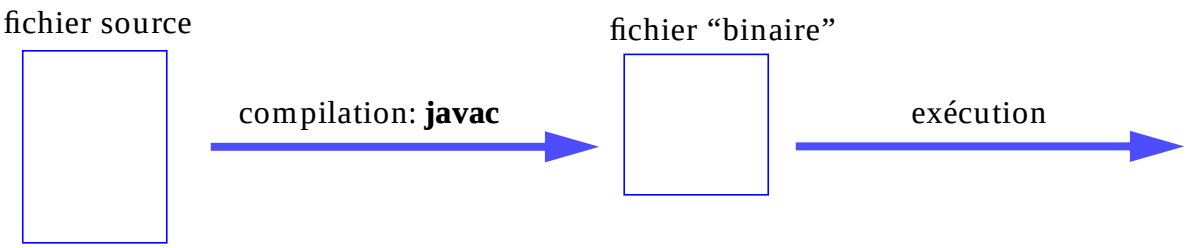
En fait les caractéristiques de Java ont permis l'éclosion de nouvelles formes d'organisation des systèmes d'information et de nombreuses technologies apparentées. Au sens strict il faut donc distinguer les différents concepts qui utilisent le vocable "Java" ("technologie Java", "langage de programmation Java").

Dans la suite de ce document chaque fois que nous utiliserons le mot "Java" sans préciser le contexte on devra comprendre "langage de programmation Java".

Dans les pages qui suivent nous allons introduire les mécanismes de fonctionnement de Java.

Code source, code exécutable

génération de code exécutable :



fichier source : **Calimero.java**

```
public class Calimero {
   public static void main (String[] args) {
      System.out.println("Hello World!") ;
   }
}
```

fichier binaire : **Calimero.class** (468 octets)

```
cafe babe 0003 002d 0020 0800 1507 0010
0700 1a07 001b 0700 1c0a 0004 0009 0900
0500 0a0a 0003 000b 0c00 0f00 0c0c 001e
0017 0c00 1f00 0d01 0003 2829 5601 0015
....
```

décompilation du fichier binaire :

```
...
Method void main(java.lang.String[])
   0 getstatic #7 <Field java.io.PrintStream out>
   3 ldc #1 <String "Hello World!">
   5 invokevirtual #8 <Method void println(java.lang.String)>
   8 return
...
```

Code Source, Code exécutable

Ce qui est exécuté par une application autonome (ou par un navigateur dans le cas d'une Applet) c'est un code binaire obtenu par compilation d'un programme source.

Un programme source est un texte contenant des instructions en langage JAVA. En général on s'attend à ce qu'un fichier `XXX.java` contienne la description d'un élément du langage appelé "classe" de nom `XXX`.

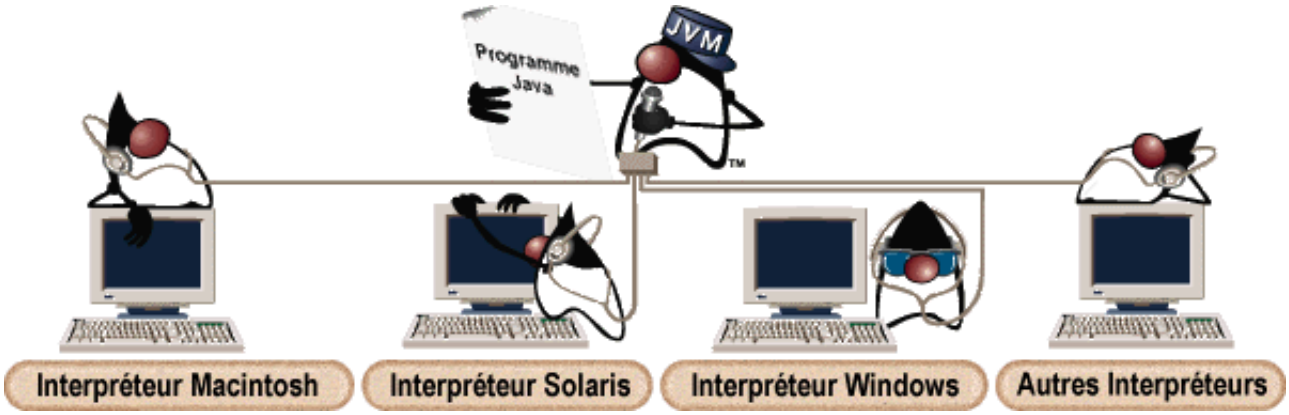
Le compilateur (`javac`) permet de générer un fichier exécutable de nom `XXX.class`.

C'est un fichier de ce type qui est (télé)chargé par un navigateur pour déclencher l'exécution d'une Applet .

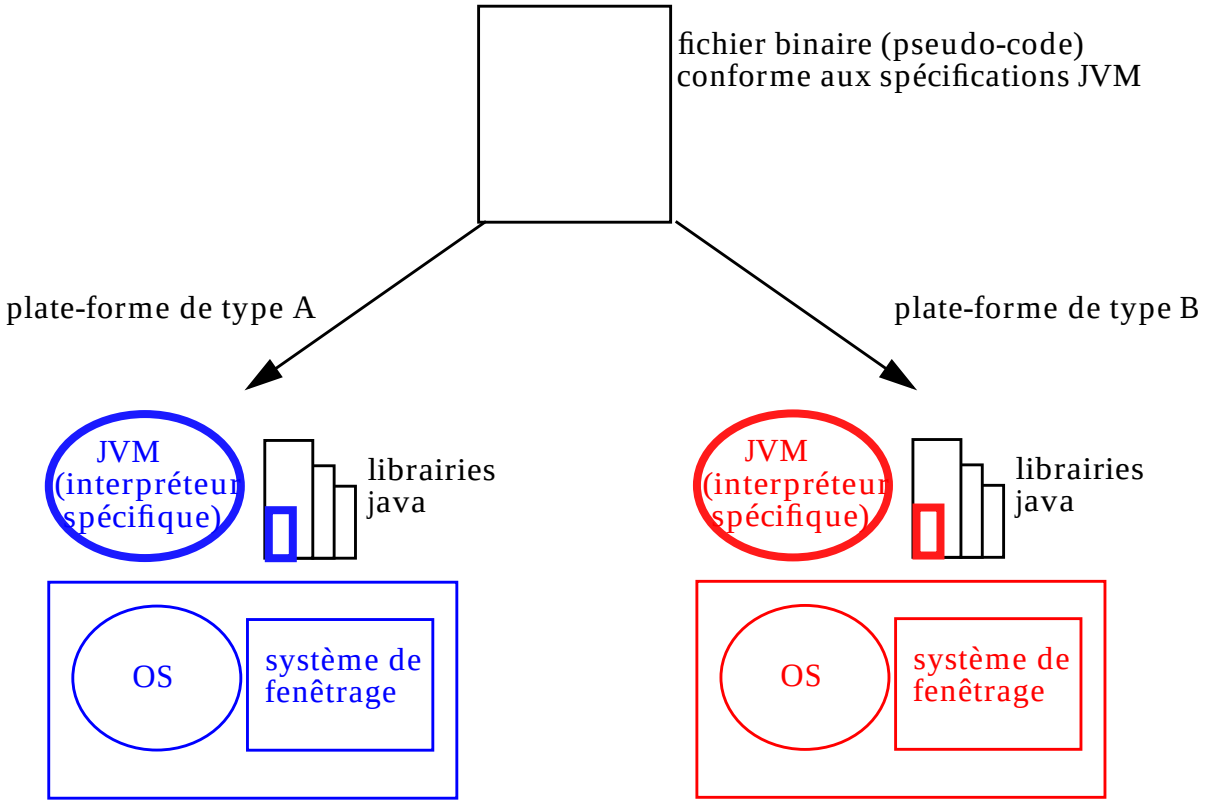
Pour démarrer une application autonome on a besoin d'un fichier ".class" disposant d'un point d'entrée (`main`) et d'un exécuteur Java.

Portabilité : la machine virtuelle

L e même “code” est exécuté par des plate-formes différentes :



Les JVM locales



La machine virtuelle

Que ce soit pour un navigateur ou pour une application autonome il y a un exécuteur de code Java.

Ce programme interprète le code binaire Java comme un code d'une **machine virtuelle**. Les spécifications du langage Java définissent très précisément la description et le comportement de cette machine virtuelle (c'est une "machine" à pile).

Le code "binaire" Java est en fait un pseudo-code (*bytecode*) portable: seules les implantations de la JVM (Java Virtual Machine) diffèrent selon les plate-formes.

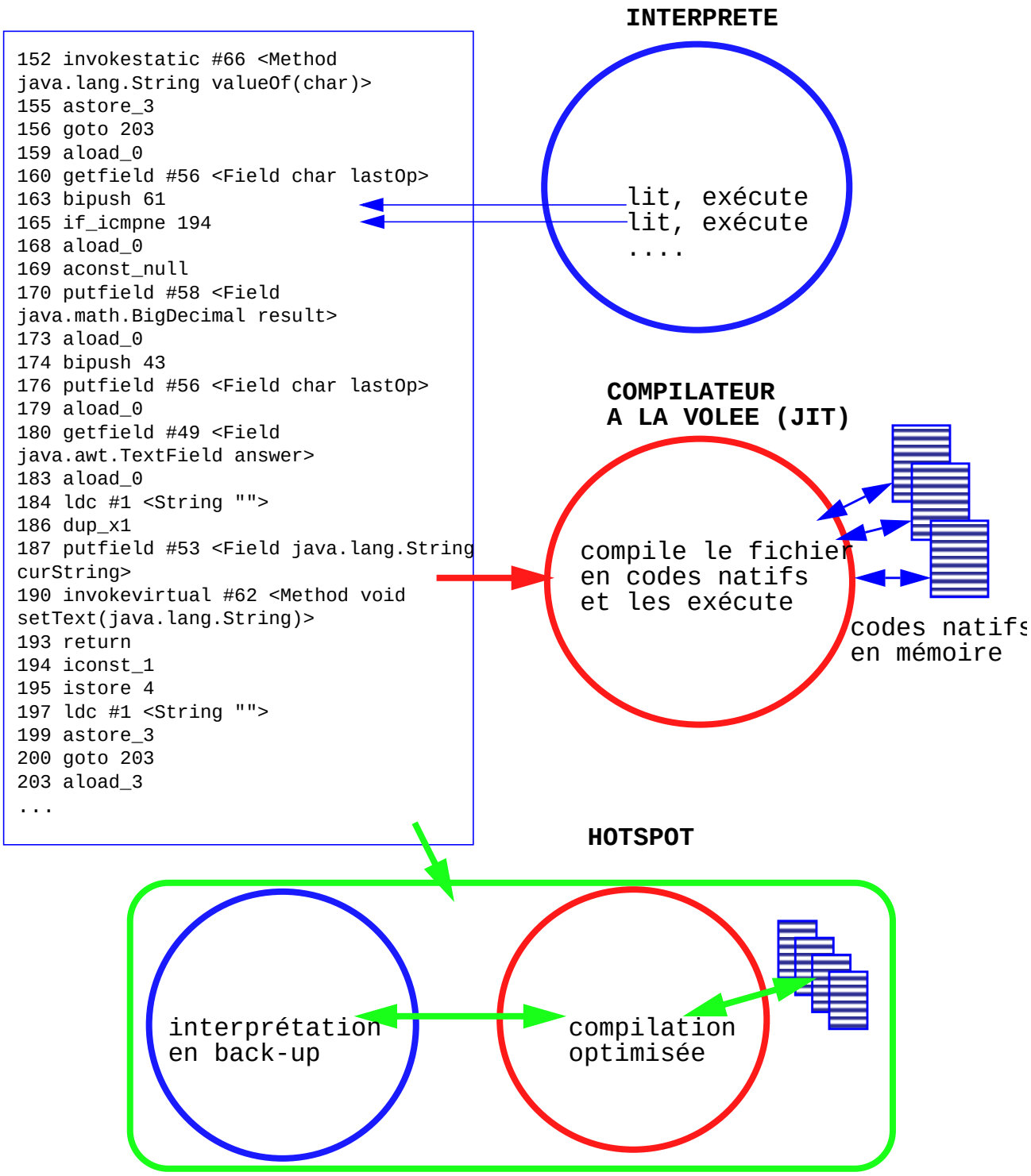
Pour exécuter un code Java sur un Système et une architecture donnée on doit donc trouver localement:

- Une implantation de la JVM (au sein d'un navigateur, ou pour lancer des applications autonomes)
- Les librairies constituant le noyau des classes standard Java. La plus grande partie est constituée de code Java (une partie est du code binaire spécifique à la plate-forme locale).

La *Java Virtual Machine Specification* définit entre autres:

- Le jeu d'instruction du pseudo-code
- La structure des registres et l'organisation de pile
- L'organisation du tas et de la mémoire (recupération automatique par *garbage-collector*)
- Le format des fichiers .class

Différents modes d'exécution



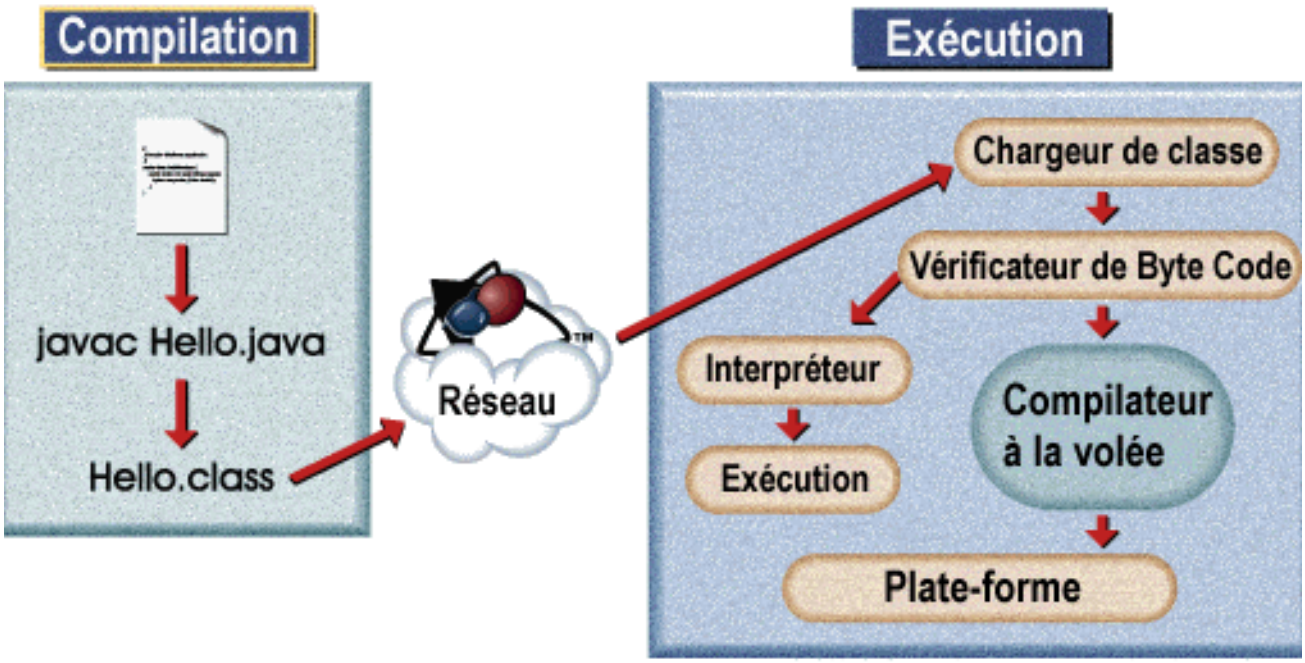
Les différents modes d'exécution

- Interprétation: Le programme qui implante le comportement de la JVM, lit le fichier de *bytecode* et exécute les opérations au fur et à mesure. C'est un interprête d'un code de bas niveau.
  Les interprêtes font des progrès constants en performance depuis les premières versions de Java.

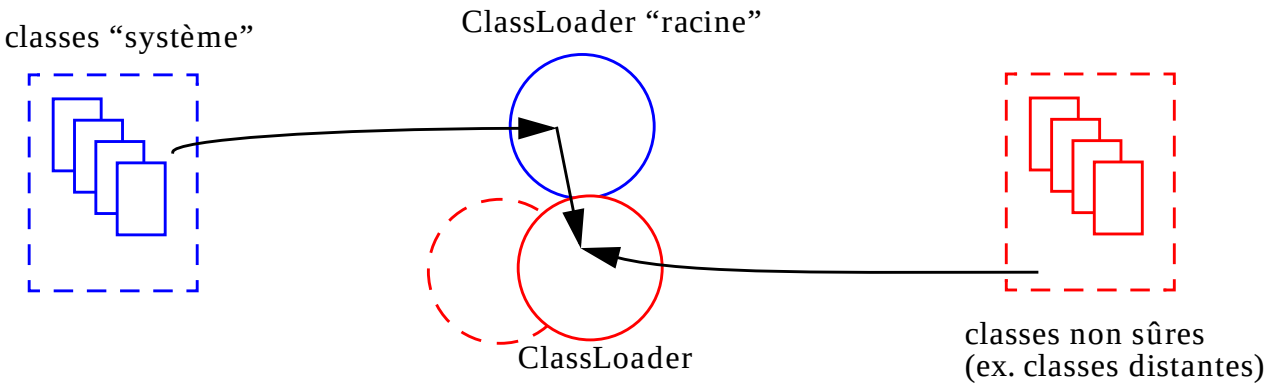
- JIT (Just-In-Time, compilation à la volée): Le programme chargé d'exécuter le code Java, compile directement ce pseudo-code en code natif de la plateforme locale, puis l'exécute. On fait le pari que le temps perdu par la compilation sera rattrapé du fait de la rapidité d'exécution du code natif.

- HotSpot : stratégies d'optimisation très complexes. L'optimiseur décide, au moment de l'exécution, s'il y a lieu de compiler ou d'interpréter (ex. boucle exécutée *N* fois : la situation n'est pas la même si *N* vaut 2 ou vaut des milliers). Par ailleurs ne cherche pas à compiler des cas rares et difficiles à compiler, fait de l'expansion de code (*inlining*), etc.

Sécurité : chargeur , vérificateur, gestionnaire de sécurité



Le chargement des classes



Sécurité : chargeur , vérificateur, gestionnaire de sécurité

Le chargement de code au travers du réseau pose un certain nombre de problêmes de sécurité :

- Quelle est le code Java que j'exécute? Est-il possible de m'adresser au travers du réseau un code (inamical?) qui remplace un de mes codes Java locaux?
  Le chargement de code au travers du réseau est de la responsabilité d'un **ClassLoader** (une classe Java).
  Un ClassLoader cherchera toujours à charger une classe dans les ressources système avant de s'adresser à d'autres ressources.

- Le code compilé téléchargé peut-il être corrompu et donc faire faire à la JVM locale des opérations illégales?
  Le **vérificateur de ByteCode** contrôle la conformité du code chargé (validité des opérations, validité des types, contrôles d'accès,...).

- Le code importé peut-il réaliser localement des opérations que je ne souhaite pas (obtention d'une information confidentielle, écriture dans le système de fichiers local, etc..)?
  Un **SecurityManager** fait respecter une **politique de sécurité**.
  Un objet **AccessControler** (une classe Java) est chargé de contrôler tous les accès critiques et de faire respecter une **politique de sécurité**
  Ainsi, par défaut, le code d'une Applet ne pourra pas connaître des informations critiques sur le site local (système de fichiers login, etc.), ne pourra pas écrire dans le fichiers locaux, déclencher des applications locales ou atteindre des sites sur le réseau autre que le site qui a distribué l'Applet elle-même.
  Des ressources locales permettent de définir des droits particuliers accordés à des intervenants dûment accrédités:

```
grant codebase “http://www.montaigne.fr/”, signedBy “laBoetie”{
permission java.io.FilePermission "/tmp/*", "read,write";
};
```

Exemple de fichier de configuration de sécurité

Robustesse: contrôles à la compilation et au run-time

La fiabilité de java s'appuie aussi sur de nombreux contrôles exercés soit par la machine d'exécution (*runtime*) soit par le compilateur (*compile-time*).

Le langage est très peu laxiste et de nombreux contrôles peuvent se faire à la compilation :

- vérifiation des "contrats de type" : affectation entre types compatibles, peu ou pas de transtypage implicite, obligation pour certains objets de remplir des obligations (*interface*),...
- vérification des obligations des méthodes (fonctions) : compatibilité des paramètres, obligation de retourner un résultat compatible avec le type déclaré en retour, ...
- vérification des droits d'accès entre objets : certains détails sont cachés et doivent le rester, interdictions éventuelles de redéfinition,..
- vérifie si une variable locale risque d'être utilisée avant d'avoir été initialisée, vérifie si certaines variables (*blank final*) ne sont bien initialisées qu'une et une seule fois,...
- signale les obsolescences (évolutions de Java).

A l'exécution la JVM vérifie dynamiquement la validité de certaines opérations :

- accès hors limites dans des tableaux
- incompatibilités entre objets découvertes au *runtime*
- opérations illégales, etc.

Un mécanisme particulier de propagation et de traitement des erreurs permet d'éviter un effondrement intempestif et incontrolé des applications.

Robustesse : récupération des erreurs, gestion de la mémoire,

Les exceptions :

Dans le cas où le système d'exécution détecte une erreur au runtime, il génère un objet particulier appelé **exception**. Cet objet contient des informations sur l'incident et le système remonte la pile d'exécution à la recherche d'un code spécifique chargé de traiter l'incident. Si un tel code n'existe pas la machine virtuelle recherche une action par défaut (comme l'affichage d'un message) avant de terminer la tâche concernée

Les classes Java sont susceptibles de génerer toute sortes d'exceptions pour rendre compte des incidents spécifiques à telle ou telle action. Les programmeurs peuvent définir leurs propres exceptions, les déclencher et écrire du code pour les récupérer et prendre des mesures appropriées.

La gestion de mémoire :

Dans de nombreux langages comme C/C++ la gestion dynamique des objets en mémoire est une source constante de bugs très subtils et très coûteux en maintenance. Les "fuites mémoire" peuvent passer au travers des tests de validation et nécessitent des outils spécifiques de recherche et de validation.

Java entre dans la catégorie des langages disposant d'un "glaneur de mémoire" (ou "ramasse-miettes" -*garbage collector*-). On a un processus léger, de priorité basse, qui est réactivé de temps en temps et qui récupère automatiquement la mémoire occupée par des objets qui ne sont plus référencés. De plus ce processus compacte la mémoire occupée et gère de manière optimum la place disponible.

Cette gestion est facilitée par le fait que le programmeur n'a aucun moyen d'accéder directement à la mémoire (pas de pointeurs).

Une application Java simple

Voici un fichier source nommé **Calimero.java**

```
//
// l'application "Hello World" en version française
//
public class Calimero {
   public static void main (String[] args) {
      System.out.println("Bonjour Monde cruel!") ;
          }
}
```

Ces quelques lignes représentent un programme autonome minimum pour imprimer un message à l'écran. Quelques commentaires sur ce source

- Les trois première lignes (commençant par "//") constituent des commentaires.

- Le bloc suivant (`public class Calimero {....}`) constitue une définition de classe. A ce niveau du cours nous dirons qu'une classe désigne un module fonctionnel. Quelques remarques :

  - Dans le langage Java il n'y a pas de code en dehors des classes. En corollaire il n'existe pas de variable ou de fonction "isolée" (c'est à dire qui ne soit pas rattachée à une classe) :
    ligne 3 du programme `println()` est rattachée à une "variable" de classe nommée `out` elle même rattachée à la classe standard `System` . Les raisons de ces différents rattachements seront expliquées ultérieurement.

  - Pour toute classe déclarée dans un fichier le compilateur créera un fichier ".class" de même nom. Il y a, au plus, une seule classe marquée `public` dans un fichier source et le nom du fichier doit correspondre exactement au nom de la classe (une classe publique "Xyz" doit se trouver dans le fichier Xyz.java).

- Pour commencer l'exécution d'une application autonome à partir d'une classe donnée l'interprête recherche dans cette classe le point d'entrée standard désigné exactement par `public static void main (String[] `*`nomargument`*`)`

Une application Java simple

Compilation et exécution de Calimero.java

- A partir du fichier source Calimero.java la compilation se fait par ;

```
javac Calimero.java
```

- Si tout se passe bien (si le compilateur de renvoie pas d'erreur) on trouvera dans le répertoire courant un fichier exécutable "Calimero.class".

- Pour exécuter l'application faire :

```
java Calimero
```

Outre les éventuels problèmes d'accès aux commandes javac et java (problèmes de PATH) les problèmes les plus courants concernent :

- A la compilation: non correspondance entre le nom de la classe publique et le nom du fichier (la correspondance doit être exacte - y compris au niveau des minuscules / majuscules).

- A la compilation: erreurs de saisie dans le texte -erreurs de syntaxe, erreurs sur les noms, etc.-

- A l'exécution: l'exécuteur ne trouve pas la classe, soit parceque le nom de classe est mal saisie (erreur courante : "java Calimero.class"), soit parceque l'exécuteur ne trouve pas la ressource correspondante. En effet la commande java recherche la classe demandée dans un ensemble de répertoires comprenant des répertoires standard d'installation et des répertoires définis par la variable d'environnement CLASSPATH. Si cette variable est mal définie l'exécuteur risque de ne pas trouver le bon fichier.

- Une autre source potentielle de déboires est une mauvaise définition du "main" qui serait absente ou qui ne correspondrait pas exactement à la signature
  `public static void main (String[] )`

Introduction à la modularité: classes et packages

Comme nous l'avons vu une *classe* constitue une unité de modularité qui permet de regrouper un certain nombre de fonctionnalités. Le Kit de développement logiciel Java 2 (SDK) offre un ensemble de classes standard (les librairies "noyau") qui implantent des services pour les tâches courantes de programmation. Toute JVM doit avoir accès localement à ces classes.

Les classes sont regroupées en *packages* en fonction de leurs affinités et de leur domaine d'action. Ainsi on retrouve dans le SDK un certain nombre de packages standard dont, en particulier, :

- `java.lang` : contient les classes fondamentales du langage comme `String` (les chaînes), `Thread` (les processus légers), `System`, `Math`, etc.
- `java.util` : contient des classes utilitaires pour le programmeur comme les collections (`Vector`, `HashMap`, `Properties`,..), les dates (`Calendar`), l'internationalisation (`Locale`, `ResourceBundle`,..)
- `java.awt` : construction et gestion d'interfaces graphiques
- `java.io` : entrées / sorties portables
- `java.net` : opérations au travers du réseau
- ....

Les packages sont organisés en hiérarchies:

```
java.rmi
java.rmi.server
...
javax.rmi //depuis 1.3
```

Architecture générale d'un fichier source

Bien que cela ne soit pas strictement obligatoire il est très vivement conseillé de développer des classes Java comme faisant partie d'un package. Dans ces conditions l'architecture générale d'un fichier source Java se présente obligatoirement de la manière suivante (abstraction faite des commentaires) :

- une déclaration d'appartenance à un package

```
package fr.macompagnie.finances.calculs.swap ;
        // noter l'organisaton hiérarchique des packages
```

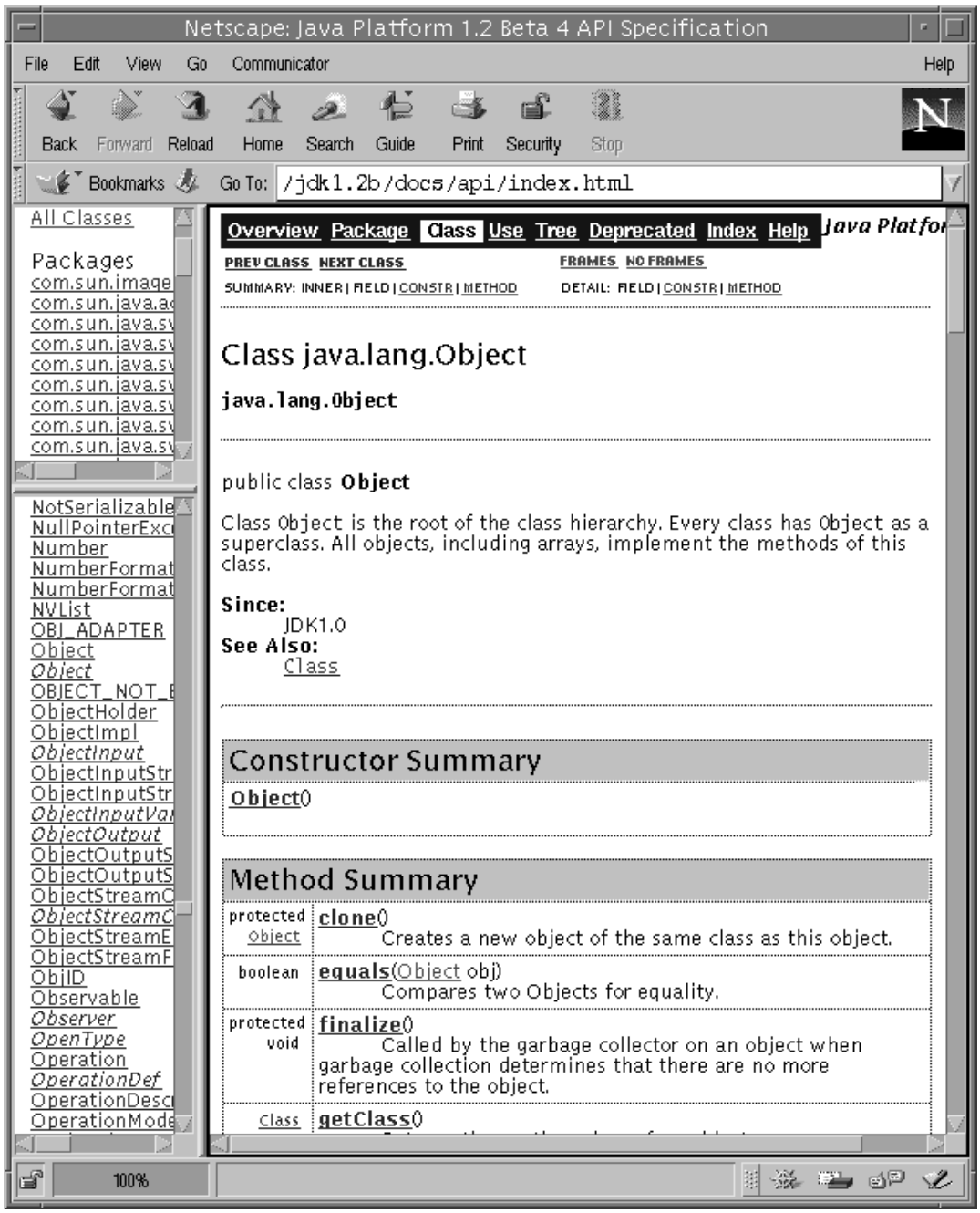
- éventuellement une liste de clauses `import` qui permet au compilateur de connaître les classes d'autres packages pour vérifier la cohérence du source local et lever toute ambiguité. Le package java.lang est toujours visible et n'a pas besoin de figurer dans une clause `import`.

```
import java.io.ObjectInputStream ;
import java.io.FileInputStream ;
import fr.macompagnie.finances.calculs.amortissements.* ;
        // permet de voir toutes les classes appartenant
        // directement à ce package
        // mais pas les classes des sous-packages
```

- les définitions de classes et d'interfaces (un autre type de définition de premier niveau que nous verrons ultérieurement).

```
public class Swap {
        .....
}
/* noter que le "vrai" nom de la classe (celui figurant dans le
binaire) est "fr.macompagnie.finances.calculs.swap.Swap" et que
c'est ce "vrai" nom qu'il faudra utiliser si vous invoquez
directement un "main" de cette classe au travers de la commande
java
*/
```

Utiliser la documentation



Utiliser la documentation

L'installation d'un SDK se fait en deux parties :

- installation de la JVM , des utilitaires et des librairies propre à votre architecture matérielle et logicielle. Voir `http://java.sun.com`

- installation de la documentation (identique pour toutes les plateformes).
  Consulter le fichier index.html dans le sous-répertoiree docs/api du répertoire d'installation de Java (bien entendu cette documentation peut-être installée pour être consultée en réseau).

La documentation de reférence a une structure d'hypertexte consultable par un navigateur HTML. Il est essentiel d'apprendre à rechercher des informations par ce biais.

La documentation des APIs Java a été directement extraite des sources de Java lui-même au moyen de l'utilitaire *javadoc*. Java dispose d'un formalisme standard qui permet d'inclure de la documentation directement dans le code. De cette manière vous pourrez vous-même documenter vos propres APIs.

Exercices :

*Exercice * :*

Reprendre le programme Calimero.java, le saisir avec un éditeur, le compiler et l'exécuter.

*Exercice ** :*

Reprendre le programme précédent et faire en sorte qu'il appartienne à un package de nom "chap1".

A ce stade vous aurez les plus grandes difficultés pour exécuter votre programme. Pourquoi? Parceque l'exécuteur tente de faire correspondre une hiérarchie de répertoires à la hiérarchie des packages. En d'autres termes il s'attend à trouver les classes du package "chap1" dans un répertoire de nom "chap1".

Quelles conclusions en tireriez-vous pour l'organisation des fichiers sources et des fichiers ".class" en sachant que :

```
javac -d repertoirecible sourcejava
```

permet d'installer le binaire correspondant à partir de "repertoirecible". Faites des essais jusqu'à obtenir une configuration satisfaisante de l'option par défaut de javac (écrire un script) et de la valeur de CLASSPATH.

Syntaxe, identificateurs, mots-clefs, types 2

Points essentiels

- Principes généraux de la syntaxe
- Identificateurs, mots-clefs
- Types primitifs
- Types “objets” : une introduction
- Types tableaux
- Allocation des différents types
- Conventions de codage

Syntaxe : généralités

Comme dans de nombreux langages informatiques un source Java est composé de “mots” qui sont délimités par des caractères (ou des ensembles de caractères) particuliers.

Ces “mots” peuvent être :

- des identificateurs : `nomDeVariable`, ...
- des mots-clefs réservés : `if` ,...
- des littéraux : `3.14`, `6.02e-9` , `“une chaîne”`, ...

Les délimitations :

- séparateurs : blancs, commentaires
- séparateurs de structure : `{` , `[` , `;` , ...
- opérateurs : `+` , `*` , `&&` , `>>=` ,...

```
if (variable < 0) {
        variable=0;// un commentaire est-il nécessaire?
}
```

Commentaires

Trois notations sont possibles pour l'insertion de commentaires :

```
// commentaire sur une ligne

/* commentaires sur une ou plusieurs lignes */

/** insertion de documentation */
```

Les commentaires pour la documentation sont placés juste avant une déclaration JAVA et indiquent que son contenu doit être inclus automatiquement dans la documentation générée par l'outil *javadoc*.

Cette documentation est au format HTML.

Les règles de description et de formatage de cette documentation, ainsi que l'utilisation de l'outil *javadoc* se trouvent dans le document `./docs/tooldocs/javadoc/index.html` (sous répertoire d'installation du SDK JAVA 2)

Séparateurs

Une instruction se termine par un point virgule (;).

Elle peut tenir une ou plusieurs lignes.

```
total = ix + iy + iz + ij + ik+ il ; // une instruction
```

est la même chose que :

```
total = ix + iy + iz
          + ij + ik + il ;// deux lignes, une instruction
```

Un bloc regroupe un ensemble d'instructions, il est délimité par des accolades.

```
{ // un bloc
          ix = iy + 1 ;
          iz = 25 ;
}
```

Les blocs peuvent s'emboîter les uns dans les autres :

```
 {
          int ix = iy + 1 ;
          while ( ik < MIN ) { // bloc emboîté
                    ik = ik * ix;
          }
}
```

Les espacements (caractère espace, tabulation, retour chariot) peuvent se répéter à l'intérieur du code source. On peut les utiliser pour améliorer la présentation.

```
if(ix<iy)ix=iy;//compact

if ( ix < iy ) {
          ix = iy ;
} // plus aéré
```

Identificateurs

Dans le langage Java un identificateur commence par une lettre, un trait de soulignement (_), ou un signe dollar ($). Les caractères suivants peuvent également comporter des chiffres.
Les identificateurs opèrent une distinction entre majuscules et minuscules et n'ont pas de limitation de longueur.

Identificateurs valides:

- `identificateur`
- `fenêtre`
- `nomUtilisateur`
- `MaClasse`
- `_var_sys`
- `$picsou`

Les identificateurs peuvent contenir des mots-clés, mais ne doivent pas eux même être des mots-clés: "`longueur`" est un identificateur valide mais "`long`" ne l'est pas.

Consulter en fin de chapitre les conventions usuelles de nommage. Eviter autant que possible les caractères "$" dans les noms

Mots-clés

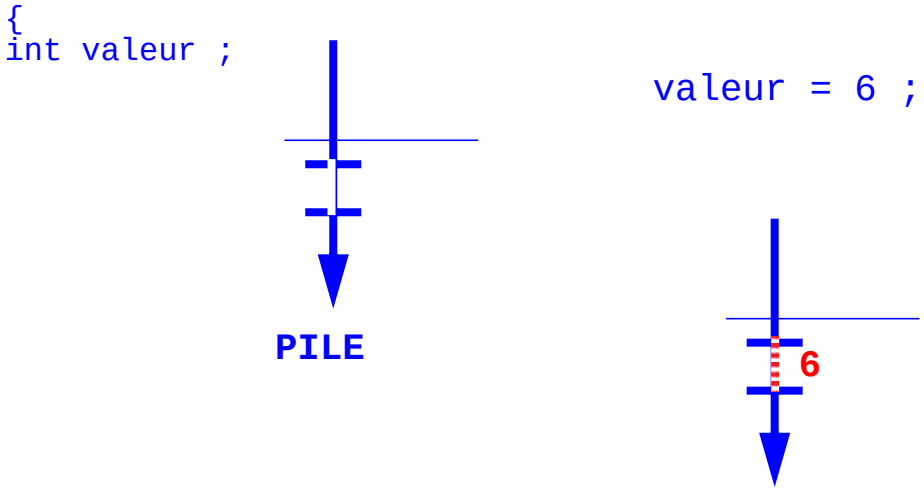
abstract	do	implements	private	this
boolean	double	import	protected	throw
break	else	instanceof	public	throws
byte	extends	int	return	transient
case	*false*	interface	short	*true*
catch	final	long	static	try
char	finally	native	strictfp	void
class	float	new	super	volatile
continue	for	*null*	switch	while
default	if	package	synchronized	

Les littéraux `true`, `false` et `null` sont en minuscules (et non en majuscules comme en C++). Au sens strict il ne s'agit pas de mots-clés.

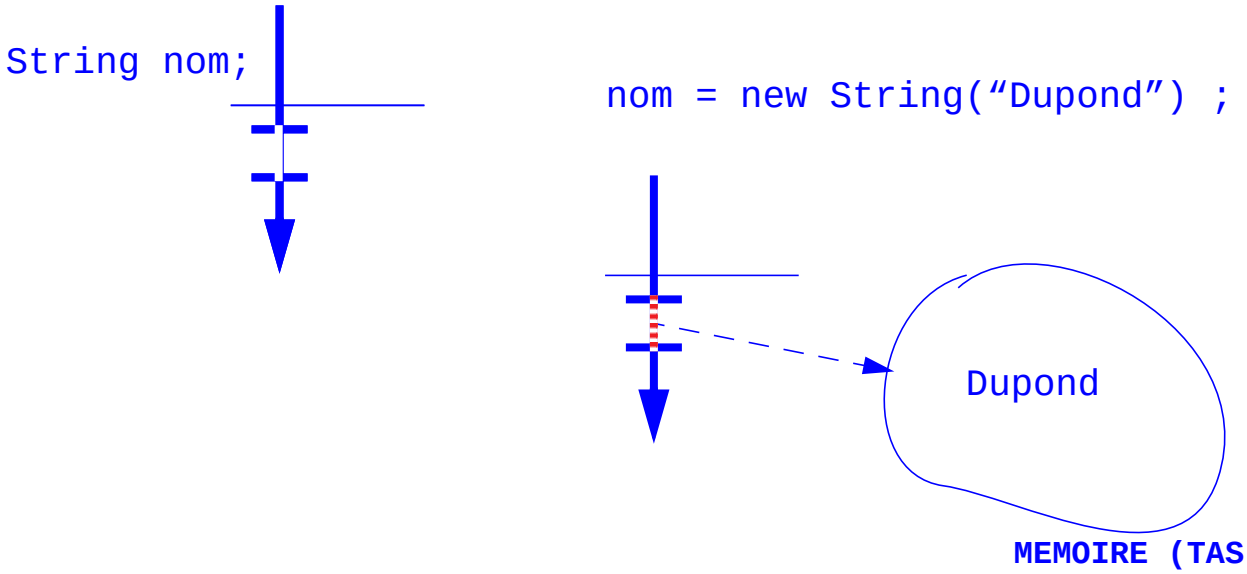
goto et const sont des mots-clés réservés mais inutilisés.
Il n'y a pas d'opérateur sizeof (comme en C++)

Types scalaires primitifs, types objets

Le langage Java distingue les types scalaires primitifs des autres types (objets). La déclaration d'un type scalaire primitif, comme `boolean`, `int`, `float`, alloue un emplacement pour stocker une valeur du type considéré.



La déclaration d'une variable "objet" ne crée pas d'emplacement en mémoire pour stocker l'objet mais crée un emplacement pour stocker une *référence* vers un objet



Types primitifs : logiques

Les valeurs logiques ont deux états : vrai ou faux. En Java une telle valeur est représentée par une variable de type `boolean`. Deux littéraux seulement peuvent être employés pour représenter une valeur booléenne : `true` et `false`.

Voici une déclaration et une initialisation d'une valeur booléenne:

```
//déclare et initialise en même temps
boolean isOK = true ;
```

Attention! Il n'y pas d'équivalences entre les types entiers (comme `int`) et les booléens. Certains langages comme C ou C++ permettent d'interpréter une expression numérique comme représentant une valeur logique
Ceci est interdit en langage Java: lorsqu'une expression booléenne est requise, rien d'autre ne peut la remplacer.

Types primitifs: texte

Un caractère isolé est représenté à l'aide du type `char` .

Une valeur de type char représente un caractère UNICODE (16 bits, non signé) qui permet de représenter des caractères très divers (langues orientales, symboles).

Un littéral de type `char` doit figurer entre deux apostrophes :

```
'a'       // le caractère a
'\t'      // une tabulation
'\u0240'  // un caractère UNICODE
          // code de la forme \uXXXX
          // (4 chiffres hexadécimaux)
```

Une chaîne de caractères peut être représentée au moyen du type `String`.

Le type `String` n'est pas un type scalaire primitif mais un type prédéfini représenté par une *classe*. Cette classe a la particularité d'avoir une notation pour les constantes littérales :

```
String titre = "MENU DU JOUR" ;
String platRecommandé = "Rôti de b\u0153uf\t 10 \u20AC";
          // 10 Euros pour un Rôti de boeuf !
          // chaînes dans le pool de constantes

String autreDupond = new String("Dupond") ;
```

Attention: contrairement à C/C++ il n'y a pas de terminateur \0 en fin de chaîne.

Caractères UNICODE : `http://charts.unicode.org`

Les chaînes `String` sont *immutables*: on ne peut pas, par exemple, modifier un caractère sans passer par la création d'une autre chaîne.

Types primitifs: numériques entiers

Il existe quatre types entiers dans le langage Java : `byte`, `short`, `int`, et `long`.

Tous ces types entiers sont des nombres signés.

La spécification du langage définit une implantation portable indépendante de la plate-forme (ordre des octets, représentation en complément à deux).

Taille implantation	**Type**	**Plage valeurs**
8 bits	**`byte`**	-2^{7} ... $2^{7}-1$
16 bits	**`short`**	-2^{15} ... $2^{15}-1$
32 bits	**`int`**	-2^{31} ... $2^{31}-1$
64 bits	**`long`**	-2^{63} ... $2^{63}-1$

Constantes littérales entières

Les constantes littérales correspondants aux types entiers peuvent se représenter sous forme décimale, octale ou hexadécimale :

```
98          // la valeur décimale 98

077         // le zero en debut indique une notation octale
            // valeur : 63

0xBEBE      // 0x indique une notation hexadécimale
            // valeur : 48830
```

Les expressions concernant des nombres entiers sont de type `int` sauf si on précise explicitement une valeur `long` au moyen d'un suffixe "L". Les "L" minuscules et majuscules sont acceptés, mais il est préférable de ne pas utiliser la minuscule qui peut être confondue avec le chiffre 1.

```
98L         // valeur décimale de type long

077L        // 63 sur un long

0xBEBEL     // 48830 sur un long
```

Types primitifs: numériques flottants

Il existe deux types numériques flottants dans le langage Java : `float` et `double` .

La spécification du langage définit une implantation portable indépendante de la plate-forme (I.E.E.E 754)

Taille implantation	Type
32 bits	**`float`**
64 bits	**`double`**

Une expression littérale numérique est un flottant si elle contient un point, une partie exponentielle (lettre E ou *e*) ou si elle est suivie de la lettre F ou D (*f* ou *d*).

```
3.14      //notation simple, valeur double -par défaut-

3.        // encore un double (valeur 3.00)

.3        // double (valeur 0.3)

3D        // double (risque confusion avec notation hexa)

6.02E23   // double en notation scientifique

123.4E-300D // D redondant

2.718F    // un float

2F        // danger : confusion avec notation hexa

2.F       // 2.00 float
```

(Pour les spécialistes) Java reconnait valeurs infinies, zeros signés et *NaN* :

```
System.out.println(-1d/0d);
 -Infinity
```

Objets: agrégats

De nombreux langages de programmation disposent de moyen de créer de nouveaux types par agrégation d'un ensemble de valeurs individuelles (ayant éventuellement des types différents). Ces regroupements sont appelés types structurés (*struct* du langage C) ou types enregistrements (*Record* de Pascal).

Une définition analogue en Java se ferait au moyen d'une classe:

```
class Individu {
          String nom ;
          int age ;
          float salaire ;
}
```

Une telle approche fournit un moyen d'associer des parties qui constituent le modèle d'un "individu" en les traitant non pas comme des éléments isolés mais comme des composants d'un ensemble plus vaste. Un seul nom est requis pour désigner une variable qui représente un "individu".

```
Individu quidam ;
```

Comme toute déclaration de type non-primitif , cette déclaration n'alloue pas la mémoire nécessaire pour stocker ses composants et les gérer, il faut explicitement demander cet espace ;

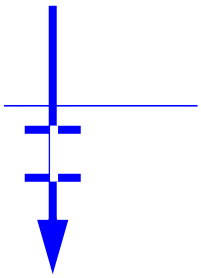
```
quidam = new Individu() ;
```

Après cette déclaration Java permet l'accès aux membres de *l'objet* ainsi créé. Ceci se fait à l'aide de l'opérateur "." :

```
quidam.salaire = 10000.f ;
quidam.age = 24 ;
quidam.nom = ”Dupond” ;
```

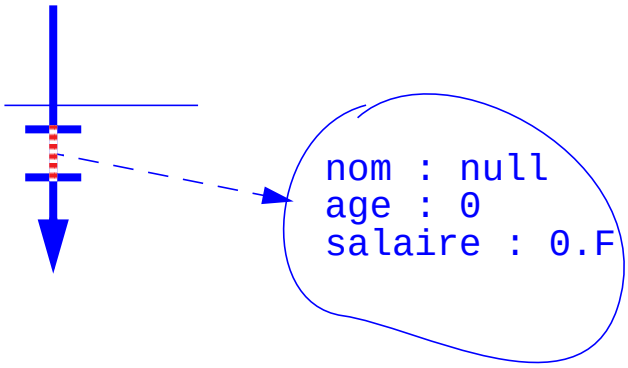
Objets: allocation

```
Individu quidam;
```



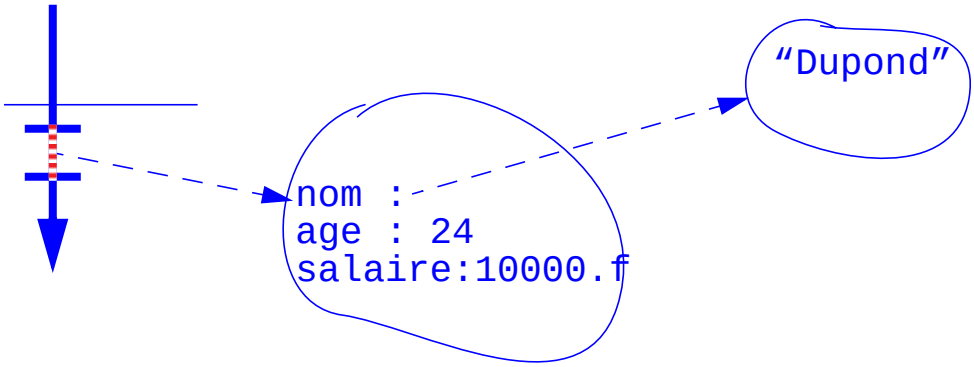
La variable “quidam” existe mais ne référence personne

```
quidam = new Individu();
```



La mémoire a été allouée, la référence désigne un Individu
les membres de l’objet ont des valeurs nulles

```
quidam.nom = “Dupond”;
quidam.age = 24 ;
quidam.salaire = 10000.f ;
```



Objets: introduction à la terminologie

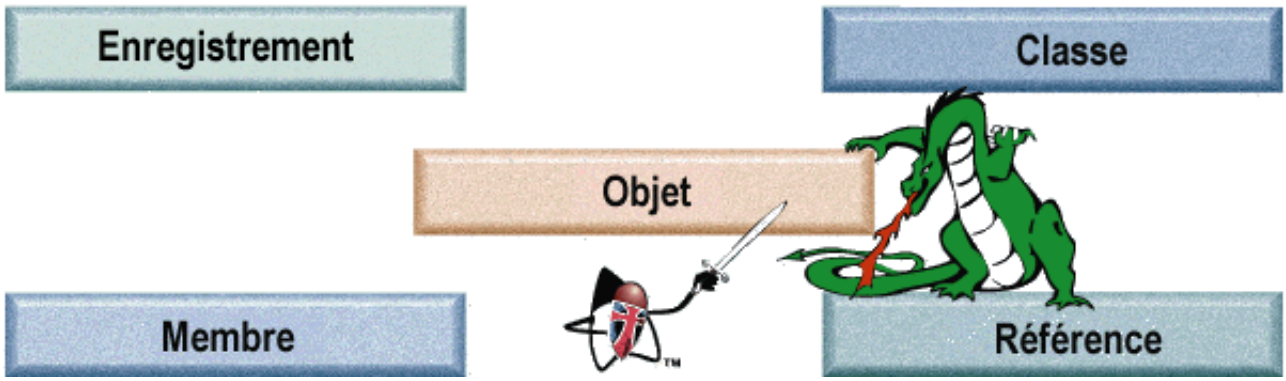
Première introduction à la terminologie “Objet” :

- *Classe* - cadre permettant de définir de nouveaux types dans le langage de programmation Java. La déclaration de classe décrit, entre autres, les composants associés à toute variable de ce nouveau type.
  De nombreux autres aspects seront vus ultérieurement.

- *Objet* - *instance* d’une classe. Une variable de type classe doit référencer une zone mémoire créée par `new` et obéissant aux contraintes imposées par la définition de classe. La classe est un modèle, l’objet est ce que vous fabriquez à partir de cette matrice.

- *Membre* - un membre est l’un des éléments qui constituent un objet. Le terme est également utilisé pour les éléments de la classe de définition.
  Les termes : *variable membre*, *variable d’instance* ou *champ* sont également utilisés. Un objet peut compter d’autres membres que des champs (ceci sera vu ultérieurement).

- *Référence* - en Java une variable définie comme ayant un type d’une classe donnée sert à désigner un objet de ce type. En fait elle référence une instance.



Tableaux : déclaration

Un tableau regroupe un ensemble de données d'un même type. C'est un objet (non-scalaire) avec un définition simplifiée par rapport à celle d'une classe.

```
char[] tChars ; // déclaration d'un tableau de "char"
                // tableau de scalaires

String[] tNoms ; // déclaration d'un tableau de String
                 // tableau d'objets
```

Les déclarations : `char tChars[];` ou `String tNoms[];` sont possibles, on notera toutefois que le type ne précise pas le nombre d'éléments du tableau.

Comme pour les autres objets ces déclarations ne créent pas d'instance, il faut donc allouer ces objets (en précisant leur dimension):

```
tChars = new char[20] ; // le tableau existe
          // avec 20 caractères '\0'

tMots = new String[3] ; // tableau de 3 références String
          // toutes nulles!
```

Il faut ensuite initialiser les éléments du tableau:

```
tChars[0] = 'A'; tChars[1] = 'B' ; ....

tMots[0] = "Ga" ; tNoms[1] = "Bu" ;....
```

On notera que les index commencent à zéro.

Il existe aussi des notations littérales de tableaux qui permettent de simplifier les déclarations et les initialisations.

Tableaux: initialisations

Une première notation (peu utilisée) permet d'allouer et d'initialiser un tableau :

```
tMots = new String[] { "Ga", "Bu", "Zo", "Meu"} ;
```

Ici l'allocation ne précise pas le nombre d'éléments qui est fixé par la description littérale qui suit.

De manière plus courante on utilise une notation raccourcie qui regroupe déclaration, allocation et initialisation :

```
String[] shadock = {
          "Ga" ,
          "Bu" ,
          "Zo" ,
          "Meu"
} ;
```

Cette notation est autorisée pour tout type d'élément :

```
char[] monnaies = {
          '\u20AC',
          '\u20A4' ,
          '$' ,
          '\u20A1'
};

Individu[] candidats = {
          new Individu() ,
          new Individu(),
          new Individu() ,
};

double[] vals = { Math.PI, Math.E } ;// constantes

Color[] palette = {
          Color.blue ,
          Color.red ,
          Color.white,
} ; // ces objets couleurs sont des constantes
```

Tableaux multidimensionnels

Le langage Java a une manière particulière de permettre la création de tableaux multi-dimensionnels. Dans la mesure où on peut déclarer des tableaux de n'importe quel type, on peut déclarer des tableaux de tableaux (ou des tableaux de tableaux de tableaux, etc.)

Voici, par exemple un tableau à deux dimensions :

```
int[][] dim2 = new int[3][] ;
dim2[0] = new int[5] ;
dim2[1] = new int[5] ;....
```

Le premier appel crée un tableau de 3 éléments. Chacun des éléments est une référence nulle qui devra désigner un tableau d'entiers, chacun de ces éléments est ensuite initialisé pour référencer un tel tableau (qui doit être ensuite rempli.)

Une déclaration comme : `new int[][3];` ne serait pas légale, par contre une notation synthétique comme : `new int[3][5];`est possible.

Du fait de ce mode de construction il est possible de réaliser des tableaux de tableaux qui ne soient pas rectangulaires. Ainsi l'initialisation d'un des éléments du tableau précédent pourrait être :

```
dim2[2] = new int[2] ;
```

Et en utilisant une notation qui regroupe déclaration, allocation et initialisation:

```
int[][] dim2 = {
          { 0 , 1, 2, 3, 4 } ,
          { 6 , 7, 8, 9, 5 } ,
          { 10, 20 }
} ;
```

Limites de tableau

Dans le langage Java, les indices des tableaux sont des entiers commençant à zero (pas d'index négatif!).

Le nombre d'éléments dans le tableau est mémorisé dans le tableau lui même à l'aide du champ `length`. Cette valeur est utilisée lors de tout accès au tableau pour détecter les dépassements d'indices. La machine virtuelle Java génère une erreur d'éxécution en cas d'accès illégal.

```
int[] dim1; .... // +allocations + intialisations
String[][] dim2; .. // + allocations + initialisations
System.out.println(dim1.length) ;
System.out.println(dim2.length) ;
System.out.prinln(dim2[0].length) ;...
```

On pourra ainsi utiliser ce champ `length` comme limite de boucle pour parcourir un tableau. Attention: `length` ne peut pas être modifié!
Une fois créé un tableau ne peut pas être redimensionné.
Cependant il est possible d'utiliser la même variable référence pour désigner un nouveau tableau :

```
int monTableau[] = new int[6] ; ....
monTableau = new int[10]; ...
```

Dans ce cas le premier tableau alloué est effectivement perdu (sauf s'il existe une autre référence active qui le désigne).

Une copie efficace de tableau s'effectue au moyen d'une fonction de service définie dans la classe `System` :

```
int[] origine = { 11, 22, 33 } ;
int[] cible = { 0, 1, 2, 3 } ;
System.arraycopy(origine, 0, cible, 1, origine.length);
// copie origine dans cible a partir de l'index 1
// resultat : { 0 , 11, 22, 33 }
```

System.arraycopy() copie des références pas des objets lorsqu'on opère sur des tableaux d'objets. Les objets eux-même ne changent pas.

récapitulation: déclarations de variables

```
// la classe “Individu” est supposée connue par cette classe

public class Affectations {
        public static void main (String[] tArgs) {
                //SCALAIRES PRIMITIFS
                int ix;
                int iy ;
                float fx, fy; //deux déclarations
                ix = -1 ;
                iy = 0xFF ;
                fx = 3.414f ;
                fy = 3e-12f ;
                double dx = 3.1416 ;//double par défaut
                boolean estVrai = true ;
                char euro = ’\u20AC’ ;

                //OBJETS
                String nom ;
                nom = ”roméo” ;
                Individu roméo ; //déclaration
                roméo = new Individu(); // allocation
                roméo.nom = nom ; //initialisation
                System.out.println(roméo.age) ; // 0 !
                Individu juliette = new Individu() ;

                Individu[] quidams; //déclaration
                quidams = new Individu[3] ; //allocation
                quidams[0] = new Individu();// initialisation
                System.out.println(quidams[1]); // null!
                Individu[] acte2scene2 = { roméo, juliette };
        }
};
```

Quelques affectations illégales :

```
iy = 3.1416 ; //le litteral est un double pas un int
fx = 3.1416 ; // idem: double pas float
dx = 3,1416 ; // pas de “,” comme séparateur décimal
estVrai = 1 ; // piège pour programmeurs C/C++
```

Conventions de codage

Il est très vivement conseillé de respecter les conventions de codage suivantes ;

- *Classes* - les noms des classes sont composés d'un ou plusieurs mots accolés, chaque mot commençant obligatoirement par une majuscule:
  ```
  class LivreDeCompte
  class Compte
  ```
  La même convention s'applique aux *interfaces* (un autre type que nous verrons ultérieurement).
  Ne pas utiliser : caractères accentués, '_', '$'

- *Variables, variables membres* - les noms de variables commencent obligatoirement par un caractère minuscule.
  ```
  int nombreClients ;
  ```
  La même convention s'applique aux *méthodes* (aux "fonctions" dans le langage Java).
  Eviter : les caractères '_' et '$' , les noms sur une seule lettre

- *Constantes* - Noms complètement en majuscules avec le caractère "_" (souligné) pour séparer les mots.
  ```
  TAILLE_MAX
  PI
  ```

- *Packages* - Noms complètement en minuscules
  fr.asso.gourousjava.util

Placez une instruction par ligne et utilisez une indentation pour que le code soit plus lisible.

voir `http://java.sun.com/docs/codeconv/html/CodeConvTOC.doc.html`. Pour les commentaires de documentation lire par exemple :"The Design of Distributed Hyperlinked Programming Documentation." `http://www.javasoft.com/doc/api_documentation.html`

Exercices :

*exercice * : une classe simple*

- Définir une classe “R2” contenant deux variables membres de type double : les champs “réel” et “imaginaire”.
- Définir un “main” dans cette classe pour la tester : créer deux instances de R2 de nom d1 et d2 , initialiser ces instances avec des valeurs et les tracer par “System.out.println”.

*exercice ** (suite du précédent): manipulation de références*

- Créer une nouvelle variable de type R2 et de nom “copie”, affecter à “copie” la variable “d2”, tracer d2, copie
- Modifier les champs de “copie”, tracer d2, copie
- Créer un tableau de “R2” contenant d1, d2, copie
- Modifier les champs du deuxième élément du tableau, tracer le contenu des éléments du tableau.

Thème de reflexion : manifestement on a besoin de quelque chose pour "tracer" simplement une instance de "R2".

Syntaxe: expressions et structures de contrôle 3

Points essentiels

- Notion de méthode
- Opérateurs
- Conversions
- Structures de contrôle
- Portée des variables

Notions de méthode: principes, paramètres, résultats

Dans de nombreux langages de programmation la réalisation d'un service est confiée à une fonction. Ces fonctions sont généralement de la forme:

```
resultat = fonction (x, y, z) ;
```

La forme générale d'une déclaration de fonction est :

type de résulat nom de la fonction (liste des paramètres)

```
double annuité ( double montantPrêt,
                 int durée ,
                 double taux ) {
        // code avec un ou plusieurs return valeur double
}
```

En Java un dispositif analogue se trouve dans la définition et l'usage de **méthodes** .

Méthodes de classe :

Exemple: pour calculer un sinus en Java on pourra faire appel à

```
double res = Math.sin(x)
```

Cette notation permet de rappeler qu'il n'existe pas de fonction "libre" en Java: on demande le service de calcul du sinus à une classe (`java.lang.Math` en l'occurence). L'intérêt de cette approche apparaît quand on peut écrire :

```
double autreRes = StrictMath.sin(x) ;//Java 1.3
```

qui propose un autre mode de calcul.
La consultation de la documentation de la classe java.lang.Math décrit la méthode "sin" avec l'en-tête suivant :

```
static double sin (double a) ...
```

Le mot-clef "static" indiquant qu'il s'agit d'une méthode correspondant à un service proposé par une classe.

```
public static void main (String[] tbArgs) {..// procédure
```

Méthodes d'instance

Exemple : soit la définition suivante d'une classe :

```
public class Position {
        int posX ;
        int posY;
        boolean estDansPremierQuadrant() {
                return (posX > 0) && (posY > 0);
        }
}
```

et son utilisation par un objet de type "Position" :

```
Position maPosition ;
... // allocation + initialisations des champs
if(maPosition.estDansPremierQuadrant()) {...}
```

On a ici demandé un service à une instance particulière de la classe position. Cette requête concerne un propriété rendue par la combinaison des valeurs contenues dans l'objet: on parle de méthode d'instance. De manière imagée on qualifie aussi cette "question" adressée à une instance d'"envoi de message".

Passage de paramètres

En Java les paramètres des méthodes sont toujours passés **par valeur**.

```
public void modificationIneffective (int val) {val = 55;}
```

"val" étant une copie d'une valeur dans le programme appelant cette méthode n'aura aucun effet sur la valeur d'origine.

```
public void modificationSansEffet (Position pt) {
        pt = new Position() ;// ne modifiera pas la ref originale
}

public void modificationPossible (Position pt) {
        pt.posX = 10 ; // modifie le contenu de la ref.
}
```

Dans les deux cas le paramètre est une copie d'une référence, la modification de la référence n'aura aucun effet sur le programme appelant, par contre une modification du contenu de la référence est possible.

Opérateurs

A quelques nuances près les opérateurs Java sont proches de ceux C/C++.

- *Opérations arithmétiques* : `+`, `-` (y compris formes unaires), `*` (multiplication), `/` (division), `%` (modulo -y compris sur flottants-)
- *Opérations bit-à-bit sur entiers* : `&` (et) , `|` (ou), `~` (non), `^` (ou exclusif), `>>` (décalage à droite - x >> 4 décale de 4-), `<<` (décalage à gauche).
- *Opérations logiques, tests*: `&&` (et), `||` (ou), `!` (négation), `==` (égalité), `!=` (différence), `?:` (test `return ( x > y) ? x : y ;`)
- *Opérations logiques sur valeurs numériques* : `<`, `>` (inférieur, supérieur), `<=`, `>=` (inférieur ou égal, supérieur ou égal)
- *Incrémentation/décrémentation* : `++` (pré ou post incrémentation), `---` (pré ou post décrémentation)

```
nb = 0 ;
ix = nb++ ; // ix = 0 , nb = 1
iy = ++nb ; // iy = 2 , nb = 2
```

- *Affectation* : `=` (l'affectation rend un resultat : `iy = (iz = it);`) pour les opérations arithmétiques et bit à bit, opérations de "réaffectation" (`x+=2` correspond à `x = x + 2`) :

Java dispose également de quelques opérateurs spécifiques que nous allons étudier.

Opérateurs

La table suivante liste les opérateurs de Java par ordre de préséance . Les indications “G a D” et “D a G” donne le sens de l’associativité (Gauche à Droite et Droite à Gauche)

Séparateur	`.  []  ()  ;  ,`

D a G	`++  --  +` unaire `-` unaire `~  !` `(`transtypage`)`
G a D	`*  /  %`
G a D	`+  -`
G a D	`<<  >>  >>>`
G a D	`<  >  <=  >= instanceof`
G a D	`==  !=`
G a D	`&`
G a D	`^`
G a D	`|`
G a D	`&&`
G a D	`||`
D a G	`?:`
D a G	`=  *=  /=  %=  +=  -=  <<=  >>=  >>>=  &=  ^=  |=`

Il est vivement conseillé de parenthéser les expressions complexes plutôt que de s’appuyer sur l’ordre de préséance donné dans ce tableau.

Opérations logiques “bit-à-bit” sur booléens

En Java le type `boolean` n’est pas assimilable à un type entier.

Pourtant certaines opérations logiques bit à bit s’appliquent entre booléens et retournent un booléen .

- & correspond au ET logique (AND)
- | correspond au OU logique (OR)
- ^ correspond au OU EXCLUSIF logique (XOR)

Toutefois l’évaluation des formes & et | diffère des opérations && et ||

Evaluation des opérations logiques

Les opérateurs && et || assurent des expressions logiques avec “court-circuit” d’évaluation. Considérons cet exemple :

```
UneDate unJour ;
... ; // déroulement du programme
if( (unJour != null) && (unJour.numeroDansMois > 20)) {
          // on fait la paye!
}
```

L’expression logique testée par le “if” est légale et, surtout, protégée contre les incidents. En effet si la première partie de l’expression (`unJour != null`) est fausse, la seconde partie (`unJour.numeroDansMois ..`) n’est pas évaluée -ce qui est une bonne chose étant donné que l’expression déclencherait alors une erreur d’évaluation de la JVM (recherche de contenu à partir d’une référence nulle)-.

De la même manière l’évaluation complète d’une expression || (OU) est abondonnée dès qu’un membre est vrai.

Concaténation

L'opérateur **+** assure la concaténation d'objets de type `String` en produisant un nouvel objet `String` :

```
String titre = ”Dr. ” ;
String nom = ”Alcofibras ” + ”Nasier” ;
String invite = titre + nom ;
```

Le résultat de la dernière ligne donnera

```
Dr. Alcofibras Nasier
```

A partir du moment où un des opérandes de l'opérateur + est un objet de type String l'autre argument est alors converti en String. Toutes les valeurs d'un type primitif scalaire peuvent être converties en String.

```
System.out.println( ”montant TVA = ” + (montant * tauxTVA)) ;
```

Si l'argument est un objet c'est la méthode `toString()` de l'instance qui est appelée (le résultat peut être surprenant et nous étudierons ultérieurement comment utiliser au mieux cette propriété).

Décalage à droite avec >> et >>>

Le langage Java dispose de deux opérateurs de décalage à droite.

L'opérateur `>>` assure un décalage à droite *arithmétique* avec conservation du signe. Le résultat de ce décalage est que le premier opérande est divisé par deux à la puissance indiquée par l'opérande de droite. Ainsi :

```
128 >> 1 donne 128/2¹ = 64
256 >> 4 donne 256/2⁴ = 16
-256 >> 4 donne -256/2⁴ = -16
```

En fait les bits sont décalés à droite et le bit de signe est conservé.

Le décalage à droite logique `>>>` (non signé) travaille purement sur la représentation des bits, des zéros sont automatiquement insérés sur les bits les plus significatifs.

```
1010 .... >> 2 donne 111010 ...
1010 ... >>> 2 donne 001010 ...
```

Les opérandes droits sont évalués modulo 32 pour les opérandes gauches de type `int` et modulo 64 pour ceux de type `long`. Ainsi :
`int x; ... x >>>32 ;` donnera une valeur inchangée pour x (et non 0 comme on pourrait s'y attendre).
L'opérateur >>> n'est effectif que sur des types `int` et `long`.

Conversions, promotions et forçages de type

Le fait d'assigner une valeur d'un type donné à une variable d'un autre type suppose une conversion. Dans certains cas, si les deux types sont compatibles, Java effectuera la conversion automatiquement : par exemple une valeur de type `int` peut toujours être affectée à une variable de type `long`. Dans ce dernier cas on a une *promotion* : le type `long` étant plus "grand" que le type `int` il n'y a pas de risque de perte d'information.

Inversement s'il y a un risque de perte d'information le compilateur exige que vous specifiiez explicitement votre demande de conversion à l'aide d'une opération de transtypage:

```
long patouf = 99L ;
int puce = (int) patouf ;
```

Dans le cas d'expressions complexes il est vivement conseillé de mettre l'ensemble de l'expression à convertir entre parenthèses pour éviter des problèmes dus à la précédence de l'opérateur de transtypage.
Le type `char` demande également des précautions : bien qu'il soit assimilable à un type numérique sa plage de valeurs va de 0 à 2^{16} -1 alors que la plage des valeurs de `short` va de -2^{15} à 2^{15} -1 une conversion explicite est donc toujours nécessaire.

Les conversions de références seront étudiées ultérieurement, elles suivent également des règles de promotion (mais dans ce cas il faudra définir ce que "promotion" veut dire). Par ailleurs un tel transtypage n'a jamais pour effet de transformer effectivement une instance.

Conversions, promotions et forçages de type

Les règles de conversion et de promotion s'appliquent également aux paramètres des méthodes :

```
class Gibi {
          void traiter(int param) { ....
          }
          ...
}

Gibi instance = new Gibi() ;
...
short petit ; .....
instance.traiter(petit) ; // conversion implicite
long gros ;
instance.traiter((int) gros) ; //conversion explicite
```

Il faut également prendre en compte deux autres aspects :

- Les méthodes Java peuvent être *surchargées* c'est à dire que la même méthode peut accepter plusieurs définitions. Ces définitions diffèrent par le type des paramètres qui, de plus, ne sont pas forcément incompatibles entre eux. Ainsi la classe `java.io.PrintStream` (voir `System.out`) connait les méthodes :

  ```
  void println(String x)
  void println(char x)
  void println(int x)
  void println(long x) ....
   // quelle est la méthode réellement appelée
   // par: println(petit) ?
  ```

- Les opérations arithmétiques entières se font automatiquement au minimum avec un `int` et donc les opérandes sont automatiquement promus :

  ```
  short a, b;
  ....// initialisations de a et b
  short res1 = a + b ; // ERREUR COMPILATION
  short res2 = (short) (a+b) ;
            // OK mais risque perte précision
  ```

Structures de contrôle: branchements

Instructions conditionnelles: if-else

La syntaxe de base pour les branchements conditionnels est :

```
if (expression_booléenne)
        instruction ou bloc
 else
        instruction ou bloc
```

Exemple :

```
double salaire = employé.getSalaire();
if (salaire < PLAFOND ) {
        salaire *= tauxAugmentation ;
} else {
        salaire += petitePrime ;
}
        // préférer les blocs dans tous les cas
```

Le `if` de Java diffère de celui du C/C++ car il utilise une expression `booléenne` et non une valeur numérique.

```
if (x) // x est un int
```

est illégal. Par contre on peut écrire :

```
if (x != 0) { //
```

La partie `else` est optionnelle et peut être omise s'il n'y a pas d'action à effectuer lorsque la condition est fausse.

Structures de contrôle: branchements

Instructions de branchement multiple : switch

La syntaxe de base pour l'aiguillage multiple est :

```
switch (expression_choix) {
          case constante1 : //entier non long ou char
                    instructions ;
                    break ; //optionnel
          case constante2 :
                    instructions ;
                    break ; //optionnel
          ....
          default :
                    instructions ;
                    break ; // optionnel
}
```

Le résultat de *expression_choix* doit pouvoir être affecté sans restriction à un entier `int`. Les types `byte`, `short` ou `char` sont possibles (ainsi que `int`!), par contre les types flottants, `long` ou les références d'objets ne sont pas possibles.

```
switch (codecouleur) {
          case 0: // permet d'avoir fond bleu et tracés rouges
                    fenêtre.setBackground(Color.blue) ;
          case 1: // tracés rouges
                    fenêtre.setForeground(Color.red);
                    break;
          case 2: // tracés verts seulement
                    fenêtre.setForeground(Color.green);
                    break;
          default: // quoiqu'il arrive
                    fenêtre.setForeground(Color.black);
}
```

L'étiquette `default` est optionnelle mais permet d'aiguiller sur un choix par défaut. L'omission de `break` permet des définitions complexes.

Structures de contrôle: boucles

Boucle avec test haut : while

```
while(test_booléen)
          instruction ou bloc
```

Bien entendu pour rentrer dans la boucle le test doit être vrai, par ailleurs la valeur testée doit évoluer dans le corps de la boucle.

```
while (maPosition.estDansPremierQuadrant()) {
          maPosition.posX -= 10 ;
          maPosition.posY -= 10 ;
}
```

Boucle avec test bas : do ... while

```
do
          instruction ou bloc
while(test_booléen) ;
```

L'instruction ou le bloc après le do est toujours executé au moins une fois.

```
do {
          maposition.posX += 10 ;
          maposition.posY += 10 ;
} while (! maPosition.estDansPremierQuadrant());
```

Structures de contrôle: boucles

Itération avec boucle for

```
for (initialisations ; test_haut ; operations_itération)
        instruction ou bloc
```

Exemple :

```
for (int incr= 1 ;
        maPosition.estDansPremierQuadrant() ;
                incr *= 2) {
        maPosition.posX += incr ;
        maPosition.posY += incr ;
}
```

* Chacun des 3 arguments du `for` peut être vide (un test vide s'évalue à true).
* Dans l'initialisation on peut déclarer des variables dont la portée est celle de la boucle
* On peut utiliser le séparateur "," dans l'initialisation ou dans les opérations d'itération qui sont exécutées en bas de boucle.

```
for (int ix = 0, iy = tb.length ; ix < tb.length ; ix++, iy--)  {
      tb[ix] = iy ;
}
```

Structures de contrôle: débranchements

Déroutements dans uneboucle

Le déroulement normal d'une boucle peut être dérouté par une des directives :

- **`break`** : sortie prématurée d'une boucle

```
for (int ix = 0; ix <tb.length; ix ++) {
          int lu = System.in.read() ; //
          if ( (-1 == lu) || ('\n' == lu)
                    || ('\r' == lu)) {break;}
          tb[ix] = lu ;
}
/* noter ici la promotion automatique en entier
des opérandes du "==" "'\n' == lu"
*/
```

- **`continue`** : renvoi direct en fin de boucle (test pour les boucles `do..while` et `while`, opérations d'itération pour boucle `for`).

```
for (int ix = 0; ix <tb.length; ix ++) {
          int lu = System.in.read() ; //
          if ( (-1 == lu) || ('\n' == lu)
                    || ('\r' == lu)) {
                    tb[ix] = '|' ;
                    continue;
          }
          tb[ix] = lu ;
}
```

Il existe d'autres possibilités de sortie d'une boucle et, en particulier, le `return` d'une méthode.

Structures de contrôle: débranchements

Déroutements étiquetés

A l'intérieur d'un bloc de programme il est possible d'étiqueter une boucle (par un *label*) pour servir de cible à une directive `break` ou `continue` enfouie dans une des sous-structures -ceci permet d'éviter l'emploi d'une directive *goto* qui n'existe pas en Java (bien que ce soit un mot réservé)-

```
bye : while (true) {
         for (int ix = 0; ix <tb.length; ix ++) {
                  int lu = System.in.read() ; //
                  switch (lu){
                   case -1 :
                            break bye ; //saut en dehors du while
                   case '\n' : case '\r' :
                            tb[ix] = '|';
                            continue ;
                   case '\t' :
                            tb[ix] = ' ';
                            break ;
                   default :
                            tb[ix] = lu
                  }
                  /* nota: toute etiquette dans un switch doit
                   * être "affectable" au type du sélecteur
                   */
                  ...; // du code
         }
         ....; // encore du code !
}
```

Portée des variables, cycle de vie

Nous avons vu des définitions de variables en trois endroits : comme membre d'une classe, comme paramètre d'une méthode ou à l'intérieur d'un bloc d'exécution (dans le cadre d'une définition de méthode par exemple). Dans ces deux derniers cas on dit que la variable est *locale* (ou *automatique*

variables locales

Une variable automatique est allouée sur la pile; elle est créée quand l'exécution entre dans le bloc et sa valeur est abandonée quand on sort du bloc. La portée de la variable (sa "visibilité") est celle du bloc -et des sous-blocs-

- Dans un bloc on ne peut pas définir une variable locale dont le nom rentre en conflit avec une autre variable locale d'un bloc englobant.

```
int ix = 0 ; int iz = 0 ;
....
for (int ix = 0;ix<tb.length;ix++) { // ERREUR sur ix
          float iz ; // ERREUR conflit avec iz
}
```

- Une variable locale **doit** être explicitement initialisée avant d'être utilisée. Cette situation est détectée par le compilateur qui provoque alors une erreur.

```
public void petitCalcul() {
          int ix = (int)(Math.random() * 100);
          int iy,iz;
          if (ix > 50) {
                    iy = 9;
          }
          iz = iy + ix;// ERREUR
                    // iy peut être non-initialisée
          ...
```

Portée des variables, cycle de vie

variables d'instance, variables de classe

Il est possible de définir des variables en dehors des blocs d'exécution (donc au plus "haut niveau" dans la définition de classe. Ces variables peuvent être :

- Des variables membres d'une instance. Ces variables sont créées et intialisées au moment de la création de l'instance par l'appel de new ClasseXyz(). Ces variables existent tant que l'objet est referencé et n'est pas recupéré par le glaneur de mémoire.

```
public class Position {
        int posX ; //initialisation implicite
        int posY = -1; // initialisation explicite
        ....
```

(rappel) utilisation d'une variable d'instance :

```
Position maPosition = new Position() ;
System.out.println(maPosition.posX); // donne 0 !
System.out.println(maPosition.posY); // donne -1
```

- Des variables de classe définies en utilisant le mot-clef `static`. Ces variables sont créées et initialisées au moment du chargement de la classe par le `ClassLoader`. Ces variables existent tant que la classe existe.
  Exemple de variable de classe bien connue dans la classe `System`:

```
public static PrintStream out
```

utilisation :

```
System.out.println(System.out); // bizarre mais essayez!
```

exemple de définition :

```
public class Contexte {
        public static String nomOS =
                  System.getProperty("os.name") ;

        public static String eMailAdministrateur ;
                             // initialisation implicite
```

Portée des variables, cycle de vie

Initialisations implicites

`byte`	`0`
`short`	`0`
`int`	`0`
`long`	`0L`
`float`	`0.0F`
`double`	`0.0D`
`char`	`'\u0000'` (NULL)
`boolean`	`false`
toutes références	`null`

Exercices :

Exercice * (utilisation de la documentation) :

- Rechercher dans la documentation de la classe `java.lang.Integer` une méthode qui tente d'analyser une chaîne pour la transformer en entier. C'est à dire une méthode telle que `f("3")` rende `3`.
  Cette méthode est une méthode de classe qui prend un argument de type String et rend un résultat de type entier primitif.

Exercice * (suite):

- Lorsque l'on utilise la méthode de classe `main`, le paramètre est un tableau de `String`. Ce tableau représente les arguments de lancement du programme.
  Ecrire une classe `Add` qui additionne les arguments qui lui sont passés en paramètres :

```
java monpackage.Add 3 -3 9
9
java monpackage.Add 10 11 12 13
46
```

Exercice * :

- Ecrire une petite calculette qui travaille en notation polonaise inverse (c'est à dire que l'opération à effectuer est notée après les arguments.)
  Pour le moment cette calculette ne fonctionne qu'en travaillant avec deux opérandes flottants.
  NOTE: pour récupérer le premier caractère d'une chaîne voir méthode `charAt(position)` de la classe String.

```
java monpackage.Calc 2.5 3 +
5.5
java monpackage.Calc -6 2 /
-3.0
```

Objets et classes 4

Points essentiels

Introduction à la programmation Objet en Java:

- Classes et instances
- Méthodes d'instance, méthodes de classe
- Encapsulation
- Constructeurs
- Référence `this`

Classes et objets

Comme nous l'avons vu une *classe* peut permettre de définir un modèle de regroupement d'un ensemble de données.

A partir de ce modèle on pourra créer des *objets* qui associeront des valeurs aux membres de la classe.

```
public class UneDate {
          public int jour, mois, année ;
          ...
}
```

La combinaison des *états* à l'intérieur d'une instance est initialisée après la création de l'objet .

```
UneDate ceJour = new UneDate() ;
ceJour.jour = 1;
ceJour.mois = 1;
ceJour.année = 2000 ;
```

Cette combinaison d'états est susceptible d'évoluer pendant la durée de vie de l'objet : c'est le code associé à la classe qui définira la manière dont ces évolutions sont possibles.

Méthodes d'instance

Dans de nombreux langages de programmation lorsqu'on définit un type enregistrement (un *agrégat*), on définit des fonctions qui opèrent sur ce type, mais il n'y a pas d'association particulière entre le code de définition de ces fonctions et celui de définition du type.

Le langage Java permet une association plus étroite, ces "fonctions" sont décrites dans le corps de la classe. De plus les *méthodes d'instance* sont décrites de manière à opérer sur l'instance sur laquelle on les invoque.

```
public class UneDate {
          public int jour, mois, année ;
          ...// autres champs

          public boolean estFérié() {
                    ...//code calcul
          }//End estFérié()

}
```

La manière d'exprimer les instructions du langage est fortement marquée par ce point de vue : on demande un service à une instance on lui "envoie un message".

```
UneDate ceJour = new UneDate() ;
...//initialisations diverses
if (ceJour.estFérié()) {
          ....
}
```

Méthodes d'instance

Les méthodes d'instance peuvent rendre une grande variété de service comme renseigner sur des propriétés de l'instance, modifier l'instance ou même créer un autre objet en fonction de l'état de l'instance courante.

```
public class UneDate {
          public int jour, mois, année ;
          ...// autres champs
          public boolean estFérié() {...}
          ...// autres méthodes

          public UneDate leLendemain() {
                    UneDate res = new UneDate() ;
                    ....// calculs, affectations champs
                    return res ;
          }//End leLendemain()
}
```

Utilisation:

```
UneDate ceJour = new UneDate() ;
...// initialisations
UneDate lendemain = ceJour.leLendemain() ;
```

Méthodes de classe

Selon un point de vue différent de celui des méthodes d'instance une méthode de classe ressemble plus aux fonctions d'un langage de programmation comme C.

```
public class UneDate {
          ....
          public static UneDate lendemainDe(UneDate jour{
                    ....
          }
```

Utilisation:

```
UneDate lendemain = UneDate.lendemainDe(ceJour) ;
```

Méthodes d'instance (suite)

Une méthode d'instance peut servir à modifier de manière cohérente et coordonnée les états internes de l'instance.

```
public class UneDate {
          ....
          public void passeAuLendemain() {
                    // modifie le(s) champ(s) jour
                    // et éventuellement mois, an
          }
}
```

Utilisation:

```
UneDate jourCourant .....
.....
jourCourant.passeAuLendemain() ;
          //"jourCourant" est modifié
          //l'instance ne contient plus les mêmes valeurs
```

Une telle méthode est intéressante dans la mesure où elle concentre la logique d'une modification de l'instance, mais encore faut-il faire en sorte qu'une modification incohérente de l'instance ne soit pas possible.

Encapsulation

Il est possible de restreindre l'accès aux variables membres d'une instance en utilisant un modificateur d'accès comme `private`.

```
public class UneDate {
          private int jour, mois, année ;

          public int getJour() {
                    return jour ;
          }
          public int getMois() { ...}
          ....
          public String toString() {
           ... // date sous forme texte
          }
          public void passeAuLendemain() {
                    // sait modifier les champs privés
          }
}
```

L'utilisation du modificateur `private` dans la déclaration des champs jour, mois, année rend impossible l'accès à ces valeurs par d'autres codes que ceux contenus dans la définition de classe. De cette manière on force un accès indirect qui oblige à passer par des méthodes qui définissent les conditions précises de ces accès et maintiennent la cohérence de l'instance.

Ainsi dans la classe "UneDate" on va pouvoir rendre impossible une modification sans contrôles du champ "jour" (on risquerait d'avoir un 30 février) et on va centraliser la logique (complexe) qui régit l'évolution d'une date.

Le fait de cacher les états internes de l'objet est qualifié d'*encapsulation*. L'encapsulation permet de découpler nettement :

- d'une part la "vision" qu'une classe offre à l'extérieur au travers de son API.
- d'autre part la réalisation concrète des services offerts par la classe.

Encapsulation

Le fait de forcer l'utilisateur d'une classe à passer par une vision abstraite d'une classe permet de découpler l'utilisation et la réalisation et permet même d'anticiper sur des évolutions possibles.

```
public class Client {
        ... //champs divers
        private double solde ;
        public double getSolde () {
                return solde ;
        }
        public void setSolde(double valeur) {
                // operations diverses
                solde = valeur ;
        }
```

Si dans la classe "Client" on avait défini le champ "solde" comme librement accessible on aurait laissé développer des applications qui affectent librement cette valeur. Si, par la suite, les règles de gestion évoluent et il apparaît qu'à toute modification du solde on doit réaliser une opération complémentaire, on est obligé de rechercher dans tous les codes les utilisations de ce champ pour pouvoir rajouter ces nouvelles instructions. Si, par contre, on a centralisé l'accès à "solde" dans une méthode de modification "setSolde", l'ajout de ce nouveau code est centralisé et immédiat.

On notera que la convention de nommage couplée pour les accesseurs / mutateurs -"typeXXX **get**XXX()" et "void **set**XXX(typeXXX)"- est standard en Java.

Initialisations : constructeur

Un autre aspect de l'encapsulation est de prévoir un mécanisme pour remplacer et contrôler les initialisations explicites comme:

```
UneDate ceJour = new UneDate();
ceJour.jour = 1;
ceJour.mois = 1;
ceJour.année = 2000 ;
```

Dans ce cas on centralise la création / initialisation de l'instance dans un ou plusieurs codes de *constructeur*.

```
public class UneDate {
          private int jour, mois, année ;
          .... // accesseurs
          .....
          public UneDate(int leJour,int leMois,int lAn){
                    jour = leJour ;
                    mois = leMois ;
                    année = lAn ;
                    // au fait :
                    // les paramètres sont-ils valides?
          }
}
```

Utilisation:

```
UneDate ceJour = new UneDate(1,1,2000) ;
```

Un constructeur doit avoir exactement le même nom que sa classe.
Attention: ce n'est pas une méthode (pas de résultat par exemple) et, de plus, on ne le considère pas comme un *membre* de la classe (comme le sont les champs et les méthodes)

Initialisations : constructeur

Dans l'exemple de la classe "UneDate" on pourrait aussi définir un constructeur qui initialise automatiquement l'instance avec la date du jour:

```
import java.util.GregorianCalendar ;

public class UneDate {
        private int jour, mois, année;
        // accesseurs
        public int getJour() {return jour;}
        ...
        public UneDate(int leJour,int leMois,int lAn){
                ...
        }

        public UneDate() {
                GregorianCalendar cl = new GregorianCalendar();
                // contructeur met à la date du jour
                jour = cl.get(GregorianCalendar.DAY_OF_MONTH) ;
                .... // autres initialisations
        }
}
```

Exemple:

```
System.out.println("jour=" + new UneDate().getJour()) ;
```

Il est intéressant de remarquer que la classe "UneDate" pourrait ainsi avoir plusieurs constructeurs avec des "signatures" différentes ::

```
UneDate dateDuJour = new UneDate() ;
UneDate dateCritique = new UneDate(1,1,2000) ;
```

On a ainsi une *surcharge* des constructeurs analogue à la surcharge possible pour des méthodes (voir, par ex. les différentes versions de "`println`" dans la classe `java.io.PrintStream`).

Construction de l'instance : compléments

Le constructeur par défaut

Chaque classe est dotée d'un constructeur par défaut: il s'agit du constructeur sans paramètres que l'on invoque par `new Xxxx()`. Toutefois il faut prendre garde au fait qu'à partir du moment où on définit un constructeur avec des paramètres pour une classe qui n'avait pas de constructeur explicite ce constructeur par défaut n'existe plus (le compilateur refusera son emploi). Il est bien sûr alors possible de définir explicitement un constructeur sans paramètres.

Opérations d'initialisations

Il est possible de faire en sorte que certaines variables membres soient initialisées au moment de la création de l'instance :

```
import java.math.BigDecimal ;
import java.util.ArrayList ;

public class Compte {
        private BigDecimal solde = new BigDecimal(.0) ;
        private long heureCréation = System.currentTimeMillis() ;
        private ArrayList opérations = new ArrayList() ;
        ...
}
```

Lors de la création d'une instance la machine virtuelle alloue l'espace nécessaire, affecte les valeurs par défaut aux variables membres et invoque le constructeur. Quand on "rentre" dans le code du constructeur:

- Les initialisations explicites sont exécutées (dans l'ordre spécifié dans le code de la classe).
- Le reste du code du constructeur est exécuté.

Instances : la référence this

Normalement un membre doit être rattaché à une instance à l'aide d'une notation "." (`ceJour.toString()`, `ceCompte.clef`). Toutefois dans une définition de classe il est possible de désigner un membre de la classe courante sans faire appel à cette notation: en particulier la reférence à l'instance courante se note **`this`** et peut être implicite dans de nombreux cas.

L'usage de "this" s'impose dans les cas suivants:

- Pour lever une ambiguité de nommage,

```
public class UneDate {
        private int jour, mois, année ;
        public UneDate(int jour,int mois,int année){
                this.jour = jour ;
                this.mois = mois ;
                this.année = année ;
        ....
```

- Pour passer une référence à l'instance courante dans un appel de méthode,

```
        public void trace() {
                System.out.println(this) ;
        }
```

- Pour invoquer un autre constructeur de la classe dans la défintion d'un constructeur. Dans ce cas cette invocation doit impérativement être la première instruction du code du constructeur.

```
public class Compte {
        ...
        public Compte(Client client,float dépôt,float découvert){
                ....
        }
        public Compte(Client client, float dépôt) {
                this(client,dépôt,DECOUVERT_STANDARD);
        }
        public Compte(Client client) {
                this(client, 0F) ;
        }
```

Récapitulation: architecture d'une déclaration de classe

Bien qu'encore incomplète voici la description du schéma général d'un fichier source java.

```
// déclaration de package
package yyy ;

// déclarations "import"
import zzz ;
import ppp.* ;
....

// déclarations de classe (une seule peut être "public")

... class XXX {
        // l'ordre des déclarations est sans importance
        // (sauf pour l'ordre relatif des initialisations)

        // MEMBRES d'INSTANCE
        champs (avec initialisations éventuelles)
        méthodes
        ...
        // MEMBRES de CLASSE (static)
        champs (avec initialisations éventuelles)
        méthodes
         // public static void main(String[] x): cas particulier
        ...
        // CONSTRUCTEURS
        constructeur(s)
}
```

Exercices

Exercice * :

On créera une classe "EntierVariable" contenant une valeur entière modifiable.
Avec des entiers primitifs de type "int" on peut faire :

```
x = x+y ;
x++;
if (x > y) { ...}
```

Implanter dans `EntierVariable` des méthodes permettant de réaliser les mêmes opérations.

Pour tester cette classe on reprendra un des exercices précédent sur la classe "Add" ou sur la classe “Calc” en utilisant des "EntierVariable".

Exercice * (suite):

On reprendra le même exercice avec une classe "EntierImmuable" dans laquelle la valeur contenue n’est pas modifiable

Exercice * (suite):

Lecture de la documentation : les types standard `String`, `StringBuffer`, `Integer`, `java.util.BigDecimal` ont-ils des instances immuables?

Composition d'objets, héritage 5

Points essentiels

Principes fondamentaux de la programmation Objet en Java (suite):

- Définition d'objets à partir d'autres objets : agrégation, association, délégation.
- Héritage
- Spécialisation de méthodes
- Polymorphisme
- Opérateur `instanceof`
- Invocation du constructeur de la classe mère.

Imbrications d'instances : agrégation, association

Comme nous l'avons vu une *classe* permet le regroupement d'un ensemble de données. Ces données ne sont pas forcément des valeurs primitives.
On peut construire une instance à partir d'autres instances.

Ainsi en supposant que l'on ait défini une classe "Adresse" on pourrait définir:

```
public class Salarie{
        private String nom;
        private String id ; // clef
        private Adresse adresse ;
        private Salarie supérieurHiérarchique ;
        ...
        public String toString() {
                  // permet d'afficher les informations contenues
        }
        // ACCESSEURS
        public String getID() { return id;}
        ...
        // Autres méthodes
        public double coûtHoraire() {
        ....
        }
}
```

Notons toutefois que d'un point de vue conceptuel on a ici deux situations différentes :

- On peut considérer qu'une instance de "Salarié" *contient* effectivement une instance de "Adresse" (relation d'*agrégation*). Les deux instances ont un cycle de vie étroitement lié (même si l'adresse peut changer).

- Le champ "supérieurHiérarchique" est une référence vers une autre instance relativement indépendante (relation d'*association*)

Imbrications d'instances : délégation

Bien entendu on peut définir des méthodes d'instance qui utilisent des méthodes des instances "contenues".
Par exemple, dans l'exemple précédent, la définition de la méthode `toString`() pourrait faire appel à une méthode `toString()` du champ "adresse".

Considérons maintenant l'exemple suivant:

```
public class Executant{
        private Salarie employé ;
        // données spécifiques
        private Tache[] tâchesDuMois;
                  ....
        // méthodes spécifiques
        public double getCoût() {
                  ...
        }
        ...
        //Constructeur
        public Exécutant(Salarie personne) {
                  employé = personne ;
        }
        // méthodes déléguées
        public int getID() {return employé.getID()}
        public double getCoûtHoraire() {
                  ...// on demande à l'instance "employé"
        }
        ....
}
```

Ici plutôt que d'élargir la définition de la classe "Salarié" pour l'utiliser dans une autre perspective (la gestion des tâches) on a préféré définir une nouvelle classe qui utilise les services de la classe précédente. Comme certaines des méthodes de la classe "Salarié" sont pertinentes pour la classe "Exécutant" on *délègue* leur réalisation à une instance associée qui fait partie de la définition de l'Executant. Il aurait été mal avisé de dupliquer les informations gérées par la classe "Salarié" dans la classe "Exécutant".

Relation “est un”

Supposons que pour la gestion du personnel on ait besoin de créer une classe particulière pour les “Managers”.
Si on définissait cette classe en partant de zéro on aurait quelque chose comme :

```
public class Manager{
        private String nom;
        private String id ; // clef
        private Adresse adresse ;
        private Salarie supérieurHiérarchique ;
        private Salarie[] subordonnés ;
        ...
```

Ici on constate que l’on est en train de reprendre les éléments principaux de la classe “Salarié” pour les recopier dans cette nouvelle classe.

Or d’un point de vue conceptuel un “Manager” **est un** Salarié (avec, certes, quelques caractéristiques supplémentaires). Il serait donc pertinent de factoriser le code qui concerne le concept de Salarié pour ne définir que ce qui concerne le cas particulier du “Manager”. L’écriture et la maintenance de ces classes en serait facilitée.

Comme tout langage à objet Java propose un mécanisme d’**héritage** pour gérer cette situation.

Héritage: mot-clef extends

```
public class Salarie{
        private String nom;
        private String id ; // clef
        private Adresse adresse ;
        private Salarie supérieurHiérarchique ;
        ...
}

public class Manager extends Salarie {
        private Salarie[] subordonnés;
        ...
```

La classe “Manager” est définie maintenant pour disposer des champs et méthodes qui sont *héritées* de la définition de la *classe parente*.

La classe “fille” (Manager) disposera en propre :

- de champs d’instance spécifiques
- de méthodes d’instance spécifiques
- de méthodes d’instance qui constituent des redéfinitions de méthodes héritées (*spécialisation*).
- de constructeurs (on n’hérite d’aucun constructeur)

Dans la documentation de l’API du SDK la description d’une classe ne comprend que ces définitions spécifiques. Lorsqu’on recherche un certain membre d’une classe il peut être nécessaire de rechercher sa description parmi les ancêtres de la classe courante.

Spécialisation des méthodes

Reprenons l'exemple de la classe "Salarié":

```
public class Salarie{
        private String nom;
        private String id ; // clef
        private Adresse adresse ;
        private Salarie supérieurHiérarchique ;
        ...
        public String toString() {
                // permet d'afficher les informations contenues
        }
        // Accesseurs
        public String getID() { return id;}
        ...
        // Autres méthodes
        public double coûtHoraire() {
        ....
        }
}
```

La classe "Manager" s'écrirait par exemple:

```
public class Manager extends Salarie {
        // champs spécifiques
        private Salarie[] subordonnés;
        ...
        // méthodes spécifiques
        public Salarie[] getSubordonnés() { return subordonnés;}
        ....
        // méthodes redéfinies
        public double coûtHoraire() {
                ...//calculé différemment pour les managers
        }

        public String toString() {
                return super.toString() +
                        " nombre de subordonnés=" +
                                subordonnés.length +
                        ..... // autre champs
        }
}
```

Spécialisation de méthodes (method overriding)

Dans l'exemple de la classe "Manager" les méthodes `coûtHoraire()` et `toString()` existent dans la classe mère et sont redéfinies dans la classe fille pour être spécialisées.

La nouvelle définition de ces méthodes peut s'appuyer sur un code complètement autonome ou sur un code faisant usage de la méthode définie dans la classe mère. Dans ce dernier cas la méthode originelle est référencée par le mot-clef **`super`** (`super.toString()` dans l'exemple).

Quelques remarques complémentaires:

- Java ne connaît que l'*héritage simple*. Au contraire d'autres langages à objet qui connaissent l'héritage multiple une classe Java ne peut dériver (mot-clef `extends`) que d'une seule classe mère. Cette caractéristique du langage est conforme à un souci de génie logiciel: il vaut mieux que le programmeur rende explicite ses intentions. Que se passerait-il si la classe "Manager" héritait à la fois de "Salarié" et de "Actionnaire", et si les deux classes mère avaient une méthode `toString()`? Plutôt que de définir des règles complexes Java préfère laisser le programmeur décrire précisément le comportement de la classe.

- Un type Java est un véritable "contrat" passé entre ceux qui définissent la classe et ceux qui l'utilisent. Le fait de créer un nouveau type par héritage d'un type existant ne permet pas d'aggraver les termes du "contrat" initial: on ne peut pas spécialiser un méthode en aggravant ses conditions d'utilisations, le compilateur Java veille donc à ce qu'une méthode redéfinie ne soit pas plus "privée" que la méthode originelle .

```
public class Mere {
        public int methode(int arg) { ....

public class Fille extends Mere {
        private int methode (int arg)// ERREUR COMPILATION
         // le type retourné doit aussi être le même
         // par contre: methode(String arg) est possible
```

Ceci nous amène à préciser la notion de "type" d'une référence vers un objet: une référence typée "Salarié" peut-elle désigner un objet dont le type effectif est "Manager"?

Polymorphisme

Conceptuellement un “Manager” **est un** “Salarié”, par ailleurs du fait de l’héritage des classes on doit s’attendre à trouver associés à une instance de “Manager” tous les membres (champs et méthodes) associés à une instance de “Salarié”. De ce fait il est légal d’écrire en Java:

```
Salarie salarié = new Manager() ;
Salarie[] équipe1 = {
                    new Salarie(),
                    new Salarie(),
                    new Manager(),
          ....
          } ;
```

Les objets ont un *type effectif*, mais les variables qui les désignent sont *polymorphes*: c’est à dire qu’elles peuvent être utilisées pour des objets sous différentes formes (ayant éventuellement des types effectifs différents). Il s’ensuit que le typage des variables a des conséquences différenciées au *run-time* et au *compile-time*:

Evaluation dynamique:

Soit le programme :

```
for(int ix = 0; ix < équipe1.length; ix++){
          System.out.println(équipe1[ix].toString());
}
```

Quelle est la méthode toString() qui va être appelée quand le membre du tableau sera effectivement un “Manager”: celle de “Salarié” ou celle de “Manager”?

En fait l’exécuteur Java appelle bien la méthode liée au type effectif de l’objet, c’est à dire que, dynamiquement, au moment de l’exécution il “choisit” la bonne méthode. Cette caractéristique importante du polymorphisme est appelée *liaison dynamique (virtual method invocation).*

Polymorphisme

Compilation et typage des variables objet

Reprenons l'exemple précédent:

```
Salarie[] équipe1 = {
                    new Salarie(),
                    new Salarie(),
                    new Manager(),
            ....
            } ;
```

Peut-on invoquer une méthode spécifique de "Manager" sur une variable de type "Salarié"?

Exemple:

```
for(int ix = 0; ix < équipe1.length; ix++){
        System.out.println(équipe1[ix].getSubordonnés().length);
        // ERREUR DE COMPILATION
}
```

Ici la méthode invoquée est spécifique à la classe "Manager" et ne fait pas partie du "contrat" de type de "Salarié".

Pour pouvoir écrire un tel programme il faut:

- S'assurer du type effectif de l'instance (l'instance courante est-elle bien un "Manager"?). Ceci demande de faire appel à l'opérateur `instanceof`.
- Utiliser une référence pour laquelle l'emploi de la méthode `getSubordonnés()` soit légale. Cette référence peut éventuellement être obtenue par *transtypage*.

Opérateur instanceof

```
Salarie[] équipe1 = { ......... } ;

for(int ix = 0; ix < équipe1.length; ix++){
        if ( équipe1[ix] instanceof Manager) {
                System.out.println(
                        ((Manager)équipe1[ix]).
                                getSubordonnés().length);
        }
}
```

L'opérateur **`instanceof`** permet de tester si une instance satisfait au contrat du type désigné par l'argument en partie droite.

ATTENTION: le test ne donne pas forcément le type effectif de l'objet! Ainsi dans l'exemple précédent l'expression "`instanceof Salarie`" appliquée à un objet de type effectif Manager rendrait un résultat positif: un Manager est un Salarié!

Forçage de type pour des références

Dans l'exemple précédent l'expression:

```
((Manager)équipe1[ix])
```

permet de forcer le type de la référence `équipe1[ix]` . Du fait de l'emploi de l'opérateur de *transtypage* (*cast*) le compilateur doit prendre en compte le fait que la référence doit accepter les obligations découlant du type proposé (et donc doit accepter de référencer des appels de méthodes spécifiques au type "Manager").

Notons que :

- Si au moment de l'exécution le type effectif de l'objet n'est pas compatible avec le type de la référence "forcée" une erreur d'exécution surviendra (`ClassCastException` : voir chapitre sur les exceptions).

- Contrairement à ce qui se passe avec des conversions de types primitifs scalaires il n'y a pas de transformation d'objet. Le fait d'écrire `(Salarie)monManager` ne transformera pas l'instance `monManager` en une instance de type effectif "Salarié".

- Comme dans les conversions normales la promotion du type est implicite. Ainsi puisqu'un "Manager" est un "Salarié":

```
public class Executant {
          ...
          public Executant(Salarie personne) { ...
}
```

permet :

```
Executant exécutant = new Executant(new Manager());
```

sans qu'une conversion soit nécessaire.

Héritage: on hérite des membres pas des constructeurs

Le fait que l'on ait:

```
public class Salarie{
        ....
        public Salarie (String id, String nom, Adresse adresse){
                ....
```

N'autorise pas automatiquement à invoquer un constructeur analogue sur une classe dérivée :

```
Manager monManager = new Manager(sonId, sonNom, sonAdresse);
```

Il faut ici que la classe "Manager" ait explicitement défini le constructeur correspondant! Les constructeurs n'étant pas considérés comme des membres de la classe on n'en hérite pas dans les classes dérivées.

ATTENTION: le cas du constructeur par défaut (sans argument) conduit à une situation particulière:
On sait que le fait de définir un constructeur avec arguments fait disparaître ce constructeur implicite (sauf si on donne une définition explicite du constructeur sans argument).
Or, lors de l'opération de construction d'une instance, les opérations de construction des instances des classes mères doivent être successivement appelées.
Si la classe mère a "perdu" son constructeur par défaut, le compilateur ne nous permettra pas de disposer d'un constructeur par défaut pour la classe courante.
Dans l'exemple courant il devient impossible d'invoquer:

```
new Manager(); // ERREUR COMPILATION
```

car on ne peut faire :

```
new Salarie(); // constructeur inexistant
```

Mot-clef super pour l'invocation d'un constructeur

Dans l'exemple précédent nous avons noté que:

- Il faut définir explicitement un constructeur pour la classe dérivée "Manager"

- Lors de la construction de l'instance de Manager, il faudra "construire" une instance de la classe mère. Comme celle-ci est dépourvu de constructeur implicite (sans argument) on est donc obligé d'appeler un constructeur explicite et ceci doit se faire obligatoirement dans la première instruction du constructeur courant:

```
public class Manager extends Salarie {
        .....
        public Manager(String id,String nom,Adresse adr,int nbS){
                  super(id,nom,adresse);
                  ...// autres instructions si nécessaire
        }
```

Ordre des évaluations au moment de la construction de l'instance (précise les informations données au chapitre précédent):
** 1) allocation de l'instance
** 2) initialisations des super-classes (evt. au travers de super())
** 3) évaluations des initialisations explicites de variables membres (l'ordre est important)
** 4) exécution du reste du code du constructeur.

Spécialisations courantes: toString, equals, clone

Toute classe est implicitement dérivée de la classe java.lang.Object. Chaque fois que l'on écrit :

```
public class MaClasse {
        ...
```

on a en fait:

```
public class MaClasse extends Object {
        ...
```

On peut donc utiliser la classe Object chaque fois que l'on veut gérer des collections c'est à dire des ensembles d'objets pour lesquels le type des instances est générique:

```
Object[] tableauDeNimporteQuoi;//utilisé aussi par les collections
        // du package java.util (par ex. ArrayList)
```

Par ailleurs toute classe hérite des méthodes de la classe Object (voir documentation) et en particulier de:

- `toString()`: donne une chaîne descriptive de l'instance courante. Ainsi la méthode `println(Object obj)` de la classe `java.io.PrintStream` appelle implicitement `obj.toString()` (il en est de même de la concaténation entre chaînes et objets).

- `equals()`: est redéfinie dans de nombreuses classes pour exprimer que le *contenu* de deux instances est le même (par défaut `equals` rend le même résultat que == c'est à dire le fait que les deux instances sont en fait le même objet en mémoire). La redéfinition de `equals` dans une classe suppose souvent une redéfinition de la méthode `hashCode()` dont la description sort du périmètre de ce cours.

- `clone()`: permet de redéfinir la manière dont on crée une nouvelle instance qui est une copie de l'instance courante.

Exercices

*Exercice ** :*

Un "sac postal" contient un tableau de "Lettres". On veut savoir quel est le poids total d'un sac postal, quel est le montant total des affranchissements et on veut également imprimer les caractéristiques des "lettres" contenues dans le "Sac postal".

- Une Lettre se caractérise par son poids (en gramme) et son adresse. L'affranchissement d'une lettre se calcule de la manière suivante :

poids jusqu'à	*20g*	*50g*	*100g*	*250g*	*500g*	*1000g*	*2000g*	*3000g*
tarifs	3,00FF	4,50FF	6,70FF	11,50FF	16,00FF	21,00FF	28,00FF	33,00FF

- Une Lettre Recommandée est une "Lettre" dotée d'un taux de recommandation.

taux	*R1*	*R2*	*R3*
prix suppl.	15,50FF	19,00FF	24,00FF

L'affranchissement se calcule en faisant la somme d'un affranchissement normal et du prix de recommandation.

- Une Lettre recommandée avec accusé de reception est une lettre recommandée dotée d'une adresse retour.
  L'affranchissement est celui d'une lettre recommandée avec une surtaxe uniforme de 8F.

On créera les classes correspondantes. Toutes seront dotées de la méthode getPrix() qui permet de donner l'affranchisssement du pli courant.

Pour montrer le bon fonctionnement du programme on initialisera un sac postal avec un tableau de "Lettres" (contenant également des Lettres recommandées et des Lettres recommandées avec accusé de réception).

Modularité, modificateurs 6

Points essentiels

La modularité en Java:

- Les paquetages
- Contrôles d'accès entre objets
- Modificateur `final`
- Rappels sur le modificateur `static`

packages

Comme nous l'avons vu précédemment les classes doivent être regroupées dans des *packages*.

Un package regroupe un ensemble de classes autour d'un thème commun. Ces classes sont supposées se "connaître" entre elles: c'est à dire qu´étant, en pratique, écrites et maintenues ensembles, elles ont la possibilité de constituer des modules logiques dans lesquels le degré d'encapsulation n'est pas le même que pour les classes "extérieures" au package.

De manière standard le package constitue l'élément de granularité des versions de logiciel Java. Un programmeur utilisant les services d'un package peut avoir besoin de connaître dynamiquement sa version (voir classe `java.lang.Package`)

Les packages sont eux-mêmes organisés en hiérarchies et il est de coutume de préfixer leur nom à partir d'un identifiant internet. Soit, par exemple, deux sociétés connues comme `acme.com` et `gibis.fr` , des packages fournis par ces sociétés pourraient être:

```
com.acme.finance.utils.swap
fr.gibis.utils
```

Rappel: à partir du moment où une classe fait partie d'un package le "vrai" nom de la classe est précédé du nom du package :

```
package fr.gibis.utils ;
public class Chapeau { ....
```

Le "vrai" nom de cette classe est: `fr.gibis.utils.Chapeau` . C'est ce nom qui sera passé au lancement de la machine virtuelle si cette classe est munie d'un point d'entrée :

```
java fr.gibis.utils.Chapeau
```

Packages

Java Platform Packages	
java.applet	Provides the classes necessary to create an applet and t
java.awt	Contains all of the classes for creating user interfaces an
java.awt.color	Provides classes for color spaces.
java.awt.datatransfer	Provides interfaces and classes for transferring data betv
java.awt.dnd	Provides interfaces and classes for supporting drag-and
java.awt.event	Provides interfaces and classes for dealing with different
java.awt.font	Provides classes and interface relating to fonts.
java.awt.geom	Provides the Java 2D classes for defining and performing
java.awt.im	Provides classes and an interface for the input method fr
java.awt.image	Provides classes for creating and modifying images.
java.awt.image.renderable	Provides classes and interfaces for producing rendering-
java.awt.print	Provides classes and interfaces for a general printing AP
java.beans	Contains classes related to Java Beans development.
java.beans.beancontext	Provides classes and interfaces relating to bean context.
java.io	Provides for system input and output through data strear
java.lang	Provides classes that are fundamental to the design of th
java.lang.ref	Provides reference-object classes, which support a limit
java.lang.reflect	Provides classes and interfaces for obtaining reflective in
java.math	Provides classes for performing arbitrary-precision integ
java.net	Provides the classes for implementing networking applic
java.rmi	Provides the RMI package.
java.rmi.activation	Provides support for RMI Object Activation.
java.rmi.dgc	Provides classes and interface for RMI distributed garba
java.rmi.registry	Provides a class and two interfaces for the RMI registry.
java.rmi.server	Provides classes and interfaces for supporting the server
java.security	Provides the classes and interfaces for the security fram
java.security.acl	The classes and interfaces in this package have been su
java.security.cert	Provides classes and interfaces for parsing and managing
java.security.interfaces	Provides interfaces for generating RSA (Rivest, Shamir a DSA (Digital Signature Algorithm) keys as defined in N
java.security.spec	Provides classes and interfaces for key specifications an
java.sql	Provides the JDBC package.
java.text	Provides classes and interfaces for handling text, dates,
java.util	Contains the collections framework, legacy collection cla tokenizer, a random-number generator, and a bit array).

Organisation pratique des répertoires

Lors d'une exécution locale les fichiers .class à exécuter par la J.V.M. doivent se trouver dans une ressource. Cette ressource peut être un répertoire ou une archive java (fichier .jar). Dans les deux cas le fichier .class se trouve dans une hiérarchie correspondant à la hiérarchie des packages.

Ainsi le fichier `Chapeau.class` contenant la classe de nom `fr.gibis.utils.Chapeau` se trouvera dans un répertoire:

```
racine_classes/fr/gibis/utils (UNIX)
racine_classes\fr\gibis\utils (WINDOWS)
```

Pour toute exécution *racine_classes* (ou archive jar) doit se trouver mentionné dans une liste décrite dans le contenu de la variable d'environnement CLASSPATH (ex. UNIX `/home/me/jclasses:.`)

Voir également le mécanisme d'extension dans la documentation `docs/guide/extensions/extensions.html`

Pour le développement :

- Tenir compte du fait que le compilateur a besoin d'accéder aux fichiers *.class* de toutes les classes utilisées dans le code (pour vérifier leur interface). On peut aussi passer des chemins d'accès en paramètre (option -classpath).

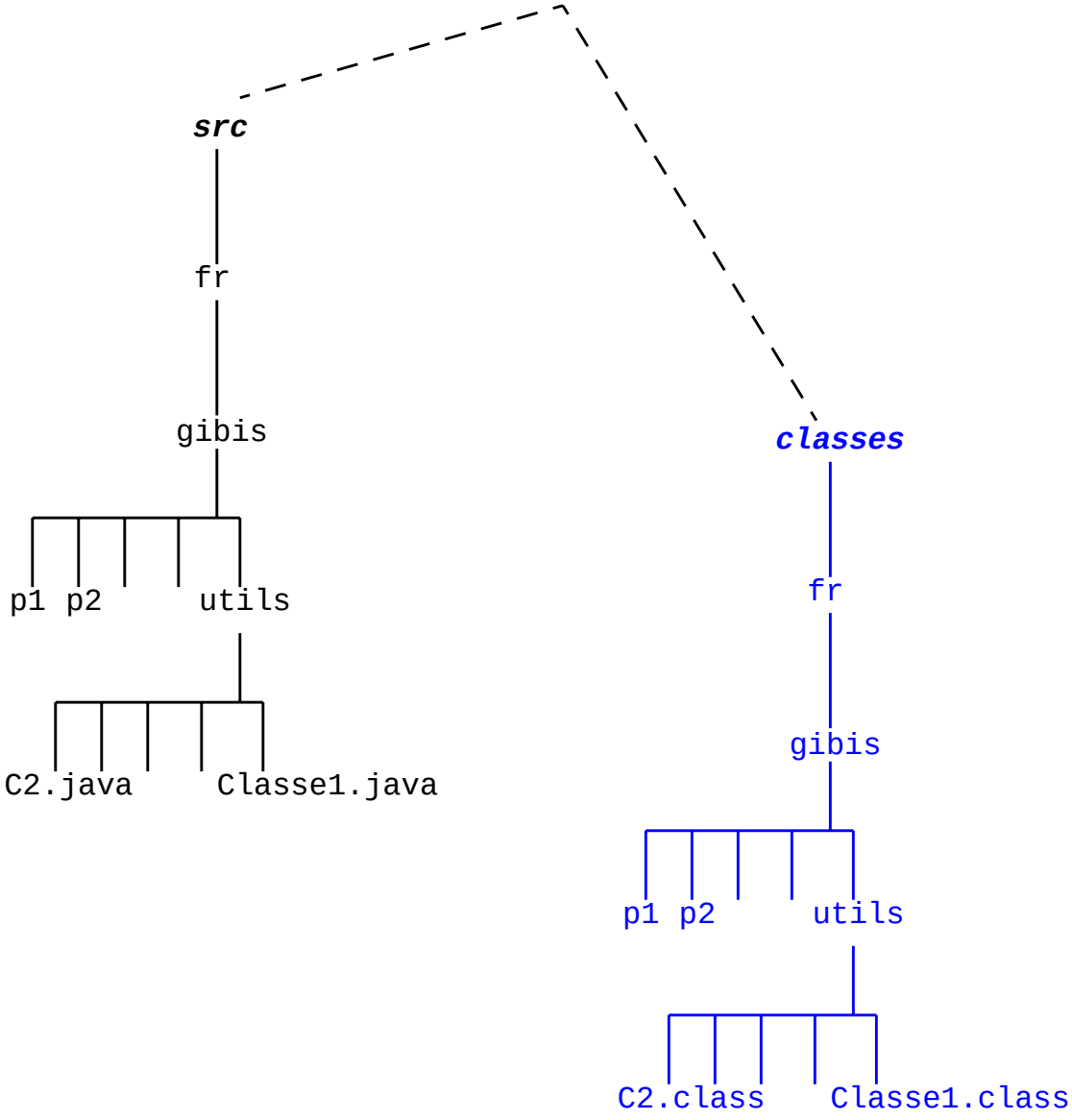
- Générer les .class dans le répertoire racine approprié :

```
java -d racine_classes fichier_java
```

- Il vaut mieux organiser les sources dans une hiérarchie de répertoires correspondant à la hiérarchie des packages et permettre au compilateur de vérifier l'antériorité des sources par rapport au .class correspondant.

```
java -d racine_classes -sourcepath racine_sources fichier_java
```

Organisation pratique des répertoires



javac -classpath *classes* -d *classes* -sourcepath *src* Classe1.java

Déclaration de visibilité (import)

Pour utliser les services d'une classe dans un code on peut donner son "vrai" nom:

```
new fr.gibis.utils.Chapeau(java.awt.Color.blue) ;
```

On peut également demander au compilateur de rechercher lui même le "vrai" nom de la classe dans une liste de packages:

```
import fr.gibis.utils.* ; // pour Chapeau
import java.awt.* ; // pour Color
......
         new Chapeau(Color.blue) ;
```

L'instruction `import` (qui doit se situer avant toute déclaration de classe) permet au compilateur de lever toute ambiguité sur un nom de classe. Noter que :

- L'on peut "importer" spécifiquement une classe sans "importer" toutes les classes de son package.

```
import java.awt.Color ;
```

- La notation * permet de voir toutes les classes du package mais pas les sous-packages

```
import java.awt.*; // on ne voit pas java.awt.event.KeyEvent
```

- Il est inutile d'importer les autres classes situées dans le package courant.

- `import` est une instruction destinée au compilateur elle n'affecte en rien les performances du *run-time*.

Contrôles d'accès

Les membres et constructeurs d'une classe peuvent avoir quatre types de droits d'accès: `public`, `protected`, accès par défaut et `private`. Au premier niveau d'un fichier source les classes peuvent être `public` ou avoir un niveau d'accès par défaut

Résumé des règles d'accès:

Modificateur	**Même Classe**	**Même Package**	**Sous-classe (hors pack.)**	**Ailleurs**
`public`	Oui	Oui	Oui	Oui
`protected`	Oui	Oui	oui *	
par défaut	Oui	Oui		
`private`	Oui			

L'accès `protected` permet en règle générale aux sous-classes situées dans un autre package d'accéder au contenu de la classe courante. Toutefois dans certains cas très particuliers cet accès n'est pas autorisé et est refusé par le compilateur (voir JLS 6.6.2; ces détails sortent du périmètre de ce cours).

Le choix des modificateurs d'accès ne doit pas être traité à la légére et demande un soin tout particulier au niveau de la conception d'une application.

Modificateurs `final`

Classes marquées `final`

Une classe définie comme `final` ne peut plus avoir de classe dérivée (on ne peut pas hériter d'une classe `final`).

Pour des raisons de sécurité de nombreux types de base (comme String, StringBuffer, les types objets encapsulant des types primitifs -Integer, Double, etc.-) sont déclarés final.

Méthodes marquées `final`

Une méthode définie comme `final` ne peut plus être redéfinie dans une sous-classe.

Exemple: dans la classe `java.awt.Window` la méthode `final String getWarningString()` permet de savoir que cette fenêtre ne vient pas d'une application locale mais d'une application supposée "peu fiable" comme une Applet. Pour d'évidentes raisons de sécurité on doit empécher de redéfinir cette méthode.

Remarque: le fait de marquer une méthode comme `final` permet au compilateur de faire de l'optimisation, puisqu'il n'y a pas de redéfintion possible l'évaluation dynamique en fonction du type effectif de l'objet n'a plus cours.

Modificateurs `final`

Variables marquées `final`

- Variables membres initialisées: on ne peut modifier la valeur d'une variable membre initialisée et marquée "final"
  `public static final int TOTAL_HORAIRE_MAX = 35 ;`
  On a de fait une constante (Attention: si cette constante est une instance on peut éventuellement modifier un de ses propres membres)

- Variables membres non initialisées (*blank final*) : une variable membre marquée "final" et non initialisée est obligatoirement initialisée au moment de la construction de l'instance. La valeur ne peut plus être ensuite modifiée:

```
public class Compte {
        public final String clef ;
        ...
        public Compte(String id, Client client,...) {
                ...
                this.clef = id ;
        }
        ...
}
```

Modificateur static (rappel)

variables partagées

Les variables liées à une classe (et non à une instance) sont marquées `static`.

```
public class Contexte {
        public static boolean surDoze =
         System.getProperty("os.name").startsWith("Win");
        ...
}
```

Ici la variable est initialisée au moment du chargement de la classe par un `ClassLoader`. Pour consulter sa valeur :
`if (Contexte.surDoze) {..`

```
// autre exemple
public class Client {
        private static int compteur ; // variable de classe
        public final int id = compteur++; // var d'instance
        ...
}
```

Ici lors de la création d'une instance on utilise la variable de classe et on l'incrémente.

méthodes de classe

On a parfois besoin de créer du code qui n'est pas lié à l'état d'une instance particulière. Dans ce cas on définit une méthode de classe (marquée `static`). Exemples de méthodes de classe:

```
Math.sin(x) ...
Integer.parseInt(s) ...
System.getProperty(s)...
```

```
// une méthode de classe
public class Contexte {
        ...
        public static int getValFromDB(String dbname, String nm){
                ....// va chercher une valeur entière
                ....// sauvegardée dans une ressource
        }
}
```

- On peut invoquer une méthode statique en la qualifiant par le nom de la classe ou par un nom d'instance de cette classe. Par contre à l'intérieur d'une méthode statique il n'y a pas d'instance courante (pas de `this`) et on ne peut accéder implicitement à des méthodes ou variables d'instance.

- `main` est `static` parcequ'il faut pouvoir l'invoquer avant que toute instanciation de classe se soit produite. Bien entendu c'est dans ce `main` qu'il faudra créer des instances et c'est sur ces instances que l'on pourra désigner des membres.

- Dans un héritage on ne peut pas redéfinir une méthode de classe avec une méthode d'instance.

bloc d'initialisation de classe

Un bloc de code marqué `static` est évalué au moment du chargement de la classe par un `ClassLoader`. Bien entendu, en ce qui concerne la classe courante, ce code ne peut faire appel qu'à des membres statiques .

```
public class Client {
private static int compteur ;
static {
         if (Boolean.getBoolean("monAppli.confdb")) {
                  compteur = Contexte.getValFromDB(
                           "client","compteur");
         } else {
                  compteur = Integer.parseInt(
                           System.getProperty(
                            "monAppli.client.compteur")) ;
         }
}
...
```

Exercices

Les exercices de ce chapître ne concernent pas de l'écriture de code mais de la lecture de code: pourquoi ne pas apprendre Java en lisant le code source de Java lui-même? Dans l'installation de votre SDK vous devez avoir un sous-répertoire "*src*" qui contient les fichiers sources. Quelques suggestions d'exercices :

- champs `public` [*] : dans le package `java.awt` voir le code de la classe `Dimension`. Voir également la classe `Component` et ses méthodes `Dimension getSize()`, etc. Pourquoi les champs `width` et `height` de `Dimension` sont-ils publics?

- champs protected[**]: pourquoi le constructeur de `java.lang.ClassLoader` est-il protected?
  Pourquoi les méthodes `defineClass`(...) de cette classe sont-elles "final protected"?

Exercices à réaliser ultérieurement:

- champs visibles dans le package [**] : voir `java.awt.Component`
  Quelle est la nature des champs accessibles dans le package?
  A l'inverse pourquoi le champ `name` est-il `private`?

- utilisation de `protected` [** et ***] : pourquoi les membres ou constructeurs suivants sont-ils marqués "protected" :

  - méthode `clone()` de `java.lang.Object`.

  - méthode `setChanged()` de `java.util.Observable`.

  - champ `modCount` de `java.util.AbstractList` .

Les exceptions

7

Points essentiels

Le traitement des erreurs en Java:

- Le mécanisme des exceptions
- Définir une nouvelle exception
- Déclencher une exception
- Capturer et traiter une exception
- Propager ou traiter?

Le traitement des erreurs

Dans un certain nombre de langage le traitement des erreurs n'est pas pleinement intégré dans les mécanismes du langage, ainsi :

- Si les fonctions sont uniquement de la forme: `resultat = f(arguments)`, on a donc l'obligation de tester les résultats qui peuvent être d'une nature anormale (la fonction retourne donc deux types de résultats: les résultats normaux et les erreurs).

```
/* exemples en langage C */
        if (( fd = open(filename, ...)) < 0 ) {
                  /* tester une variable d'environnement
                   * qui indique l'erreur */
        }
        ....
        if( (fp = fopen(filename,...)) == NULL) {
                  /* idem tests environnement*/
        }
        ....
```

- Lorsque l'exécution se trouve dans un état anormal, le programme peut s'interrompre brutalement en laissant l'environnement dans un état incohérent (et en laissant l'utilisateur perplexe...)

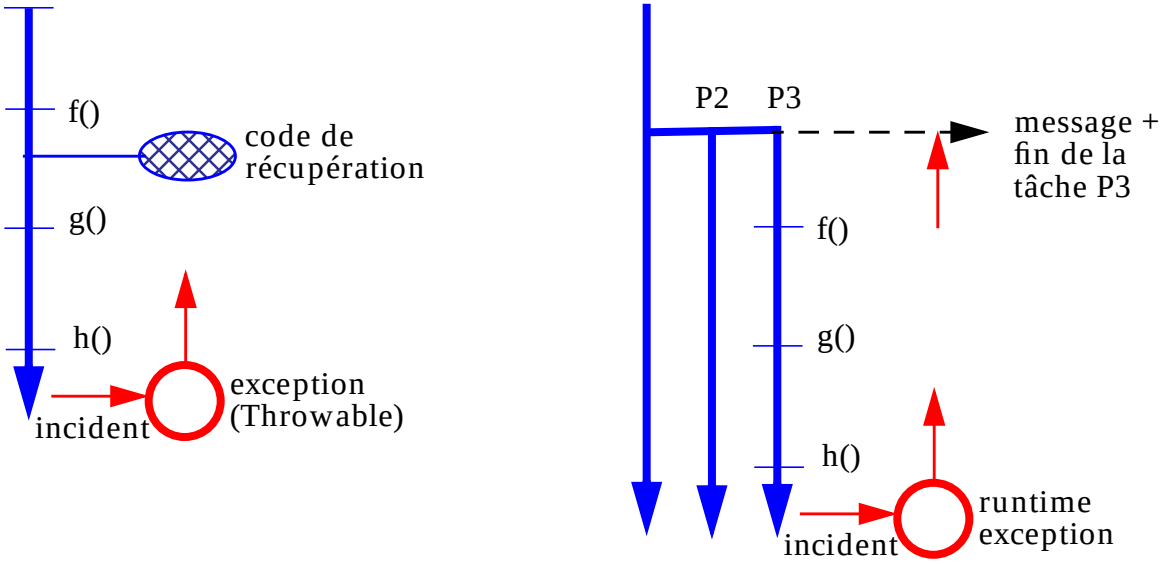
Dans des sytèmes modernes il est d'usage de distinguer la "sortie" normale des résultats d'une sortie distincte pour les erreurs. Ainsi, par exemple, un programme Java pourra afficher des résultats sur `System.out` et afficher les erreurs sur `System.err` .

Le même principe s'applique aux méthodes Java: il y a deux manières de "sortir" de l'éxécution, l'une est la voie normale (avec un éventuel retour de résultat) et l'autre est la voie exceptionnelle qui est gérée par un mécanisme spécifique: le mécanisme des exceptions.

Ce mécanisme permet de définir par programme des conditions d'erreurs. Il est aussi pleinement intégré à la machine virtuelle Java pour permettre de récupérer des erreurs d'exécution comme des erreurs d'index dans un tableau, des divisions par zéro, des déréférencements de variables de valeur `null`, etc.

Le mécanisme des exceptions Java

Lorsqu'une condition d'erreur est détectée (soit par le programme, soit par le système d'exécution) on génère un objet particulier (une exception) qui, éventuellement, contiendra toutes les informations nécessaires au diagnostic. Cet objet est ensuite "jeté" dans la pile (il dérive d'ailleurs de la classe `Throwable`), c'est à dire que l'exécution sort de son déroulement normal et remonte la pile jusqu'à ce qu'elle rencontre un code particulier chargé de récupérer l'objet diagnostic.



Les classes Java sont susceptibles de génerer toute sortes d'exceptions pour rendre compte des incidents spécifiques à telle ou telle action. Les programmeurs peuvent définir leurs propres exceptions, les déclencher et les récupérer.

Dans la plupart des cas quand un programmeur écrit un code qui déclenche une exception, le compilateur s'assure qu'il y a bien un code pour récupérer l'incident. Dans le cas contraire on remonte jusqu'au sommet de la pile (liée au processus courant) et l'exception est traitée par un mécanisme par défaut (qui écrit un diagnostic sur la sortie d'erreur standard).

Exemple de récupération d'exception

Soit le programme:

```
public class Addition{
        public static void main (String[] args) {
                int somme = 0 ;
                for (int ix = 0 ; ix < args.length; ix++) {
                        somme += Integer.parseInt(args[ix]) ;
                }
                System.out.println("somme=" +somme);
        }// main
}
```

et son exécution avec des arguments erronnés:

```
java Addition 1 2 douze 1OO
Exception in thread "main" java.lang.NumberFormatException: douze
        at java.lang.Integer.parseInt(Integer.java:409)
        at java.lang.Integer.parseInt(Integer.java:458)
        at Addition.main(Addition.java:7)
```

Ici la machine virtuelle a été arrétée au premier argument erroné. Si l'on voulait récupérer les erreurs pour les traiter d'une autre manière il faudrait mettre en place un mécanisme adapté.

En standard sur le SDK java 2 les exécutions se font avec un compilateur à la volée. Pour obtenir des informations sur les numéros de ligne des méthodes dans la pile exécuter java avec une variable d'environnement ; `JAVA_COMPILER=NONE`

Exemple de récupération d'exception

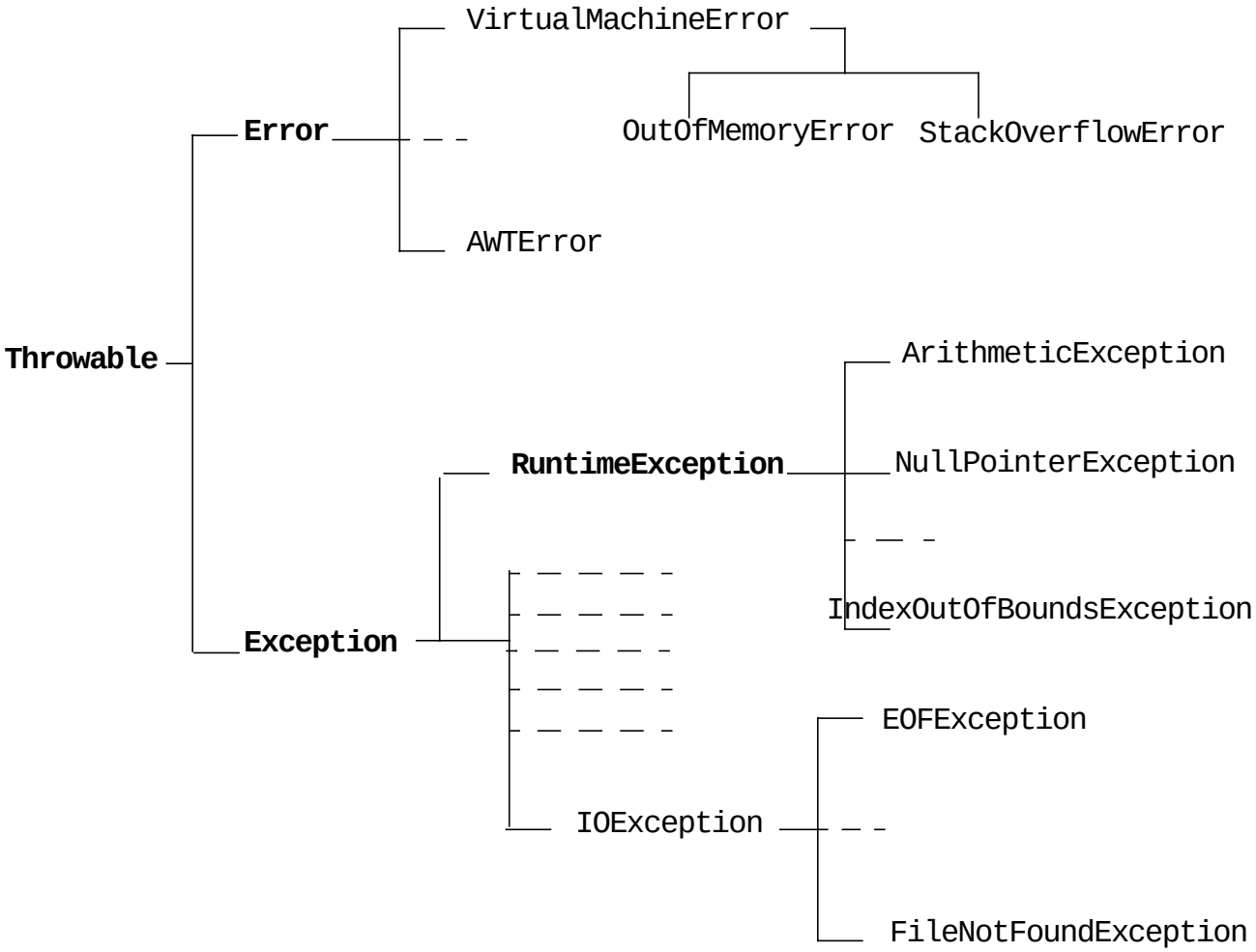
La recupération d'exception passe par la mise en place de blocs `try-catch`;

```
public class Addition{
        public static void main (String[] args) {
                int somme = 0 ;
                boolean ok = true ;
                for (int ix = 0 ; ix < args.length; ix++) {
                        try {
                                somme += Integer.parseInt(
                                        args[ix]) ;
                        } catch (NumberFormatException exc){
                                ok = false ;
                                System.err.println(exc);
                        }
                }
                if (ok) {
                        System.out.println("somme=" +somme);
                }
        }// main
}
```

Le résultat de l'exécution (ici 100 est écrit avec deux lettres 'o' majuscules):

```
java Addition 1 2 douze 1OO
java.lang.NumberFormatException: douze
java.lang.NumberFormatException: 1OO
```

Hiérarchie, exceptions courantes



La classe `Throwable` ne doit pas être utilisée:

- `Error` indique un problème sévère de JVM dont la récupération peut s'avérer fort difficile voire impossible.
- `RuntimeException` indique un problème d'exécution souvent imprévisible (et souvent détecté par l'exécuteur lui-même). Sauf à faire un code bavard et sur-protégé l'implantation du code de récupération de ces exceptions doit être soigneusement justifiée.
- Les autres exceptions doivent impérativement être récupérées et le compilateur lui-même va le contrôler.

Hiérarchie, exemples d'exceptions standard

- **`RuntimeException`** :
  - `ArithmeticException` : ex. division entière par zéro.
  - `NullPointerException` : tentative de déréférencement sur une variable objet dont la valeur est `null`.
  - `ClassCastException` : transtypage impossible au *run-time.*
  - `ArrayIndexOutOfBoundsException` : dans un tableau tentative d'accès à un index qui n'existe pas.
  - `SecurityException` : lors d'une exécution sous le contrôle d'un `SecurityManager` (par ex. dans une Applet), l'`AccessControler` est susceptible de refuser l'accès à une ressource locale.
- **exceptions contrôlées**:
  - `ClassNotFoundException`: au *run-time* la demande de chargement dynamique d'une classe échoue.
  - `InterruptedException`: interruption d'un Thread
  - `java.io.FileNotFoundException`: fichier non trouvé
  - `java.net.MalformedURLException`: erreur dans la syntaxe de description d'une U.R.L.
  - `java.beans.PropertyVetoException`: la modification d'une propriété est refusée par une instance chargée de donner son avis.
  - ...

Définir une nouvelle exception

On peut créer une nouvelle exception par dérivation de la classe Exception.

```
public class ExceptionAffectationHoraire extends Exception {
        public Executant exécutant;
        public Tache tâche;

        public ExceptionAffectationHoraire(String motif,
                            Executant exec,
                            Tache tâche) {
                super(motif);
                this.exécutant = exec;
                this.tâche = tâche ;
        }

        // utiliser getMessage() de la classse Exception
        // pour récupérer le motif de l'erreur

        // par contre : s'agissant d'un objet compte-rendu
        // la mise en place d'accesseurs pour les champs
        // est généralement superfétatoire
}
```

Un autre exemple:

```
public class ExceptionSauvegarde extends java.io.IOException {
        public Exception detail;

        public ExceptionSauvegarde(String s, Exception ex) {
                super(s);
                detail = ex;
        }

        public String getMessage() {
                if (detail == null){return super.getMessage();}
                else {
                 return super.getMessage() + "; " + detail;
                }
        }
}
```

Déclencher une exception

Pour écrire du code qui génère une exception il faut :

- Une instance d'un objet dérivé de Exception
- Utiliser la directive **`throw`** pour provoquer la remontée de cette instance dans la pile des appels.

Exemple :

```
if (totalHoraire > TOTAL_HORAIRE_MAX) {
        throw new ExceptionAffectationHoraire(
                  "dépassement maximum légal " ,
                  exécutantCourant, tâcheCourante) ;
}
```

Autre Exemple:

```
import java.io.* ;
....
try {
        ....
} catch (IOException exc) {
        .... // code éventuel
        throw new ExceptionSauvegarde(
                  opérationCourante + " non réalisée" , exc) ;
}
```

Blocs try-catch

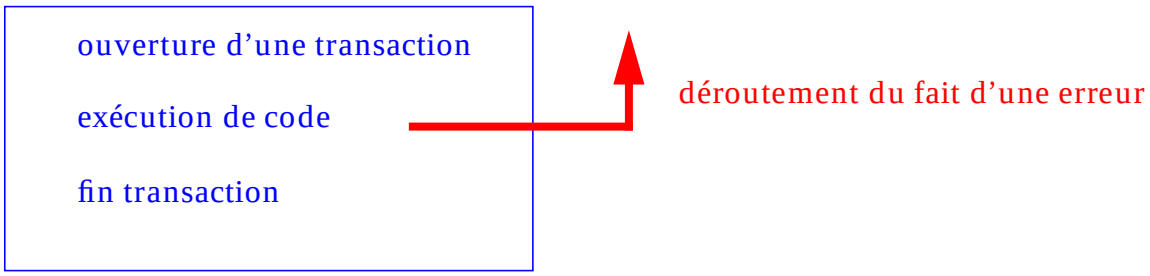
Pour mettre en place du code protégé :

- Mettre le code susceptible de propager une ou plusieurs exceptions dans un bloc **`try`**
- Adjoindre au bloc try un ou plusieurs blocs **`catch`**. Chaque bloc catch gérant une exception particulière ou un ensemble d'exceptions représentées par un ancêtre commun (l'ordre des blocs est important en cas de recouvrement des types d'exceptions).

```
try {
        ....
        // code susceptible de propager des exceptions
        ...
} catch (ExceptionSauvegarde exc) {
        ...
        // code spécifique à cette exception
        ...
} catch (IOException exc) {
        ...
        // code pour les autres erreurs entrées/sorties
        ...
} catch (Exception exc) {
        ...
        // code pour les autres cas
        ...
}
```

Bloc finally

Du fait du déroulement particulier des exceptions dans un bloc try certaines instructions du bloc peuvent ne pas être exécutées (au détriment du bon fonctionnement du programme) :



Dans ce cas la "fin de transaction" ne sera jamais exécutée et l'environnement peut se trouver dans un état incohérent.

Un bloc **`finally`** accolé à un bloc `try` englobe du code qui sera toujours exécuté quelque soit la façon dont on sort du bloc try : exception, return, déroutement étiqueté, ... (sauf si on invoque `System.exit(...)`).

```
try {
        ... //ouverture transaction
        ...// exécution de code
        ...
} finally {
        /* fermeture transaction (code compatible avec un
         * échec de l'ouverture !)
         */

}
```

Le plus souvent on aura :

```
try {
        ...
} catch ( ...) {
        ...
} finally {
        .. // code exécuté dans tous les cas
        ...// même si le catch refait un "throw"
}
```

Règle : déclarer ou traiter

Pour encourager l'écriture de code robuste le langage Java exige que pour tout emploi de code susceptible de lever une Exception qui ne soit pas une erreur d'exécution liée au *run-time* (classes `Error` et classes `RuntimeException`), il soit garanti qu'il y ait du code de traitement récupérant ces exceptions.

Si un programmeur utilise un code susceptible de déclencher une exception le compilateur vérifie que :

- Dans la méthode courante il y a un bloc try-catch qui englobe le code critique et qui sait agir sur les exceptions à traiter:

```
public void methodeF() {
          ....
          try {
                    ... // code susceptible de provoquer
                    // Une exception de type UneException
          } catch (UneException exc) {
                    ....
          }
}
```

- Si un tel bloc n'existe pas alors la méthode courante déclare propager ces exceptions :

```
public void methodeF() throws UneException {
          // code susceptible de provoquer une Exception
}
```

De cette manière tout code utlisant la méthode saura qu'il doit à son tour traiter ou propager les exceptions impliquées.

A la suite du mot-clef `throws` vient une liste de toutes les exceptions susceptibles d'être propagées par la méthode :

```
public Taches[] tachesDuMois()
          throws ExceptionAffectationHoraire,IOException{
```

Cette liste fait partie des caractéristiques significatives de la méthode et apparaît dans sa documentation.

Récapitulation: modèle des méthodes

Il nous faut maintenant modifier et préciser notre précédente définition des caractéristiques significatives d'une méthode :

modif.+accès type_résultat **nom_méthode** (liste_paramètres) liste_exceptions

```
public double annuité ( double montantPrêt,
                        int durée ,
                        double taux )
                                  throws Exception1, Exception2, {
          // code avec un ou plusieurs return valeur double
}
```

Une méthode comporte donc:

- Un nom : plusieurs méthodes peuvent avoir le même nom et des paramètres différents (méthodes *surchargées*)
- Des paramètres éventuels: caractérisent la *signature* de la méthode qui permet de distinguer les méthodes surchargées entre elles.
- Un type de résultat (ou void) : lorsqu'une méthode d'instance est redéfinie dans une sous-classe le type du résultat doit être le même.
- Un modificateur d'accès . Lors d'une redéfinition de méthode d'instance dans une sous-classe le contrat de type ne peut pas être aggravé: la nouvelle méthode ne peut pas être plus "privée" que la méthode initiale. D'autres modificateurs complémentaires (`final`, `native`, `abstract`, etc.) sont possibles.
- Une liste éventuelle d'exceptions propagées : lors d'une redéfinition de méthode d'instance dans une sous-classe le contrat de type ne peut pas être aggravé: la nouvelle méthode ne peut pas propager "plus" d'exceptions controlées que la méthode initiale. Tout au plus peut-elle déclarer propager des sous-classes de ces exceptions controlées.

Exercices

Exercice * : “listeur" de répertoires [**]

Ce programme liste les fichiers contenus dans les répertoires dont les noms sont passés dans les paramètres d’appel.

On vérifiera que le nom de répertoire passé en paramètre correspond à un fichier existant (définir ou utiliser une exception appropriée) et est un répertoire (définir ou utiliser une exception appropriée).

Voir classe "java.io.File" et ses méthodes

```
exists()
isDirectory()
list() ;
```

Pour réaliser ce programme pensez à définir une classe que vous pourrez réutiliser à d’autres occasions.

Perfectionner ensuite le programme en faisant en sorte que tout répertoire affiche récursivement son contenu.

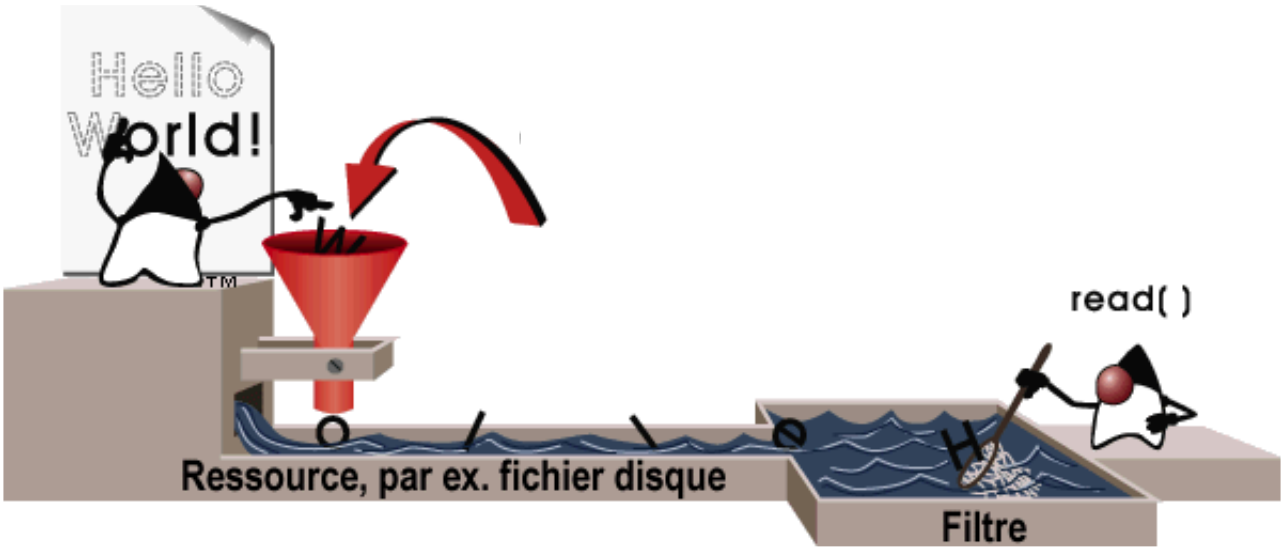
Introduction aux entrées/sorties 8

Points essentiels

Java dispose d'un modèle d'entrée/sortie portable entre plate-formes:

- Les *flots* (*streams*) sont le concept de base des E/S
- Certains types de flots sont associés à des *ressources* (fichier sur disque, buffer en mémoire, ...)
- Certains autres sont des *filtres* c'est à dire des flots qui transforment la manière dont opère l'entrée/sortie.
- Les E/S s'opèrent en combinant divers filtres sur une ressource.

Entrées/Sorties portables, notion de flot (stream)



Notion de flot (stream)

Selon les systèmes d'exploitation les notions fondamentales d'entrées/sorties varient considérablement. Java s'est trouvé dans l'obligation de proposer un concept unificateur qui soit adaptable à des situations très différentes.

Au travers du concept de *flot* Java présente les échanges d'E/S comme une succession continue de données. Un producteur qui écrit *dans* un flot écrit des données les unes à la suite des autres. Un consommateur qui lit ces données *depuis* un flot, les lit successivement les unes à la suite des autres.

Dans cette idée unificatrice on peut englober des modalités de réalisation très différentes:

- des flots qui diffèrent par l'origine effective des données. On pourra, par exemple, "lire" des données *depuis* des fichiers, une ligne de communication, un buffer en mémoire,...
  C'est la nature de la *ressource* qui différencie ces flots.

- des flots qui diffèrent par la manière dont ils opèrent ces lectures/écritures. On pourra, par exemple, avoir des lectures qui opèrent avec bufferisation, compression, cryptage, traduction de types de données, etc..
  Ces flots sont appelés des *filtres (filter)* et peuvent être combinés entre eux pour lire/écrire de/vers des *ressources (node streams)*.

Flots d'octets (InputStream, OutputStream)

La définition fondamentale des flots est de décrire un dispositif pour lire / écrire des octets (`byte`).

Ces classes sont donc munies de méthodes de bas niveau comme :

```
java.io.InputStream
         int read() throws IOException
          // lit en fait un byte dans un int
          // + d'autres read(...) de bas niveau

java.io.OutputStream
         void write(int octet) throws IOException
          // écriture d'un byte passé comme int
          // + autres write(...) de bas niveau
```

Les deux classes sont munies d'une méthode de fermeture

```
         void close() throws IOException
```

Flots de caractères (Reader, Writer)

Dans la mesure où Java utilise de manière interne des caractères UNICODE codés sur 16 bits. on a été amené à définir d'autres types fondamentaux de flots adaptés aux caractères. Ce sont les **`Reader`**s et les **`Writer`**s.

```
java.io.Reader
         int read() throws IOException
          // lit en fait un char dans un int
          // + d'autres read(...) de bas niveau

java.io.Writer
         void write(int carac) throws IOException
          // écriture d'un caractère passé comme int
          // + autres write(...) de bas niveau
          // dont write(string s)
```

Bien entendu ces classes disposent également de `close()`.

Typologie par “ressources”

En prenant l’exemple des seuls flots d’entrées comparons les `InputStream`s et les `Reader`s

catégorie	**`InputStream`**	**`Reader`**
lecture dans tableau mémoire	`ByteArrayInputStream`	`CharArrayReader` `StringReader`
mécanisme producteur/consommateur entre Threads	PipedInputStream	PipedReader
fichiers	FileInputStream	FileReader **InputStreamReader**
flots fabriqués par d’autres classes	java.lang.Process getInputStream() java.net.Socket getInputStream() java.net.URL openStream() ...	

Une classe permet de passer du domaine des `InputStream`s à celui des `Reader`s : **`InputStreamReader`**

Conversions octets-caractères

On utilise les caractères UNICODE à “l’intérieur” du monde Java, mais de nombreuses ressources texte n’utilisent pas ce mode de codage. On a ainsi des fichiers “texte” qui peuvent avoir été fabriqués par des systèmes divers qui emploient des jeux de caractères différents (le plus souvent avec des caractères codés sur 8 bits).

Lorsqu’on échange du texte entre le “monde” Java et le monde extérieur il est essentiel de savoir quel est le mode de codage de texte adapté à la plateforme cible.

Deux classes `InputStreamReader` et `OutputStreamWriter` permettent de passer d’un flot d’octets à un flot de caractères on opérant les conversions appropriées.

```
try {
      FileInputStream fis = new FileInputStream("fichier.txt") ;
      Reader ir = new InputStreamReader(fis,"ISO8859_1") ;
         .....
} catch (UnsupportedEncodingException exc) {....
} catch (FileNotFoundException exc) {....
}
```

Le second argument du constructeur indique le mode de codage (“ISO8859-1” dans la plupart des pays européens, “Cp1252” sous Windows). La liste des codages acceptés se trouve dans la documentation sous `docs/guide/internat/encoding.doc.html` -voir aussi l’outil `native2ascii`- On notera qu’il existe un codage nommé UTF8 qui permet de fabriquer un flot de caractères UNICODE codés sur 8 bits.

Il existe également des constructeurs pour ces deux classes dans lesquels on ne précise pas le mode de codage: c’est le mode de codage de la plate-forme locale qui est pris par défaut.

Filtres

Les *filtres* sont des dispositifs qui transforment la manière dont un flot opère: bufferisation, conversion des octets en données Java, compressions, etc.

- Les filtres sont des flots dont les constructeurs utilisent toujours un autre flot:

```
public BufferedInputStream(InputStream in) ...
public DataInputStream(InputStream in)...
```

- On utilise les filtres en les associant avec un flot passé en paramètre du constructeur. Ce flot peut lui même être un autre filtre, on combine alors les comportements:

```
bis = new BufferedInputStream(new FileInputStream (fichier)) ;
dos = new DataOuputStream(new BufferedOutputStream (
         new FileOutputStream (fichierOut))) ;
```

- Chaque filtre dispose de méthodes spécifiques, certaines méthodes ont la propriété de remonter la chaîne des flots :

```
dos.writeInt(235);
dos.writeDouble(Math.PI);
dos.writeUTF("une chaîne accentuée") ; // chaîne écrite en UTF8
dos.flush() ; //transmis au BufferedOutputStream
...
dos.close() ; // ferme toute la chaîne
```

Filtres courants

InputStream/OutputStream

- `Buffered*` : la bufferisation permet de réaliser des E/S plus efficaces (en particulier sur le réseau).
- `Data*` : permet de lire/écrire des données primitives Java
- `Object*` : permet de lire/écrire des instances d'objets. Ce type de flot est très important (mais sort du périmètre de ce cours).
- `PushBackInputStream`: permet de "remettre" dans le flot des données déjà lues.
- `SequenceInputStream`: permet de considérer une liste ordonnée de flots comme un seul flot.

Reader/Writer

- `Buffered*` : bufferisation pour des caractères. Permet de lire un texte ligne à ligne.
- LineNumberReader : BufferedReader avec numérotation des lignes.
- `PrintWriter`: pour disposer des méthodes print/println (voir aussi `PrintStream`)
- `StreamTokenizer` : permet de faire de l'analyse syntaxique de texte.

Dans d'autre packages on trouvera des filtres spécialisés : java.util.zip.ZipInputStream, java.security.DigestInputStream,...

Il existe également un dispositif d'entrée/sortie qui ne s'inscrit pas dans la logique des Streams : `RandomAccessFile` (fichier à accès direct)

Exercices

Exercice ** testeur d'une calculette: :

Il vous est donné le code d'une Classe. Cette classe représente le "moteur" d'une calculette en notation polonaise inverse.

L'objectif de l'exercice est d'écrire un testeur des fonctionnalités de cette classe. Les opérations à effectuées seront lues dans un fichier, les résultats seront affichées sur la sortie standard.

Le fichier des opérations sera un texte simple :

- chaque ligne comporte soit un nombre soit une opération
- si c'est un nombre on le "pousse" sur la pile
- si c'est une opération on exécute l'opération et on affiche le résultat.

Exemple de fichier de test:

```
1234.00
345789.77
345
+
*
33.333333
/
```

Le code de la classe:

```
import java.util.* ;

/**
          * code d'une calculette en notation polonaise inverse
          * <P>
          * L'interface (ses méthodes publiques)
                    opère sur des chaînes de caractères
                    les calculs internes se font sur des flottants
          <P> attention aux exceptions runtime
   */


public class CalcDouble  {
          protected Stack stack = new Stack();

          /**
          * depile et rend un représentation chaîne de l'objet dépilé
          */
          public String popAsString() {
                    return  stack.pop().toString() ;
          }
          /**
          *  rend un représentation chaîne de l'objet sur sommet de la pile
          */
          public String peekAsString() {
                    return stack.peek().toString() ;
          }

          /**
          *  empile un objet correspondant à sa description par la chaîne
          */
          public void pushAsString(String str) {
                    stack.push( new Double(str)) ;
          }

          /**
          *  depile tout!
          */
          public void clear() {
                    stack.clear() ;
          }
```

```
        /*
         * réalise l'opération passée en paramètre, dépile les deux derniers
         * objets sur la pile, opère et rempile le résultat
        * <P> rend un représentation chaîne du résultat
        */
        public String perform(String operation) {
                char opChar = operation.charAt(0) ;
                Double arg2 = (Double)stack.pop() ;
                Double arg1 = (Double)stack.pop() ;
                Double res = null ;

                switch(opChar) {
                        case '+' :
                                res = new
Double(arg1.doubleValue() + arg2.doubleValue()) ;
                                break;
                        case '-' :
                                res = new
Double(arg1.doubleValue() - arg2.doubleValue()) ;
                                break;
                        case '*' :
                                res = new
Double(arg1.doubleValue() * arg2.doubleValue()) ;
                                break;
                        case '/' :
                                res = new
Double(arg1.doubleValue() / arg2.doubleValue()) ;
                                break;
                }
                if (res != null) {
                        stack.push(res) ;
                }
                return res == null? "" : res.toString() ;
        }

 }
```

Les applets 9

Points essentiels

- Les Applets constituents des petites applications Java hébergées au sein d'une page HTML, leur code est téléchargé par le navigateur.

- Une Applet est, à la base, un panneau graphique. Un protocole particulier la lie au navigateur qui la met en oeuvre (cycle de vie de l'Applet).

- Les codes qui s'exécutent au sein d'une Applet sont soumis à des restrictions de sécurité.

Applets

Une *Appliquette* (*Applet*) est une portion de code Java qui s'exécute dans l'environnement d'un navigateur au sein d'une page HTML. Elle diffère d'une application par son mode de lancement et son contexte d'exécution.

Une application autonome est associée au lancement d'un processus JVM qui invoque la méthode `main` d'une classe de démarrage.

Une JVM associée à un navigateur peut gérer plusieurs Applets, gérer leur contexte (éventuellement différents) et gérer leur "cycle de vie" (initialisation, phases d'activité, fin de vie).

Lancement d'une Applet

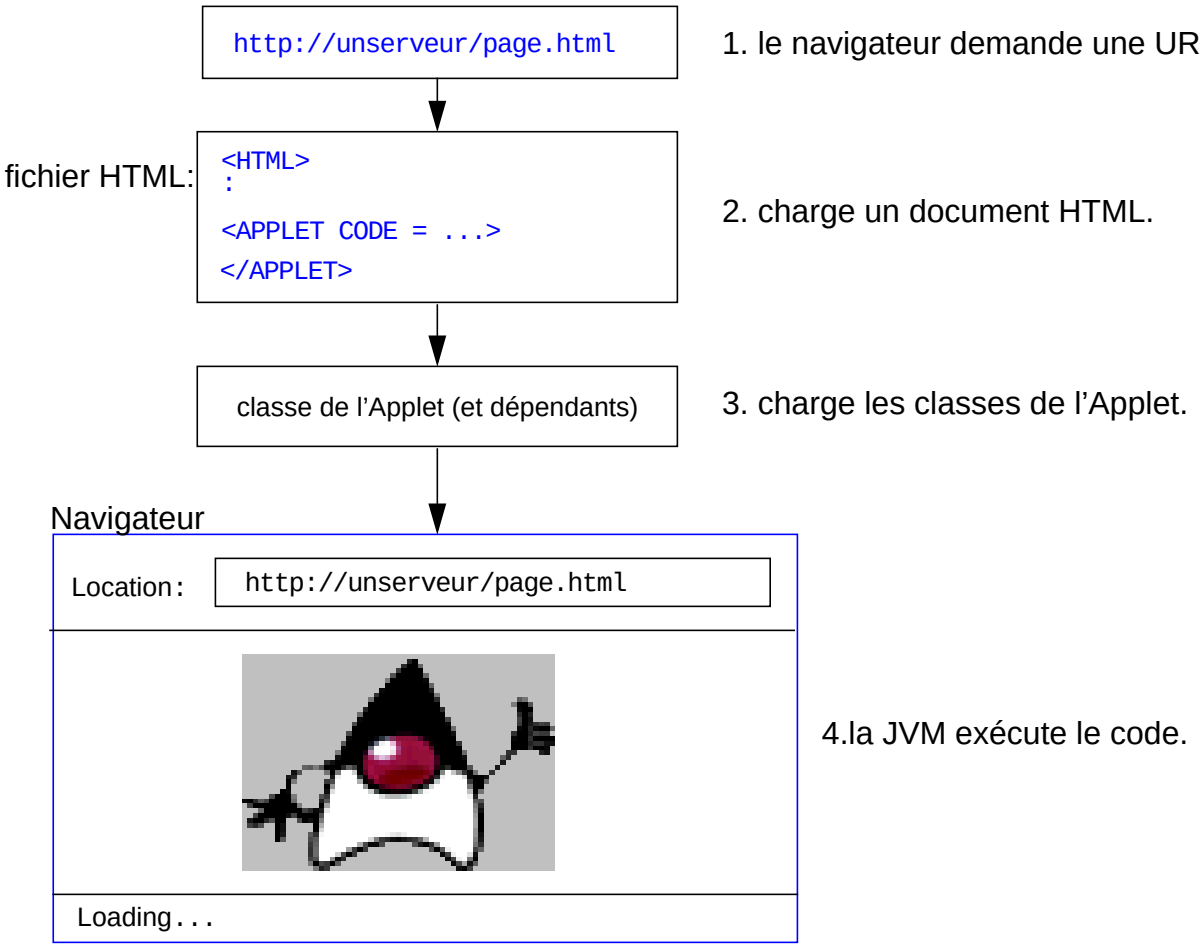
Une applet s'exécutant dans le cadre d'une "page" HTML, la requête de lancement et les paramètres associés sont en fait contenus dans le fichier source décrivant la page.

Le langage HTML (voir `http://www.w3.org/MarkUp`) permet de définir une présentation à partir d'un texte contenant des *balises* (qui sont des instructions de mise en page). Certaines de ces balises demandent le chargement de données non-textuelles comme des images ou des Applets Java.

Le lancement d'une Applet suppose donc :

- La désignation d'un document au navigateur au travers d'une adresse URL (voir `http://www.w3.org/Adressing`)
- Le chargement et la mise en page du document HTML associé
- Le chargement du code de l'Applet spécifié dans le document (et éventuellement le chargement des codes des classes distantes appelées par l'Applet).
- La gestion, par la JVM du navigateur, du cycle de vie de l'Applet au sein du document mis en page.

Applets: lancement



Exemple d'utilisation de la balise APPLET :

```
<P> et maintenant notre Applet :

<APPLET code="fr.gibis.applets.MonApplet.class"
        width=100 height=100>
</APPLET>
```

Applets: restrictions de sécurité

L'applet est chargée par un `ClassLoader` particulier. On a un contexte d'exécution lié à ce ClassLoader et associé au site distant dont est originaire l'Applet.

Dans ce contexte on va exécuter des codes de classe : classses "distantes" chargées par le ClassLoader et classes "système" de la librairie de la J.V.M locale. Pour éviter qu'un code distant puisse réaliser des opérations contraires à une politique de sécurité élémentaire, des règles par défaut s'appliquent (*sandbox security policy*):

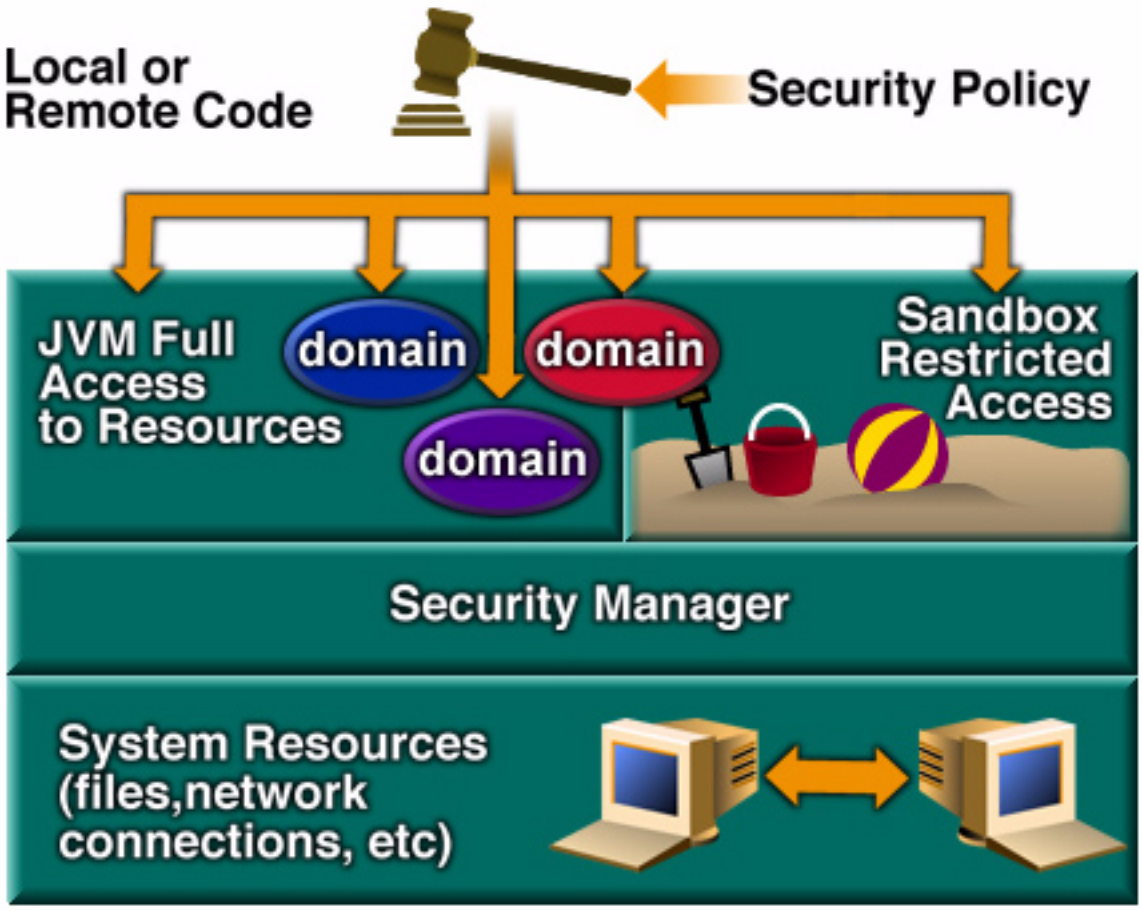
- L'Applet ne peut obtenir des informations sur le système courant (hormis quelques informations élémentaires comme la nature du système d'exploitation, le type de J.V.M., etc.).
- L'Applet ne peut connaître le système de fichiers local (et donc ne peut réaliser d'entrée/sortie sur des fichiers).
- L'Applet ne peut pas lancer de processus
- Les communications sur réseau (par *socket*) ne peuvent être établies qu'avec le site d'origine du code de l'Applet.

Bien entendu un code d'Applet ne peut pas contenir de code "natif" (par essence non portable).

A partir de la version 2 de Java un système standard de mise en place de domaines de sécurité (*Protection Domain*) permet d'assouplir ces règles pour permettre certaines opérations à des codes dûment authentifiés venus de sites connus. On peut ainsi imaginer que l'Applet qui vous permet de consulter votre compte en banque vous permet aussi de rapatrier des données à un endroit du système de fichier défini par vous.

Applets: restrictions de sécurité

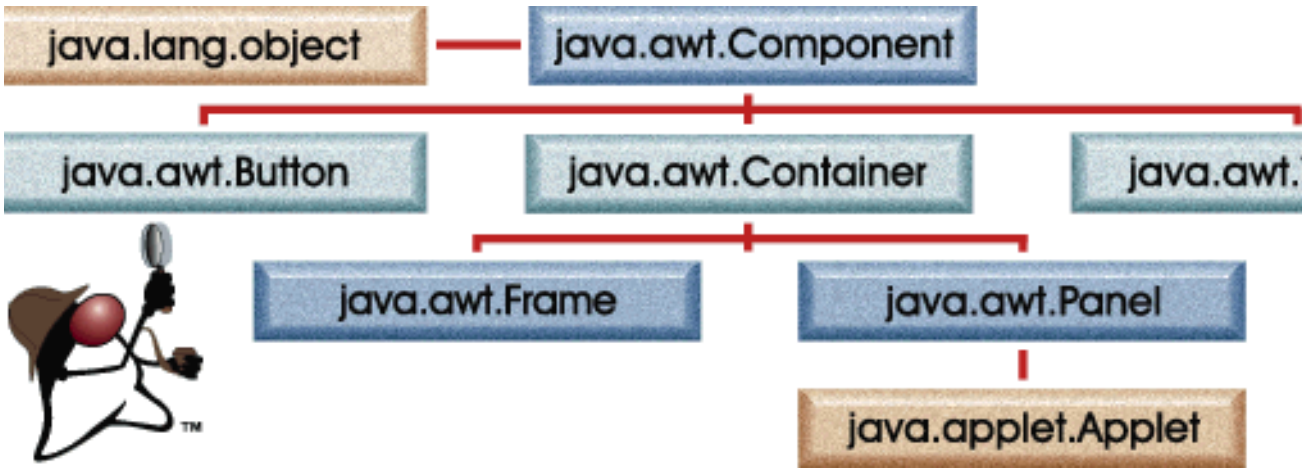
Le modèle de sécurité Java 2 s'applique dès qu'un `SecurityManager` est présent et ce aussi bien pour du code téléchargé que pour du code local



Hiérarchie de la classe Applet

De par l'incorporation d'une Applet à une présentation graphique (la "page" du navigateur) toute Applet est, de manière inhérente, une classe graphique (même si on peut créer des Applets qui n'ont aucune action de nature graphique). A la base l'incorporation d'une Applet dans une page HTML revient à signaler au navigateur qu'une zone de dimensions données n'est plus gérée directement par l'interpréteur de HTML mais par un programme autonome qui prend le contrôle de la représentation graphique de cette zone.

Toute classe qui est une Applet doit dériver de `java.applet.Applet` qui est elle-même une classe dérivée de `java.awt.Panel` qui représente un "panneau" graphique dans la librairie graphique AWT.



Applets: groupes de méthodes

Une Applet étant un `Container` graphique, elle hérite des méthodes qui permettent de disposer des composants graphiques dans la zone gérée par la classe. Ce point sera vu dans des chapitres ultérieurs, et, pour l'instant, nous allons nous attacher à trois groupes de méthodes:

- Méthodes qui sont appelées par le système graphique pour la notification d'une demande de rafraîchissement d'une zone de l'écran:
  `repaint(), update(graphics), paint(Graphics)`
  Dans le cas où l'Applet souhaite gérer le graphique de bas niveau (au lieu de laisser agir le système automatique attachés à des composant AWT de haut niveau) elle doit conserver un modèle logique des éléments graphiques qu'elle gère et être capable de les redessiner à la demande.

- Méthodes spécifiques aux Applets et qui leur permettent de demander au navigateur des informations ou des actions liées au contexte: informations sur la page HTML courante, chargement d'une ressource sur le site d'origine de la page, etc.

- Méthodes spécifiques aux Applets et à leur cycle de vie : le navigateur doit pouvoir notifier à l'Applet qu'il veut l'initialiser (`init()`), qu'il veut la rendre active ou inactive (`start(), stop()`), ou qu'il veut la "détruire" (`destroy()`).
  Par défaut ces méthodes ne font rien et il faut en redéfinir certaines d'entre elles pour obtenir un comportement de l'Applet.

H.T.M.L.: la balise `Applet`

Syntaxe des balises HTML :

```
<APPLET
        [archive=ListeArchives]
        code=package.NomApplet.class
         width=pixels height=pixels
        [codebase=codebaseURL]
        [alt=TexteAlternatif]
        [name=nomInstance]
        [align=alignement]
        [vspace=pixels]  [hspace=pixels]
>
  [<PARAM name=Attribut1 value=valeur>]
  [<PARAM name=Attribut2 value=valeur>]
  . . . . .
  [HTMLalternatif]
</APPLET>
```

Exemple :

```
<APPLET
        code=fr.acme.MonApplet.class
        width=300 height=400
>
</APPLET>
```

Evolutions futures (HTML 4) :

```
<OBJECT codetype=”application/java”
        classid="fr.acme.MonApplet.class"
        width=300 height=400>
        >
</OBJECT>
```

HTML: la balise `Applet`

. La balise HTML APPLET comporte les attributs suivants :

- code=*appletFile.class* - Cet attribut *obligatoire* fournit le nom du fichier contenant la classe compilée de l'applet (dérivée de `java.applet.Applet`). Son format pourrait également être *aPackage.appletFile.class.*

  Note – La localisation de ce fichier est relative à l'URL de base du fichier HTML de chargement de l'applet.

- width=*pixels* height=*pixels* - Ces attributs *obligatoires* fournissent la largeur et la hauteur initiales (en pixels) de la zone d'affichage de l'applet, sans compter les éventuelles fenêtres ou boîtes de dialogue affichées par l'Applet.

- codebase=*codebaseURL* - Cet attribut facultatif indique l'URL de base de l'applet : le répertoire contenant le code de l'applet. Si cet attribut n'est pas précisé, c'est l'URL du document qui est utilisé.

- name=*appletInstanceName* -- Cet attribut, facultatif, fournit un nom pour l'instance de l'applet et permet de ce fait aux applets situées sur la même page de se rechercher mutuellement (et de communiquer entre-elles).

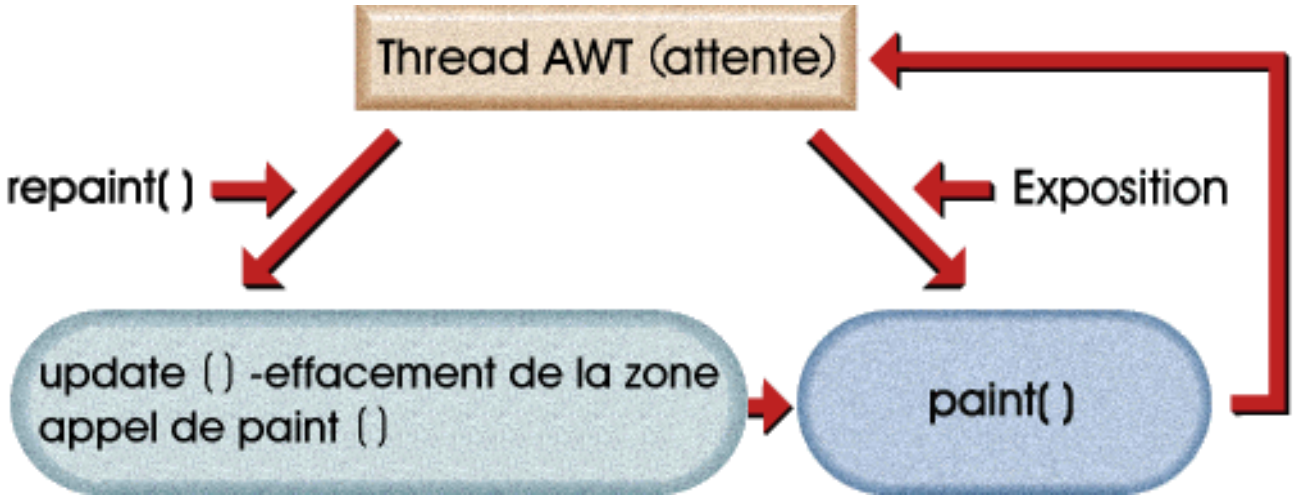
- archive=*ListeArchives* permet de spécifier une liste de fichiers archive .jar contenant les classes exécutables et, éventuellement des ressources. Les noms des archives sont séparés par des virgules. Ce point ne sera pas abordé dans ce cours.

- object=*objectFile.ser* permet de spécifier une instance d'objet à charger. Ce point ne sera pas abordé dans ce cours.

`<param name=`*appletAttribute1* `value=`*value*`>` -- Ces éléments permettent de spécifier un paramètre à l'applet. Les applets accèdent à leurs paramètres par la méthode `getParameter()`.

Méthodes du système graphique de bas niveau

Les mises à jour du système graphique de bas niveau :



- **`repaint()`** : demande de rafraichissement de la zone graphique. Cette demande est asynchrone et appelle update(Graphics). L'instance de `java.awt.Graphics` donne un contexte graphique qui permet aux méthodes de dessin d'opérer.

- **`update(Graphics`**) : par défaut efface la zone et appelle paint(Graphics). Cette méthode peut-être redéfinie pour éviter des papillotements de l'affichage (séries d'effacement / dessin).

- **`paint(Graphics)`** : la redéfinition de cette méthode permet de décrire des procédures logiques de (re)construction des dessins élémentaires qui constituent la présentation graphique.
  Lorsque l'on utilise des composants graphiques de haut niveau il n'est souvent pas nécessaire d'intervenir et le système AWT gère automatiquement ces (re)affichages.

Méthodes du système graphique de bas niveau

La méthode `paint()` prend un argument qui est une instance de la classe `java.awt.Graphics`. Cette classe fournit les primitives graphiques fondamentales.

Voici un exemple minimal d'Applet qui écrit du texte de manière graphique:

```
import java.awt.* ;
import java.applet.Applet ;

public class HelloWorld extends Applet {

         public void paint(Graphics gr) {
                   gr.drawString("Hello World?", 25 ,25) ;
         }
}
```

Les arguments numériques de la méthode drawString() sont les coordonnées x et y du début du texte. Ces coordonnées font référence à la "ligne de base" de la police. Mettre la coordonnée y à zero aurait fait disparaître la majeure partie du texte en haut de l'affichage (à l'exception des parties descendantes des lettres comme "p", "q" etc.)

Méthodes d'accès aux ressources de l'environnement

L'applet peut demander des ressources (généralement situées sur le réseau). Ces ressources sont désignées par des URLs (classe `java.net.URL`). Deux URL de référence sont importantes :

- l'URL du document HTML qui contient la description de la page courante. Cette URL est obtenue par `getDocumentBase()` .
- l'URL du code "racine" de l'Applet (celui qui est décrit par l'attribut "code"). Cette URL est obtenue par `getCodeBase()` .

En utilisant une de ces URLs comme point de base on peut demander des ressources comme des images ou des sons :

- `getImage(URL base, String désignation)` : permet d'aller rechercher une image; rend une instance de la classe **`Image`**.
- `getAudioClip(URL base, String désignation)` : permet d'aller rechercher un son; rend une instance de la classe **`AudioClip`**.

Les désignations de ressource par rapport à une URL de base peuvent comprendre des cheminoms relatifs (par ex: "../../images/truc.gif"). Attention toutefois : certaines configurations n'autorisent pas des remontées dans la hiérarchie des répertoires.

Le moteur son de la plateforme Java2 sait traiter des fichiers .wav, .aiff et .au ainsi que des ressources MIDI. Une nouvelle méthode `newAudioClip(URL)` permet de charger un AudioClip.

La méthode `getParameter(String nom)` permet de récupérer, dans le fichier source HTML de la page courante, la valeur d'un des éléments de l'applet courante, décrit par une balise `<PARAM>` et ayant l'attribut `name`=*nom* . Cette valeur est une chaîne `String`.

Méthodes d'accès aux ressources de l'environnement

Utilisation de la récupération de paramètres. Exemple de source HTML:

```
<APPLET code="fr.gibis.graph.Dessin.class" width=200 height=200>
        <PARAM name="image" value="duke.gif">
</APPLET>
```

Le code Java correspondant :

```
package fr.gibis.graph ;
import java.awt.*;
import java.applet.Applet;

public class Dessin extends Applet {
        Image img ;

        public void init() {// redef. méthode standard cycle vie
                String nomImage = getParameter("image") ;
                if ( null != nomImage) {
                        img = getImage(getDocumentBase(),
                                nomImage);
                }
        }

        public void paint (Graphics gr) {
                if( img != null) {
                        gr.drawImage(img,50,50, this) ;
                } else {
                        gr.drawString("image non chargée",
                                25,25) ;
                }
        }
}
```

Les méthodes de chargement de media comme `getImage()` sont des méthodes *asynchrones*. On revient de l'appel de la méthode alors que le chargement est en cours (et, possiblement, non terminé). Il est possible que `paint()` soit appelé plusieurs fois au fur et à mesure que l'image devient complètement disponible.

Méthodes du cycle de vie

`init()` : Cette méthode est appelée au moment où l'applet est créée et chargée pour la première fois dans un navigateur activé par Java (comme AppletViewer). L'applet peut utiliser cette méthode pour initialiser les valeurs des données. Cette méthode n'est pas appelée chaque fois que le navigateur ouvre la page contenant l'applet, mais seulement la première fois juste après le changement de l'applet.

La méthode `start()` est appelée pour indiquer que l'applet doit être "activée". Cette situation se produit au démarrage de l'applet, une fois la méthode `init()` terminée. Elle se produit également lorsque le navigateur est restauré après avoir été iconisé ou lorsque la page qui l'héberge redevient la page courante du navigateur. Cela signifie que l'applet peut utiliser cette méthode pour effectuer des tâches comme démarrer une animation ou jouer des sons.

```
public void start() {
        musicClip.play();
}
```

La méthode `stop()` est appelée lorsque l'applet cesse de "vivre". Cette situation se produit lorsque le navigateur est icônisé ou lorsque le navigateur présente une autre page que la page courante. L'applet peut utiliser cette méthode pour effectuer des tâches telles que l'arrêt d'une animation.

```
public void stop() {
        musicClip.stop();
}
```

Les méthodes `start()` et `stop()` forment en fait une paire, de sorte que `start()` peut servir à déclencher un comportement dans l'applet et `stop()` à désactiver ce comportement.

`destroy()` : Cette méthode est appelée avant que l'objet applet ne soit détruit c.a.d enlevé du cache du navigateur.

Méthodes du cycle de vie

```
// Suppose l'existence du fichier son "cuckoo.au"
// dans le répertoire "sounds" situé dans
// le répertoire du fichier HTML d'origine

import java.awt.Graphics;
import java.applet.*;

public class HwLoop extends Applet {
        AudioClip sound;

        public void init() {
                sound = getAudioClip(getDocumentBase(),
                        "sounds/cuckoo.au");
                // attention syntaxe d'URL!!!
        }

        public void paint(Graphics gr) {
                // méthode de dessin de java.awt.Graphics
                gr.drawString("Audio Test", 25, 25);
        }

        public void start () {
                sound.loop();
        }

        public void stop() {
                sound.stop();
        }
}
```

Exercices

En règle générale une Applet est exécutée au sein d'un navigateur comme Netscape Navigator. Néanmoins pour simplifier et accélérer le développement votre SDK est fourni avec un outil simple conçu pour ne visualiser que les Applets.

Cet outil est appletviewer et vous pouvez le lancer en lui passant un fichier HTML simplifié par la commande :

```
appletviewer url_ou_fichier
```

Exercice * un message en couleur :

Réaliser une Applet qui affiche un rectangle plein de couleur rouge de largeur 200, de hauteur 50 en coordonnées (50,25). Afficher dans ce rectangle un message de couleur verte "BONJOUR DE MON APPLET".

Consulter la documentation pour pouvoir utiliser les méthodes de la classe Graphics :

```
fillRect(...)
setColor(...)
drawString(...)
```

générer un fichier html associé prévoyant pour l'Applet un emplacement de 300 de largeur et de 100 de hauteur

Exercice * image et son:

Ecrire une Applet qui recoive en paramètre :

- La référence d'une image à charger (et à afficher).
- La référence d'un son à charger.\

Ce son doit démarrer au moment du démarrage de l'applet et doit s'arréter si on change de page.

I.H.M. portables : AWT 10

Points essentiels

- AWT permet des mises en page indépendantes de la plate-forme
- Les composants d'interaction graphiques sont disposés à l'intérieur de composants particuliers : les Containers.
- Cette disposition se fait automatiquement et est contrôlée par un gestionnaire de disposition associé au Container.
- Etude de quelques gestionnaires de disposition standard: FlowLayout, BorderLayout, GridLayout.

Le package AWT

Le package AWT fournit les composants fondamentaux pour réaliser des interactions graphiques (I.H.M. *interactions Homme-Machine*). Les logiciels écrits avec AWT sont portables: le même code s'exécutera sans modifications sur des machines différentes et des systèmes de fenêtrages différents.

La manière dont cette portabilité est assurée doit être mentionnée: pour un composant graphique comme `Button`, il existe dans la librairie locale de la JVM une classe de correspondance (*peer class*) qui s'appuie sur les caractéristiques du système de fenêtrage local. Ainsi quand on utilisera un `Button` Java on obtiendra un bouton Motif sous Motif, un bouton Windows sous Win*, un bouton Mac sur Apple, etc. Une implantation locale de la classe `Toolkit` assure la liaison entre les composants Java et les composants "natifs".

Une autre librairie graphique, `javax.swing`, s'appuie essentiellement sur des composants pur Java et offre donc une plus grande variété puisqu'on n'a pas besoin de disposer d'un composant correspondant existant sur tous les sytèmes de fenêtrage.

Parmi les classes du package AWT on trouve:

- des composants standard (`Window, Panel, Button, Label`,..) et des classes associées (`Menu, CheckboxGroup`,..)
- des gestionnaires de disposition (`LayoutManager,..`) qui permettent d'agencer des composants.
- des classes liées au graphique de bas niveau : `Graphics, Graphics2D, Color, Font, Image`,…
- des événements et gestionnaires de tâches graphiques: `AWTEvent, MediaTracker, PrintJob`,…
- des "structures de données" : `Point, Rectangle, Dimension`,…

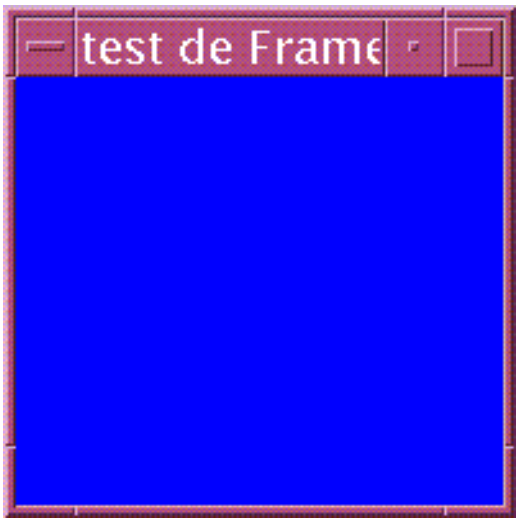
Composants et Containers

Les interfaces graphiques sont bâties à partir de composants comme des boutons (`Button`), des champs de saisie (`TextField`), des étiquettes (`Label`), etc. Ces composants dérivent de la classe `Component`.

Certains `Component`s sont des `Container`s c'est à dire des composants qui "contiennent" d'autres composants (y compris, éventuellement, d'autres `Container`s). Ainsi à la "racine" d'une application autonome on utilisera une instance dérivée de la classe `Window` comme `Frame` et on disposera "dans" cette fenêtre des composants Label, TextField ainsi que des `Container`s comme `Panel` (panneau) qui contiendront à leur tour d'autres composants.

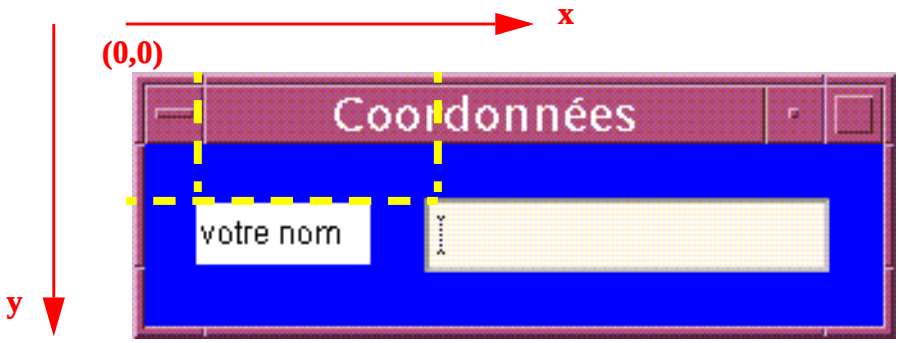
Exemple simple de mise en place de `Frame` sans contenu:

```
import java.awt.* ;

public class TestFrame{

   public static void main(String[] args) {
      Frame fenêtre = new Frame("test de Frame");
      fenêtre.setSize(200,200); // taille en points
      fenêtre.setBackground(Color.blue) ;
      fenêtre.show() ;
   }
}
```



Taille et position des composants: les gestionnaires de disposition

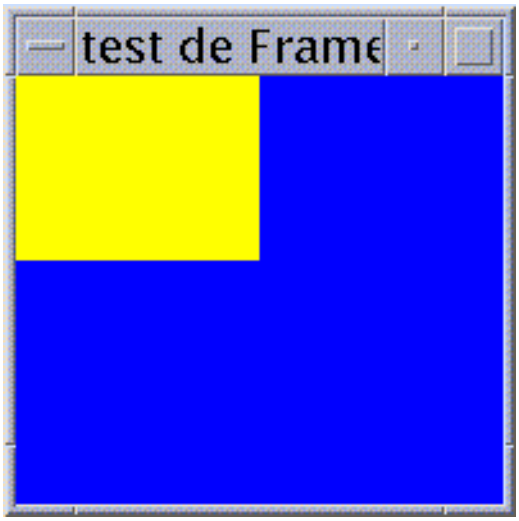
Quand on est “dans” un Container particulier on a affaire à un système de coordonnées qui part du coin supérieur gauche du container. Les composants peuvent se repérer par rapport à ce système de coordonnées exprimées en points.



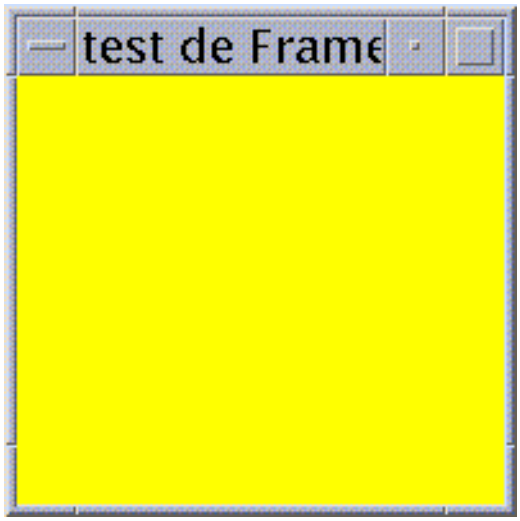
Bien qu’il soit possible de positionner les composants à l’intérieur d’un container en utilisant le système de coordonnées c’est une opération compliquée et surtout non portable. En Java on laisse généralement un gestionnaire de disposition (LayoutManager), associé au container, prendre en charge automatiquement le dimensionnement et la mise en place des composants.

Examinons le programme suivant:

```
import java.awt.* ;

public class TestAvecPanel{

   public static void main(String[] args) {
      Frame fenêtre = new Frame("test de Frame");
      fenêtre.setSize(200,200);
      fenêtre.setBackground(Color.blue) ;
      if (args.length != 0) { fenêtre.setLayout(null) ;}
      /* on rajoute un panneau */
      Panel panneau = new Panel();
      panneau.setBackground(Color.yellow);
      panneau.setSize(100,100);//ne sert pas si controlé par
Manager
      fenêtre.add(panneau);
      fenêtre.show() ;
   }
}
```



sans LayoutManager



avec LayoutManager

On a ici disposé un Panel à l'intérieur d'un Frame (`fenêtre.add(panneau)`). Ce Frame dispose par défaut d'un gestionnaire de disposition que l'on peut enlever (`fenêtre.setLayout(null)`). Dans ce cas on retrouve un panneau jaune de dimension 100,100 en coordonnées 0,0.

Si le gestionnaire de disposition du `Frame` reste actif, l'effet obtenu n'est pas celui qu'on pense car la dimension du panneau n'est plus sous contrôle direct et l'instruction `panneau.setSize(100,100)` est remise en cause par ce gestionnaire.

En fait cette prise de contrôle est avantageuse: en fonction d'une logique définie à l'avance le LayoutManager va disposer les composants et, éventuellement, changer leurs dimensions. Il suivra toutes les déformations du Container (demandées par la plate-forme, par l'utilisateur, par le Container englobant,...) et conservera la logique de la disposition.

Chaque type fondamental de Container a un LayoutManager par défaut qui peut être changé (par ex. par la méthode `setLayout(LayoutManager)`)

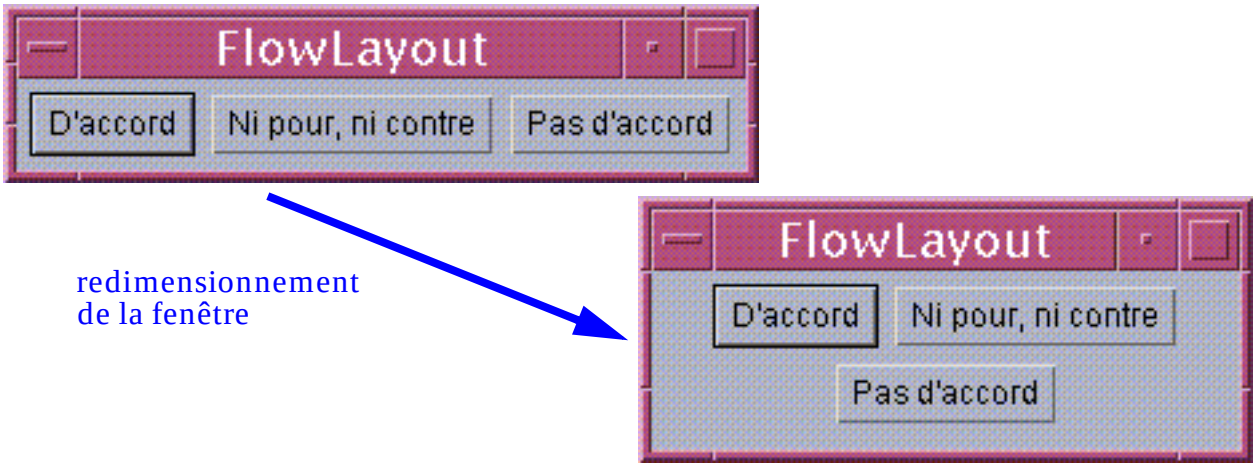
FlowLayout

Le gestionnaire de disposition FlowLayout positionne les composants “les uns à la suite des autres”. Au besoin il crée une nouvelle ligne pour positionner les composants qui ne tiennent pas dans la ligne courante.

A la différence de beaucoup d’autres gestionnaires de présentation une instance de FlowLayout ne modifie pas la taille des composants (tous les composants comportent une méthode `getPreferredSize()` qui est appelée par les gestionnaires de disposition pour demander quelle taille le composant voudrait avoir).

Exemple d’utilisation:

```
import java.awt.* ;

public class FlowFrame extends Frame{

   protected Component[] tb ;

   public FlowFrame(Component[] contenu) {
      super("FlowLayout") ;
      setLayout(new FlowLayout()) ;
      tb = contenu ;
      for ( int ix = 0 ; ix < tb.length; ix++) {
         add(tb[ix]);
      }
   }

   public void start() {
      pack();
      show() ;
   }

   public static void main(String[] args) {
      Button[] boutons = { new Button ("D’accord"),
         new Button ("Ni pour, ni contre"),
         new Button ("Pas d’accord"),
      } ;
      FlowFrame fenêtre = new FlowFrame(boutons) ;
      fenêtre.start() ;
   }
}
```



- La fenêtre (Frame) ne disposant pas de gestionnaire FlowLayout par défaut on la dote d'une instance de FlowLayout.
  Notons que le `FlowLayout` est le gestionnaire par défaut associé aux containers de type Panel.

- Il existe plusieurs manières de construire un FLowLayout qui permet de faire varier l'alignement (à gauche, à droite, centré,..) ou de faire varier la taille des gouttières séparant les composants.

- Les composants sont ajoutés au Container en employant la méthode `add(Component)` , bien entendu l'ordre des `add` est important puiqu'il détermine l'ordre des composants dans la disposition.

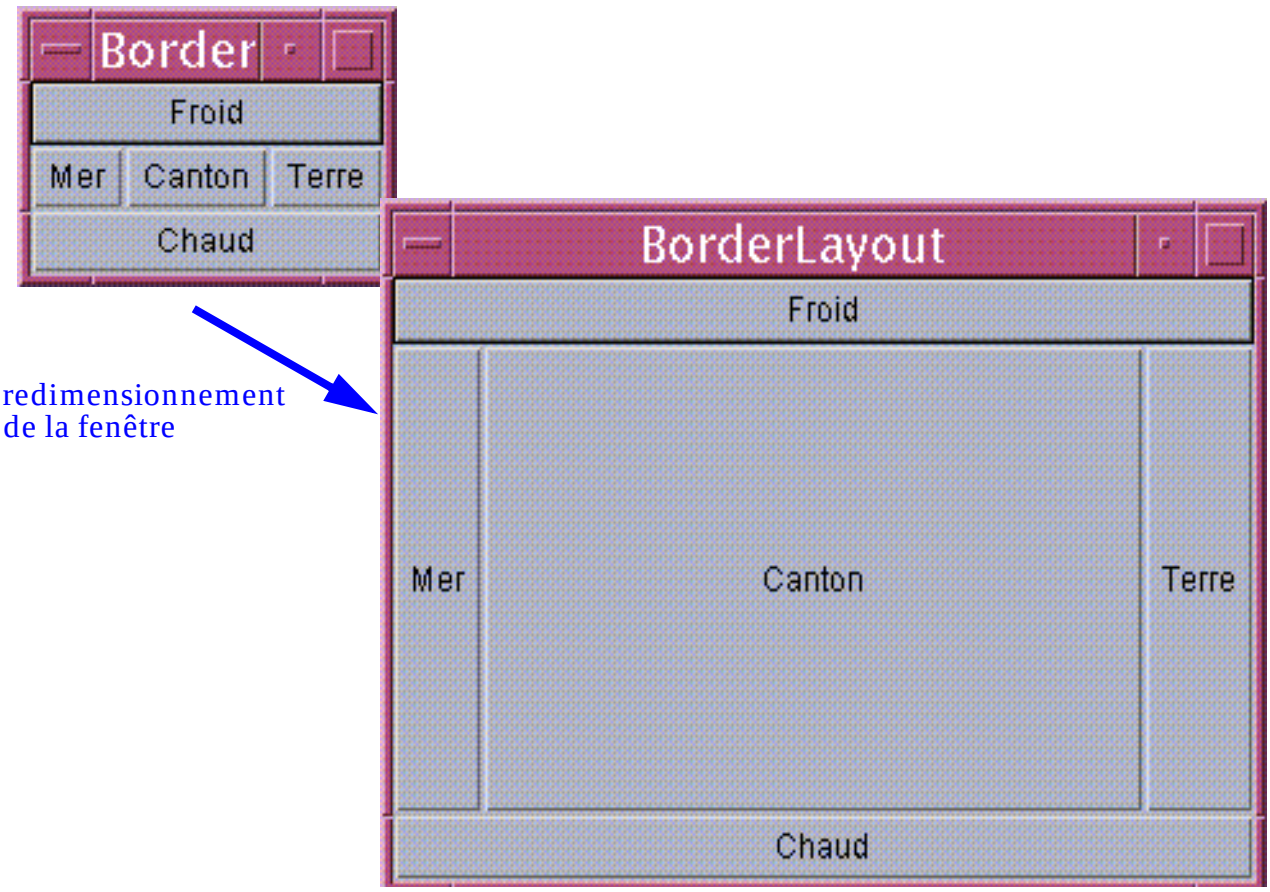
Notons également dans l'exemple l'emploi de la méthode `pack()` sur la fenêtre: plutôt que de fixer une taille arbitraire on demande à la fenêtre de prendre la plus petite taille possible en fonction des composants qu'elle contient.

BorderLayout

`BorderLayout` divise son espace de travail en cinq zone nommées et les composants sont ajoutés explicitement à une zone donnée. On ne peut placer qu'un composant par zone, mais toutes les zones ne sont pas nécessairement occupées.

Exemple d'utilisation:

```
import java.awt.* ;

public class BorderTest {
   private Frame fen;
   private Button nord, sud, est, ouest, centre ;

   public BorderTest() {
      // initialisations
      fen = new Frame("BorderLayout") ;
      nord = new Button("Froid") ;
      sud = new Button("Chaud") ;
      est = new Button("Terre") ;
      ouest = new Button("Mer") ;
      centre = new Button("Canton") ;

      // dispositions
      fen.add(nord, BorderLayout.NORTH) ;
      fen.add(sud, BorderLayout.SOUTH) ;
      fen.add(est, BorderLayout.EAST) ;
      fen.add(ouest, BorderLayout.WEST) ;
      fen.add(centre, BorderLayout.CENTER) ;
   }

   public void start() {
      fen.pack() ;
      fen.show() ;
   }

   public static void main(String[] args) {
      BorderTest bt = new BorderTest() ;
      bt.start() ;
   }
}
```



- Dans l'exemple `setLayout` n'a pas été utilisé car `BorderLayout` est le gestionnaire par défaut des containers de type `Window`.

- Les `add` sont qualifiés: il faut préciser dans quelle zone on met le composant (par défaut un `add` simple met le composant dans la zone centrale).

- En cas d'élargissement du container le gestionnaire respecte les hauteurs "préférées" des composants en NORTH et SOUTH, et les largeurs "préférées" des composants en EAST et WEST. La zone centrale tend à occuper toute la place restante dans les deux directions.

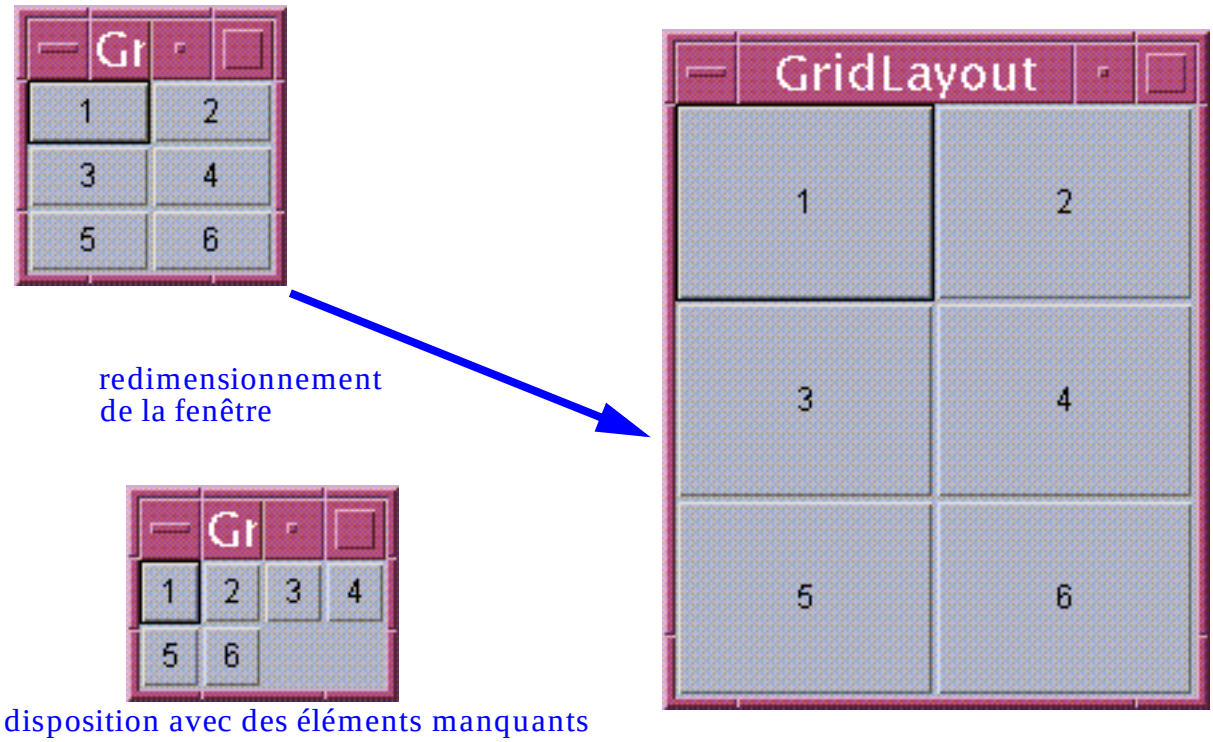
GridLayout

GridLayout permet de disposer les composants dans un tableau.

On peut créer un GridLayout en précisant les nombre de lignes et de colonnes (par exemple `new GridLayout(3,2)`).Toutes les lignes et les colonnes prennent la même dimension de façon à ce que les composants soient alignés.

Exemple d'utilisation:

```
import java.awt.* ;

public class ButtonPanel extends Panel{

   public ButtonPanel( int cols, String[] noms) {
      setLayout(new GridLayout(0, cols)) ;
      for (int ix = 0 ; ix < noms.length; ix ++) {
            add(new Button(noms[ix])) ;
      }
   }

   public static void main(String[] args) {
      Panel clavier = new ButtonPanel (
            Integer.parseInt(args[0]),
            new String[] { "1", "2", "3", "4", "5", "6", });
     Frame fenêtre = new Frame("GridLayout") ;
     fenêtre.add(clavier) ;
     fenêtre.pack() ;
     fenêtre.show() ;
   }
}
```



- Le constructeur de GridLayout permet de ne pas préciser une dimension (en la fixant à zéro) le gestionnaire calcule alors le nombre nécessaire de lignes ou de colonnes.
  Il existe également un constructeur permettant de fixer la taille des gouttières entre cellules.

- On utilise ici un `add` non qualifié: l'ordre des appels est important pour déterminer la position dans la grille.

- Le gestionnaire GridLayout ne respecte pas les tailles "préférées" des composants et tend à occuper tout l'espace nécessaire.

On notera, par ailleurs, que, dans l'exemple, la disposition s'est faite dans un `Panel` qui a été lui même ajouté à un `Frame`. Comme le gestionnaire de `Frame` est un `BorderLayout` et qu'on a utilisé un `add` non qualifié, le `Panel` occupe toute la zone centrale et fait office de "fond" sur la fenêtre principale.

Combinaisons complexes

Les présentations sont, en général, obtenues en utilisant des `Panels` intermédiaires d'agencement. Ce type d'imbrication est essentiel pour des présentation complexes.

Les effets des gestionnaires de présentation sont propagés d'un LayoutManager à l'autre. Pour calculer sa taille "préférée" un Panel opère en fonction des composants qu'il contient et des contraintes propres à son LayoutManager. Si le conteneur auquel il appartient demande une modification de dimension, le LayoutManager local va recalculer sa disposition et éventuellement retailler ses composants.

nota: `getParent()` permet d'obtenir le Container dans lequel se trouve le composant courant.

```
import java.awt.* ;

public class Barres extends Panel {
   protected Component fou ;
   protected Panel barre =
      new Panel(new FlowLayout(FlowLayout.LEFT)) ;
   protected Label base = new Label() ;

   public Barres(Component cpt) {
      super(new BorderLayout()) ;
      add(fou = cpt , BorderLayout.CENTER) ;
      add(barre, BorderLayout.NORTH);
      add(base, BorderLayout.SOUTH) ;
   }

   public void emBarre(Component cpt) {
      barre.add(cpt) ;
      validate() ;// demande action du LayoutManager
            // mais pas suffisant pour retailler conteneur
   }
   public void horsBarre(Component cpt) {
      barre.remove(cpt) ;
      validate() ;
   }
   public void sendMessage(String message) {
      base.setText(message) ;
   }
}
```

Utilisation du composant défini (et du composant “ButtonPanel” précédemment défini):

```
import java.awt.* ;

public class BarZoom  {

   public static void main(String[] args) throws Exception {
      Panel carter = new ButtonPanel ( 2,
            new String[] { "Venus", "Terre", "Jupiter", "Mars" });
      Barres fond = new Barres(carter) ;
      Frame fen = new Frame("testBarre") ;
      fen.add(fond);
         // le “add” non qualifié avec un BorderLayout met
         // dans la zone CENTER (comme il n’y a que cette zone
         // le “fond” occupe toute la fenêtre)
      fen.pack();
      fen.show();
      for(int ix = 0 ; ix < 20; ix++) {
         Thread.sleep(1000);// permet d’attendre un peu
         fond.sendMessage("on  ajoute le bouton " +ix) ;
         fond.emBarre(new Button(""+ix));
         fen.pack() ;
      }
   }
}
```

Autres gestionnaires de disposition

Dans le package AWT on trouve deux autres gestionnaires de disposition standard:

- `CardLayout` : permet de disposer des composants en pile les uns au dessus des autres. Un seul composant est visible et, par programme, on peut remettre un des composant sur le “dessus” de la pile.

- `GridBagLayout` : est un gestionnaire très utile, très souple et très complexe. Il permet de disposer des composants dans une grille. Les lignes et les colonnes ne sont pas toutes de même taille et l’occupation des cellules peut être finement paramétrée (respect des tailles “préférées”, position dans la cellule, occupation de plusieurs cellules, etc.).

L’étude de ces gestionnaires sort du périmètre de ce cours.

Exercices

Exercice * dessin d'une calculette :

Dans une fenêtre autonome (`Frame`) on disposera des boutons (`Button`) représentant les touches d'une calculette et un `TextField` destiné à afficher les résultats.

Comme il s'agit d'une calculette à notation polonaise inverse (que nous programmerons par la suite) on rajoutera les touches :

- "Enter" (pour ajouter une valeur dans la pile)
- "Pop" (pour dépiler)
- "Clear" (pour effacer la saisie courante)
- "AllClear" (pour effacer la pile)

NOTE IMPORTANTE : à ce stade vous ne pouvez pas "sortir" de votre application en fermant la fenêtre (option "close"). Générer une interruption au clavier (^C) pour mettre fin à l'exécution du programme.

Exercice * dessin d'une calculette dans une Applet:

On reprendra l'exercice précédent mais, cette fois, le conteneur de base est une Applet et non un Frame.

thème de réflexion : quelles idées auriez-vous pour faire converger les codes des applications autonomes avec ceux des Applets?

Interfaces et classes abstraites 11

Points essentiels

Notions avancés de programmation Objet en Java: les types abstraits

- Interfaces
- Conception avec des Interfaces Java
- Les classes abstraites

Types abstraits

Nous avons vu qu'un type "objet" de Java pouvait être considéré comme un "contrat". Lorsqu'une référence est déclarée on s'intéresse à un ensemble de propriétés de l'objet désigné, mais pas nécessairement à toutes les propriétés effectives de l'objet. Ainsi il n'est pas choquant d'écrire:

```
Reader ir = new InputStreamReader(fis,"ISO8859_1") ;
```

Cette instruction est possible car la relation d'héritage permet d'assurer que le "contrat" de type est effectivement rempli par l'objet référencé.

On pourrait imaginer que le polymorphisme (cette capacité à ne s'intéresser qu'à certaines propriétés) puisse s'appliquer au travers de types abstraits purs. C'est à dire :

- des types définis par ce qu'il font et non par ce qu'il sont.
- des types tels que le "contrat" de type puisse être réalisé par des objets n'ayant aucune relation de parenté (c'est dire des objets qui appartiennent à des hiérarchies d'héritage différentes).

Imaginons, par exemple, une application bancaire dans laquelle on définisse une classe "Client" :

```
public class Client {
          .....
          Messager messager ;
          .....
```

L'idée du champ "messager" est d'avoir un dispositif qui permette de prévenir le client de la réalisation d'une opération.

Selon les clients ces dispositifs peuvent être: un simple courrier, un fax, ou un courrier électronique. On aura donc un service qui pourra être rendu par des objets de nature très différente. Et pourtant le "service" est le même partout, on a besoin d'un objet qui sache répondre au "contrat" décrit par :

```
public void envoiMessage(String message)
```

Types abstraits

En Java la défintition de ce “service” se ferait par:

```
public interface Messager {
          public void envoiMessage(String message);
}
```

Les classes qui seraient effectivement capables de rendre le service seraient définies par :

-

```
public class Fax extends Telephone implements Messager {
          public void envoiMessage(String message) {
          ...
```

-

```
public class EMail extends AgentReseau implements Messager {
          public void envoiMessage(String message) {
          ...
```

-

```
public class Courrier extends Imprime implements Messager {
          public void envoiMessage(String message) {
          ...
```

“Messager” va maintenant s’employer comme un véritable type:

```
          Client dupond, durand ;
          ....
          dupond.messager = new Fax(“0141331733”) ;
          durand.messager = new EMail(“durand@schtroumpf.fr”) ;
          ....
          for (int ct= 0 ; ct < tableauClient.length; ct++){
                    tableauClient[ct].messager.envoiMessage(
                              “Tout va bien!”);
          }
```

Déclarations d'interface

Une interface est un type qui se déclare d'une manière analogue à une classe:

- Une interface publique par fichier source. Une interface fait partie d'un package.
- Le fichier source doit porter le même nom que l'interface
- La compilation génère un fichier ".class" (par ex. Messager.class)

La déclaration d'une interface ne contient que:

- des en-têtes de méthodes publiques. Il n'y a pas de "corps" de définition des méthodes (on dit qu'elles sont *abstraites*). La déclaration peut comporter des clauses `throws`.
- des déclarations de constantes statiques.

Exemple:

```
package fr.gibi.util ;

public interface Messager {
         public static final int NORMAL = 0 ;
         public static final int URGENT = 1;
         public static final int NON_URGENT = 2 ;

         public void envoiMessage(String message)
                   throws ExceptionRoutage, ExceptionDelai ;
                   // pas de corps: méthode abstraite
}
```

Noter aussi la possibilité de définir des interfaces dérivées comme:

```
public interface MessagerFiable extends Messager, AgentFiable {
         //on réunit automatiquement les contrats des 2 interfaces
}
```

Réalisations d'une interface

Lorsqu'on déclare une classe qui réalise effectivement le contrat d'interface, il ne suffit pas d'implanter les méthodes du "contrat" il faut explicitement préciser "je connais la sémantique du contrat décrit par cette interface, et je m'engage à la respecter" et ceci se fait par la déclaration `implements`. Par ailleurs une même classe peut s'engager sur divers contrats d'interface :

```
public class Fax extends Telephone
         implements Messager,ConstantesGIBI,//interf. spécifiques
         Cloneable,Comparable, Serializable{ // standard
```

Il est possible de tester si un objet correspond à un contrat d'interface en faisant appel à l'opérateur `instanceof`. Certaines interfaces sont d'ailleurs juste des "marqueurs": il n'y a aucun corps dans leur déclaration et elles sont utilisées pour signaler une propriété d'un objet (c'est le cas de `Cloneable` et de `Serializable` dans l'exemple).

A partir du moment où un objet implante une interface il est possible de l'utiliser avec une référence du type de l'interface :

```
monClient.messager = new Fax(numero) ;
```

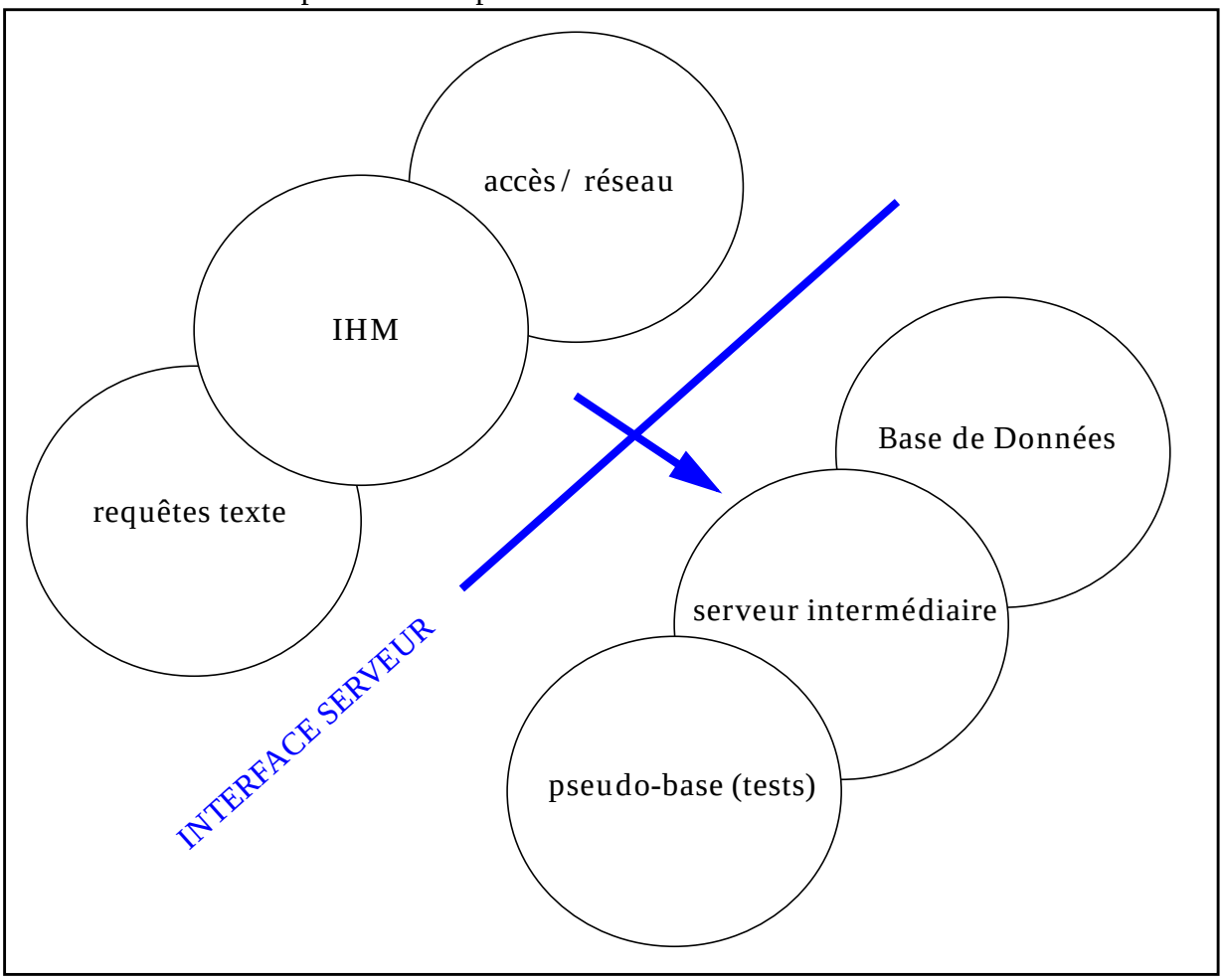
Autre Exemple :

```
public static void envoiEtArchivage(
                              String message,
                              Messager messager) {
         ....
}

...
Utils.envoiEtArchivage("Tout va très bien!", faxCourant) ;
```

Lorsqu'un objet réalise un interface ses descendants (les classes qui en dérivent) héritent de cette propriété.

Conception avec des interfaces

Les interfaces sont un mécanisme essentiel de la conception en Java. Elle permettent un **découplage** entre un service et sa réalisation. En définissant une architecture basée sur des interfaces on obtient une architecture plus modulaire dans laquelle un service donné peut être rendu par des composant différents



Dans cet exemple le service défini par “interface serveur” peut être rendu par un programme simulant le comportement d’une base de données (important pour les tests de la partie “cliente” du service) puis par une vraie base ou par un serveur intermédiaire (dans une architecture à trois parties).

Conception avec des interfaces

On trouvera de nombreuses utilisations des interfaces dans les *collections* du package `java.util`.

Une collection permet de regrouper un ensemble d'objets, le choix de la méthode de stockage se fait en fonction de la manière dont on veut accéder aux objets (objets ordonnés: `List`, objets dans un dictionnaire `Map`, etc.), ou en fonction d'autre propriétés (ensembles sans duplication: `Set`, etc.).

Supposons maintenant que l'on veuille obtenir tous les objets d'une collection sans s'intéresser à la réalisation effective de cette collection. On demandera alors à la collection de nous fournir un itérateur de type Iterator :

```
Iterator iter = maCollection.iterator() ;
while(iter.hasNext()){
        System.out.println(iter.next());
}
```

`Iterator` est une interface (tout comme `Collection` d'ailleurs).

Les classes abstraites

Une démarche courante en conception objet est d'abstraire: plusieurs concepts qui semblent superficiellement différents (et qui sont rendus par des classes différentes) peuvent être associés au travers d'une super-classe commune qui va factoriser les comportements.

Cette super-classe n'est pas nécessairement complète: certaines parties de son comportement sont définies "contractuellement" (comme dans une interface Java) mais ne connaissent pas de réalisation concrète. On a ainsi une super-classe qui dispose éventuellement de variables et de méthodes membres, de constructeurs, etc. mais pour laquelle la reálisation effective de certaines méthodes reste en suspens.

On a donc une classe (*classe abstraite*) pour laquelle l'instanciation est impossible et qui impose à ses classes dérivées de définir des comportements effectifs pour les méthodes *abstraites*.

Exemple: dans une application graphique on veut définir des objets géométriques `UnRectangle`, `UnOvale`, `UnLosange`, etc... ces différentes figures s'inscrivent dans un rectangle ayant des coordonnées, une largeur, une hauteur (la classe AWT `Rectangle` sert à décrire cela). Pour simplifier la gestion de ces différents objets on peut les faire dériver d'une classe abstraite Figure :

```
import java.awt.* ;
public abstract class Figure {
        protected Rectangle posDims ;
        protected Figure(Rectangle def) {
                posDims = def;
        }
        public abstract void paint(Graphics gr) ;
        public String toString() {
                return posDims.toString() ;
        }
        ... // autre méthodes communes
}
```

Chacune des figures concrètes qui deriveront de `Figure` devra savoir comment utiliser la méthode `paint(Graphics)` pour se dessiner.

Ainsi la classe `UnOvale` pourrait s'écrire :

```
public class UnOvale extends Figure {
        public UnOvale (Rectangle def) {
                super(def);
        }
        public UnOvale(int x, int y, int w, int h) {
                this(new Rectangle(x,y,w,h))
        }
        public void paint(Graphics gr) {
                gr.drawOval(posDims.x, posDims.y,
                        posDims.width, posDims.height);
        }
        ...
}
```

Bien qu'on ne puisse créer d'instance dont le type effectif soit une classe abstraite on peut utiliser une classe abstraite pour typer une référence:

```
public class MonApplet extends Applet {
        Figure[] tbFig = { new UnOvale(0,0, 12, 12),
                new UnLosange(20,40,33,33),
                ...
                } ;
        ....
        public void paint(Graphics gr) {
                ...
                for (int ix = 0 ; ix <tbFig.length ; ix++) {
                        tbFig[ix].paint(gr) ;
                }
        }
}
```

- A partir du moment où on déclare une méthode `abstract` la classe doit être marquée `abstract`.
- Si une sous-classe d'une classe abstraire ne fournit pas de réalisation pour une des méthodes abstraites dont elle hérite elle doit, à son tour, être marquée `abstract`.
- Bien entendu seules des méthodes d'instance peuvent être `abstract`.

Exercices

Exercice ** définition et utilisation d'une interface "moteur de calculette NPI":

Dans le cadre des exercices sur les entrées/sorties on a utilisé une classe utilisant une pile de Double et pouvant être utilisée comme "moteur de calculette en notation polonaise inverse" et on avait mis en place un testeur pour cette classe.

L'objectif de cet exercice est de préparer une réalisation ultérieure dans laquelle l'interface graphique représentant la calculette et le "moteur" soient découplés. C'est à dire que l'on pourra utiliser des "moteurs" différents pour la même interface (par exemple en calculant avec des java.util.BigDecimal) et que ces moteurs pourront être testés indépendamment de l'interface graphique.

On définira donc une interface Java décrivant le service de calculette (s'inspirer du code de CalcDouble.java) on réecrira CalcDouble et une nouvelle classe CalcInf (pour le calcul en précision infinie grace àl'utilisation de java.util.BigDecimal) qui implanteront toutes deux l'interface du service de calcul.

On adaptera le testeur de manière à ce qu'un test puisse de dérouler indifferemment sur un "moteur" ou sur un autre.

(exercice complémentaire) adapter ce testeur de manière à ce qu'il découvre dynamiquement la classe qui lui est passée en paramètre d'appel.
Pour créer dynamiquement une instance d'une classe dotée d'un constructeur sans paramêtre :

```
....    Class.forname(nomDeClasse).newInstance() ;
```

Le modèle d'événements AWT 12

Points essentiels

Le traitement des événements permet d'associer des comportements à des présentations AWT :

- Il faut associer un gestionnaire d'événement à un composant sur lequel on veut surveiller un type donné d'événement.
- A chaque type d'événement correspond un contrat d'interface.
- Lorsque la réalisation de ce contrat d'interface conduit à un code inutilement bavard on peut faire dériver le gestionnaire d'événement d'une classe "Adapter".

Les événements

Lorsque l'utilisateur effectue une action au niveau de l'interface utilisateur, un *événement* est émis. Les événements sont des objets qui décrivent ce qui s'est produit. Il existe différents types de classes d'événements pour décrire des catégories différentes d'actions utilisateur.

Sources d'événements

Un événement (au niveau de l'interface utilisateur) est le résultat d'une action utilisateur sur un composant AWT source. A titre d'exemple, un clic de la souris sur un composant bouton génère un ActionEvent. L'ActionEvent est un objet contenant des informations sur le statut de l'événement par exemple:

- `getActionCommand()` : renvoie le nom de commande associé à l'action.
- `getModifiers()` : renvoie la combinaison des "modificateurs", c'est à dire la combinaison des touches que l'on a maintenues pressées pendant le click (touche Shift par exemple)..

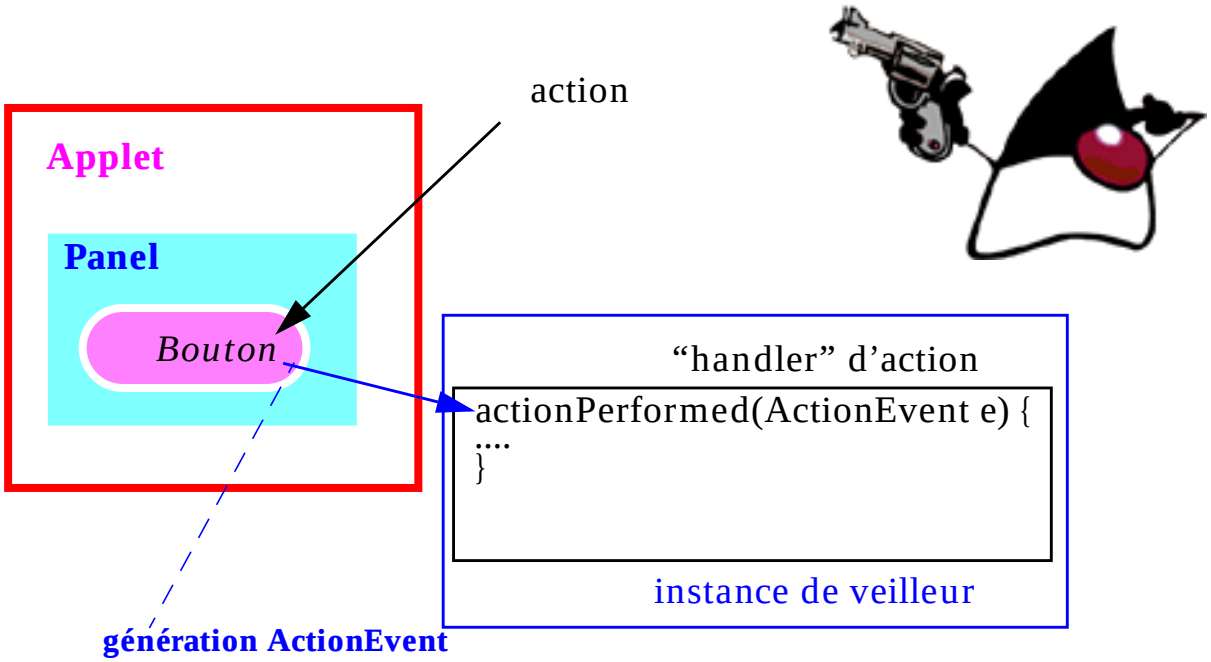
Handlers d'événements

Un "handler" d'événement est une méthode qui reçoit un objet `Event` , l'analyse et traite l'interaction utilisateur.

Modèle d'événements

Modèle par délégation

JDK 1.1 a introduit un nouveau modèle d'événement appelé modèle d'événement par délégation. Dans un modèle d'événement par délégation, les événements sont envoyés au composant, mais c'est à chaque composant d'enregistrer une routine de traitement d'événement (appelé veilleur: `Listener`) pour recevoir l'événement. De cette façon, le traitement d'événement peut figurer dans une classe distincte du composant. Le traitement de l'événement est ensuite délégué à une classe séparée.



modèle d'événements

Les événements sont des objets qui ne sont envoyés qu'aux veilleurs enregistrés. A chaque type d'événement est associé une interface d'écoute correspondante.

A titre d'exemple, voici une fenêtre simple comportant un seul bouton :

```
import java.awt.*;
public class TestButton {
   public static void main (String args[]){
      Frame fr = new Frame ("Test");
      Button bt = new Button("Appuyer!");
      bt.addActionListener(new ButtonHandler());
      fr.add(bt,BorderLayout.CENTER);
      fr.pack();
      fr.setVisible(true);
   }
}
```

La classe ButtonHandler définit une instance de traitement de l'événement .

```
import java.awt.event.*;
public class ButtonHandler implements
                              ActionListener{
   public void actionPerformed(ActionEvent evt) {
      System.out.println("Quelque chose s'est produit!");
   }
}
```

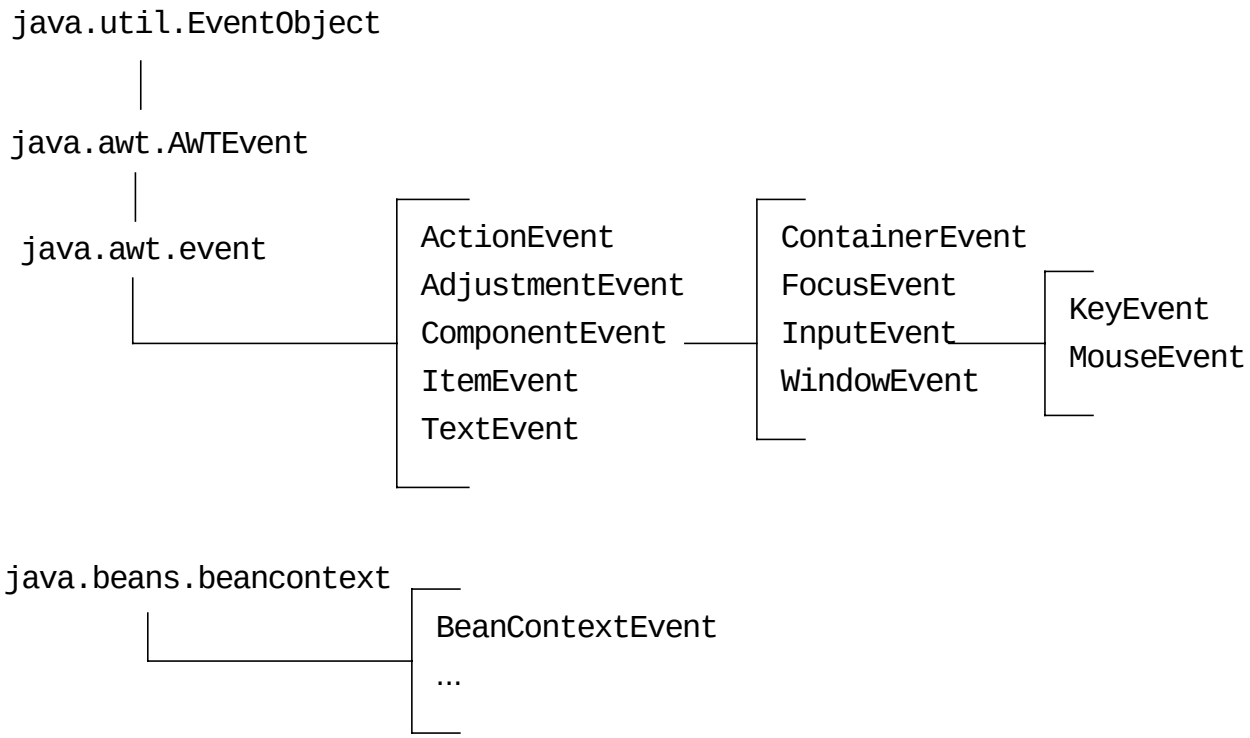
Modèle d'événements

- La classe `Button` comporte une méthode `addActionListener(ActionListener)`.
- L'interface `ActionListener` définit une seule méthode , `actionPerformed` qui recevra un `ActionEvent`.
- Lorsqu'un objet de la classe `Button` est créé, l'objet peut enregistrer un veilleur pour les `ActionEvent` par l'intermédiaire de la méthode `addActionListener`, on passe en paramètre un objet d'une classe qui "implémente" l'interface `ActionListener`.
- Lorsque l'on clique sur l'objet Bouton avec la souris, un `ActionEvent` est envoyé à chaque `ActionListener` enregistré et la méthode `actionPerformed (ActionEvent)est invoquée`.

Catégories d'événements

Plusieurs événements sont définis dans le package `java.awt.event`, et d'autres événements sont définis ailleurs dans l'API standard..

Pour chaque catégorie d'événements, il existe une interface qui doit être "implémentée" par toute classe souhaitant recevoir ces événements. Cette interface exige qu'une ou plusieurs méthodes soient définies dans les classes de réalisation. Ces méthodes sont appelées lorsque des événements particuliers surviennent. Le tableau de la page suivante liste les catégories et indique le nom de l'interface correspondante ainsi que les méthodes associées. Les noms de méthodes sont des mnémoniques indiquant les conditions générant l'appel de la méthode



On remarquera qu'il existe des événements de bas niveau (une touche est pressée, on clique la souris) et des événements abstraits de haut niveau (Action = sur un bouton on a cliqué, sur un TextField on a fait un <retour chariot>, ...)

Tableau des interfaces de veille

Catégorie	Interface	Methodes
Action	ActionListener	actionPerformed(ActionEvent)
Item	ItemListener	itemStateChanged(ItemEvent)
Mouse Motion	MouseMotionListener	mouseDragged(MouseEvent) mouseMoved(MouseEvent)
Mouse	MouseListener	mousePressed(MouseEvent) mouseReleased(MouseEvent) mouseEntered(MouseEvent) mouseExited(MouseEvent) mouseClicked(MouseEvent)
Key	KeyListener	keyPressed(KeyEvent) keyReleased(KeyEvent) keyTyped(KeyEvent)
Focus	FocusListener	focusGained(FocusEvent) focusLost(FocusEvent)
Adjustement	AdjustmentListener	adjustementValueChanged(AdjustementEvt)
Component	ComponentListener	componentMoved(ComponentEvent) componentHidden(ComponentEvent) componentResize(ComponentEvent) componentShown(ComponentEvent)
Window	WindowListener	windowClosing(WindowEvent) windowOpened(WindowEvent) windowIconified(WindowEvent) windowDeiconified(WindowEvent) windowClosed(WindowEvent) windowActivated(WindowEvent) windowDeactivated(WindowEvent)
Container	ContainerListener	componentAdded(ContainerEvent) componentRemoved(ContainerEvent)
Text	TextListener	textValueChanged(TextEvent)

Evénements générés par les composants AWT

Composant AWT	*Action*	adjust.	compnt	cont	focs	item	key	mse	mtn	text	win
Button	●		●		●		●	●	●		
Canvas			●		●		●	●	●		
Checkbox			●		●	●	●	●	●		
CheckboxMenuItem						●					
Choice			●		●	●	●	●	●		
Component			●		●		●	●	●		
Container			●	●	●		●	●	●		
Dialog			●	●	●		●	●	●		●
Frame			●	●	●		●	●	●		●
Label			●		●		●	●	●		
List	●		●		●	●	●	●	●		
MenuItem	●										
Panel			●	●	●		●	●	●		
Scrollbar		●	●		●		●	●	●		
ScrollPane			●	●	●		●	●	●		
TextArea			●		●		●	●	●	●	
TextComponent			●		●		●	●	●	●	
TextField	●		●		●		●	●	●	●	
Window			●	●	●		●	●	●		●

Détails sur les mécanismes

Obtention d'informations sur un événement

Lorsque les méthodes de traitement, telles que `mouseDragged()` sont appelées, elles reçoivent un argument qui peut contenir des informations importantes sur l'événement initial. Pour savoir en détail quelles informations sont disponibles pour chaque catégorie d'événement, reportez-vous à la documentation relative à la classe considérée dans le package `java.awt.event`.

Récepteurs multiples

La structure d'écoute des événements AWT permet actuellement d'associer plusieurs veilleurs au même composant. En général, si on veut écrire un programme qui effectue plusieurs actions basées sur un même événement, il est préférable de coder ce comportement dans la méthode de traitement.

Cependant, la conception d'un programme exige parfois que plusieurs parties non liées du même programme réagissent au même événement. Cette situation peut se produire si, par exemple, un système d'aide contextuel est ajouté à un programme existant.

Le mécanisme d'écoute permet d'appeler une méthode `add*Listener` aussi souvent que nécessaire en spécifiant autant de veilleurs différents que la conception l'exige. Les méthodes de traitement de tous les veilleurs enregistrés sont appelées lorsque l'événement survient.

L'ordre d'appel des méthodes de traitement n'est pas défini. En général, si cet ordre a une importance, les méthodes de traitement ne sont pas liées et on ne doit pas utiliser cette fonction pour les appeler. Au lieu de cela, il faut enregistrer simplement le premier écouteur et faire en sorte qu'il appelle directement les autres. C'est ce qu'on appelle un multiplexeur d'événements

Adaptateurs d'événements

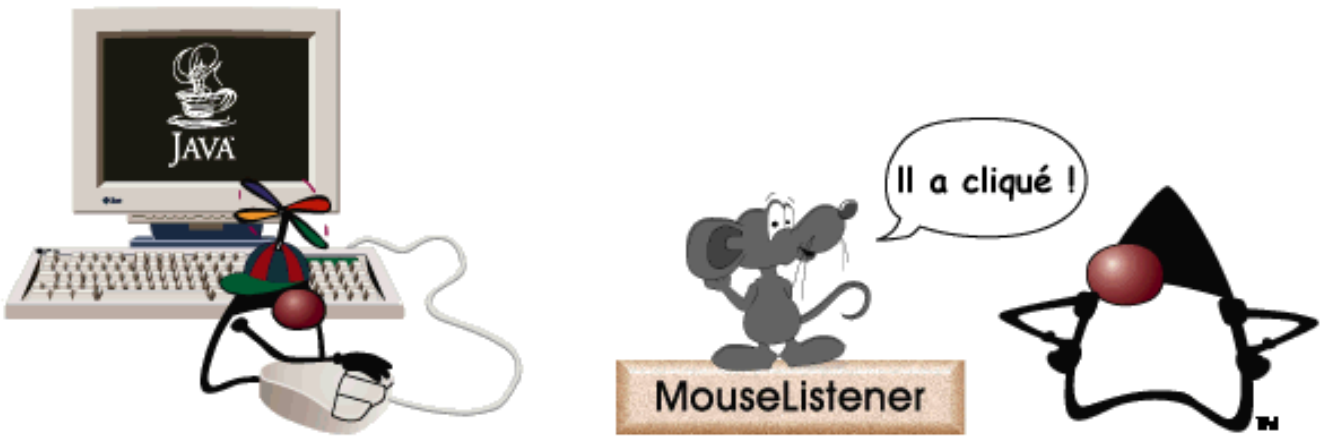
Il est évident que la nécessité d'implanter toutes les méthodes de chaque interface de veille représente beaucoup de travail, en particulier pour les interfaces MouseListener et ComponentListener.

A titre d'exemple, l'interface MouseListener définit les méthodes suivantes :

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

Pour des raisons pratiques, Java fournit une classe d'*adaptateurs* pour pratiquement chaque interface de veille, cette classe implante l'interface appropriée, mais ne définit pas les actions associées à chaque méthode.

De cette façon, la routine d'écoute que l'on définit peut hériter de la classe d'adaptateurs et ne surcharger que des méthodes choisies.



Adaptateurs d´événements

Par exemple :

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

public class MouseClickHandler extends MouseAdapter {

  //Nous avons seulement besoin du traitement mouseClick,
  //nous utilisons donc l'adaptateur pour ne pas avoir à
  //écrire toutes les méthodes de traitement d'événement

  public void mouseClicked (MouseEvent evt) {
    //Faire quelque chose avec le clic de la souris . . .
  }
}
```

Attention à la spécialisation de méthodes: une déclaration comme
`public void mouseclicked(MouseEvent evt)`
est légale et crée simplement une nouvelle méthode par contre le veilleur construit ne réagira pas au moment de la mise en oeuvre.

Exercices

Exercice * destruction d'une fenêtre:

On reprendra l' exercice précédent (disposition d'une calculette,...)

On captera sur le `Frame` l'événement de fermeture de fenêtre (`windowClosing`) et on arrétera l'application par

```
System.exit(0)
```

Classes membres, classes locales 13

Points essentiels

Aspects avancés de la programmation en Java: définition de classes “à l’intérieur” d’une autre classe.

Introduction: un problème d'organisation

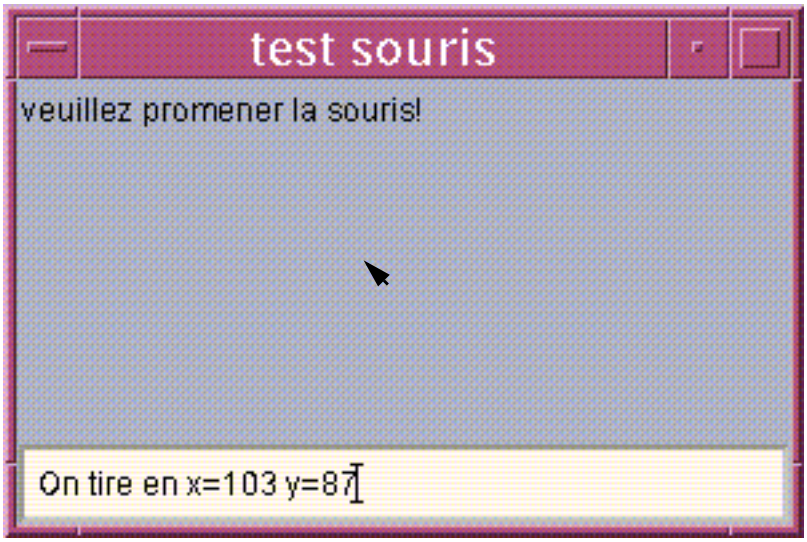
Soit le programme suivant :

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

public class IHMsouris {

   private Frame frame  = new Frame("test souris");
   private TextField txt = new TextField(30) ;
   private int initszH ;
   private int initszW ;

   public IHMsouris(int largeur, int hauteur) {
      initszH = hauteur ;
      initszW = largeur ;
      frame.add(new Label("veuillez promener la souris!"),
         BorderLayout.NORTH) ;

      frame.add(txt,BorderLayout.SOUTH) ;
      .....
   }// constructeur

   public void start() {
      frame.setSize(initszW, initszH) ;
      frame.show() ;
   }

   public static void main (String[] args) {
      IHMsouris ihm = new IHMsouris(300,200) ;
      ihm.start() ;
   }
}
```

L'objectif est ici d'avoir une zone centrale (dans un BorderLayout) dans laquelle on puisse suivre le déplacement d'une souris qu'on "tire". Un message informatif doit être affiché dans un autre composant ("txt").

L'événement à suivre est `MouseMotion` et la méthode qui nous intéresse est `mouseDragged(MouseEvent)`

Il faut compléter le code de manière à obtenir à peu près l'aspect et le comportement suivant :



Envisageons diverses solutions:

- Ecrire une classe externe qui dérive de MouseMotionAdapter.
  Pour que le veilleur puisse manipuler le `TextField` correspondant il faudra passer ce composant en paramètre du constructeur:
  ```
  public MyAdapter(TextComponent tc) { ....
  ```
  puis faire dans le code :
  ```
  frame.addMouseMotionListener(new MyAdapter(txt));
  ```

- Faire en sorte que la classe "IHMsouris" implante l'interface `MouseMotionListener` et que la méthode réalisant `mouseDragged` accède au `TextField`.

Dans un cas on crée une classe générale *ad hoc* pour répondre à un besoin très particulier. Dans l'autre cas on a un code bavard qui peut, en plus, devenir extraordinairement compliqué s'il faut gérer différents suivis de souris en fonction des composants sur lesquels on "tire".

Il est possible d'écrire ce code autrement en utilisant des classes locales imbriquées.

Introduction: organisation avec une classe locale

Voici une manière d'écrire le code correspondant au problème précédent:

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

public class IHMsouris {

   private Frame frame  = new Frame("test souris");
   private TextField txt = new TextField(30) ;
   private int initszH ;
   private int initszW ;

   public IHMsouris(int largeur, int hauteur) {
      initszH = hauteur ;
      initszW = largeur ;
      frame.add(new Label("veuillez promener la souris!"),
         BorderLayout.NORTH) ;

      frame.add(txt,BorderLayout.SOUTH) ;
      frame.addMouseMotionListener( new MouseMotionAdapter() {
         public void mouseDragged(MouseEvent evt) {
            txt.setText("On tire en x=" + evt.getX()
                     + " y=" + evt.getY()) ;
         }
      });
   }// constructeur

   public void start() {
      frame.setSize(initszW, initszH) ;
      frame.show() ;
   }

   public static void main (String[] args) {
      IHMsouris ihm = new IHMsouris(300,200) ;
      ihm.start() ;
   }
}
```

On a ici utilisé une classe locale anonyme pour répondre à des besoins très locaux.

Analysons le code :

- `new MouseMotionAdapter() {....}`
  permet d'invoquer un constructeur d'une classe que nous définnissons comme dérivant de MouseMotionAdapter. C'est, sous une forme condensée, l'écriture de
  ```
  class MouseMotionAdapter1
   extends MouseMotionAdapter { ...}
  ```
  et de
  ```
  new MouseMotionAdapter1() ...
  ```

- Cette nouvelle classe ne porte pas de nom (d'où "classe anonyme") et cette invocation / définition rappelle également les tableaux anonymes:
  ```
  method( new String[] {"un", "deux"}) ;
  ```
  Ceci dit il y a effectivement un nouveau fichier ".class" qui est généré et qui porte un nom lié à la classe hôte + un numéro (ici `IHMsouris$1.class`).Ce fichier ".class" devra toujours accompagner celui de la classe qui permet de le définir.

- Cette instance de nouvelle classe est "imbriquée" avec celle de la classe englobante: toutes les deux ont accès au champ "txt" bien que celui-ci soit privé dans la définition de la classe "IHMsouris".

Nous venons d'avoir ici un aperçu sur d'autres caractéristiques de Java:

- Il est possible de définir des classes à l'intérieur d'une classe. Ces classes peuvent être des classes membres (au même titre qu'un champ ou une méthode membre) ou des classes locales (définies dans un bloc).

- Les classes membres d'instance ou les classes locales peuvent, sous certaines conditions, avoir un accès privilégié aux membres de l'instance englobante (ou un accès à des variables locales).

Classes et interfaces membres statiques

On peut définir une classe ou une interface comme membre statique d'une classe.

```
public class Operation {

   public static class Stats {
      long appels;
      double ratio ;
   } // classe utilitaire

   private static Stats stats = new Stats() ;
         .....
         // méthodes statiques modifiant les statistiques
         ....
   public static Stats getStats() { return stats;}
}
```

Ici la classe "interne" peut être utilisée de la manière suivante :

```
Operation.Stats stat = Operation.getStats() ;
```

On pourrait également invoquer un constructeur avec la même convention de nommage.

Bien entendu le fait d'imbriquer les définitions de classe doit correspondre à un lien de dépendance conceptuelle.Toutefois on a ici plus qu'une simple hiérarchie analogue à la hiérarchie des packages. En effet la classe ou l'interface interne bénéficie des droits d'accès à l'intérieur de la classe englobante.

La classe interne peut également être `private` ou `protected` (ce que ne peuvent pas être les autres classes).

Classes et interfaces membres statiques

Classes membres d'instance

Une classe peut être un membre d'instance.

```
public class MyFrame extends Frame{

        public class ButtonListener implements ActionListener{
                public void actionPerformed(ActionEvent evt) {
                        ....
                }
        }
}
```

Ici on a une classe interne liée à une instance. Depuis une autre classe la classe interne est de type `MyFrame.ButtonListener`. Par contre un constructeur de `ButtonListener` a besoin d'une instance de `MyFrame`. Ainsi , par exemple, depuis le "main"(statique) de `MyFrame`:

```
 MyFrame fen = new MyFrame() ;
MyFrame.ButtonListener bl = fen.new ButtonListener();
```

Une telle classe à la posssibilité d'accéder aux autres membres de l'instance englobante.

Classes membres d'instance

ex. réalisation d'une interface par une classe privée:

`java.util.Enumeration` est une interface publique permettant de référencer une instance d'objet qui permet de parcourir une collection. Un tel objet doit avoir deux méthodes: `boolean hasMoreElement()` et `Object nextElement();`

```
// la classe Pile gère une pile d'objets
public class Pile {
        private Object[] tableExtensible ;
        private int sommetDePile ;
        ...
        private class ParcoursPile
                  implements java.util.Enumeration {
                  int index = sommetDePile ;
                  public boolean hasMoreElements(){
                   return index >= 0;
                  }
                  public Object nextElement(){
                   return tableExtensible[index--];
                  }
        }// fin parcoursPile
        public java.util.Enumeration elements() {
         return new ParcoursPile();
        }
        ...
}
```

Classes dans un bloc

Il est possible de définir une classe locale dans un bloc de méthode.

Une telle classe peut accéder soit à des variables d'instance, soit à des variables locales (ou des paramètres de la méthode).

```
public class MyFrame extends Frame {
        protected TexField messager = new TextField(30) ;
        ....
        public Button createButton(String nom,final String mess){
                Button res = new Button(nom) ;
                class BListener implements ActionListener{
                        public void
                         actionPerformed(ActionEvent evt) {
                         messager.setText(mess) ;
                        }
                }
                Blistener list = new BListener() ;
                ...
                res.addActionListener(list) ;
                return res;
        }
        ...
}
```

L'accès à des variables locales nécessite que celles-ci soient marquées `final`.

Classes anonymes

L'utilisation de classes anonymes permet parfois une simplification de l'écriture de classes locales.

```
public class MyFrame extends Frame {
        protected TexField messager = new TextField(30) ;
        ....
        public Button createButton(String nom,final String mess){
                Button res = new Button(nom) ;
                res.addActionListener(new ActionListener(){
                        public void
                         actionPerformed(ActionEvent evt) {
                         messager.setText(mess) ;
                        }
                });
                return res;
        }
        ...
}
```

On remarquera dans l'exemple la notation :

```
new ActionListener(){...}
```

`ActionListener` étant une interface on a en fait une écriture synthétique qui marque la création d'un nouvel objet qui "implémente" l'interface considérée.

On ne peut pas définir un constructeur dans une classe anonyme, mais on peut l'invoquer au travers d'un constructeur de la classe mère:
`new ClasseMere(arg1, arg2) { ...` *redéfinitions* `...}`

Récapitulation: architecture d'une déclaration de classe

Plan général des déclarations de classe premier niveau:

```
package yyy;
import z.UneCLasse ;
import w.* ;

class XX {
        // ordre indifférent sauf pour les expressions
        // d'initialisation de premier niveau

        //MEMBRES STATIQUES
        variables statiques
                  var. statiques initialisées
                  constantes de classe
        méthodes statiques
        classes ou interfaces statiques

        //MEMBRES D'INSTANCE
        variables d'instance
                  var.d'instance initialisées
                  var d'instance "blank final"
        méthodes d'instance
                  "patrons" de méthodes d'instance (abstract,...)
        classes d'instance
                  classes avec accès instance englobante

        //BLOCS DE PREMIER NIVEAU
        blocs statiques
         /* évalués au moment du chargement de la classe */
        blocs d'instance
         /* évalués au moment de la création de l'instance */

        // CONSTRUCTEURS
        constructeurs

}
```

Exercices

Exercice ** mise en place du fonctionnement de la calculette

Reprendre l'exercice sur la calculette , et faire fonctionner cette calculette (c.a.d mettre en place des veilleurs d'évenements sur les boutons et mettre en place les réactions associées).

On s'appuiera sur les moteurs de calcul précédemment définis.

Annexe: Packages fondamentaux 14

Points essentiels

- Les “packages” standard constituent les librairies qui doivent accompagner toute JVM.
- Ils fournissent, de manière harmonisée, un cadre pour des opérations essentielles :
  - services communs
  - structures de données, fonctions de calcul
  - interactions graphiques portables
  - entrée / sorties,
  - accès réseau
  - sécurité
  - internationalisation
  - composants beans
  - APIs et classes pour accès SQL, pour l’invocation distante (RMI) et pour CORBA

java.lang

Ce package est automatiquement importé dans tout source Java.

On y trouve des objets et des services fondamentaux.

- La classe racine : `Object` et interfaces utilitaires: `Cloneable`, `Comparable`
- Introspection et exécution dynamique : `Class`, `Package`, package `java.lang.reflect`, …
- Exécution : `ClassLoader`, `SecurityManager`,
- Services dépendant du contexte local d'exécution : `System`, `Runtime`, `Process`,
- Processus légers : `Thread`, `ThreadGroup`, interface `Runnable`
- Manipulations avancées sur la gestion des variables: `ThreadLocal`, package `java.lang.ref`
- Racine des classes exceptions : `Throwable`
- Classes "d'encapsulation" des types scalaires : `Integer`, `Byte`, `Short`, `Float`, `Double`, (`+ Number`), `Character`, `Boolean`, (`+Void`)
- Chaînes de caractères : `String`, `StringBuffer`
- Fonctions mathématiques : `Math`

java.lang

Classes d'encapsulation

Dans certaines situations (collections d'objets, paramètres génériques,...) on ne peut pas utiliser une valeur primitive scalaire mais on peut utiliser une instance d'une classe d'encapsulation (`Integer`, `Float`, `Character`, etc.)

Chacune de ces instances encapsule une valeur scalaire immuable. Les constructeurs, méthodes, méthodes statiques fournissent de nombreux services associés :

```
Integer oInt = new Integer(500);
Integer oInt2 = new Integer("500");
short val = oInt.shortValue();
int valeur = Integer.parseInt("500") ;
String hexS = Integer.toHexString(500) ;
```

StringBuffer

Les objets de type String étant également immuables il peut être préférable, lorsqu'on doit faire des opérations de transformation de chaîne de caractères, d'utiliser des objets `StringBuffer`.

```
StringBuffer buf = new StringBuffer("Nacho");
buf.insert(2,"bu").append(new
StringBuffer("sonid").reverse()).append("aure ").append(6) ;
String res = buf.toString() ;
```

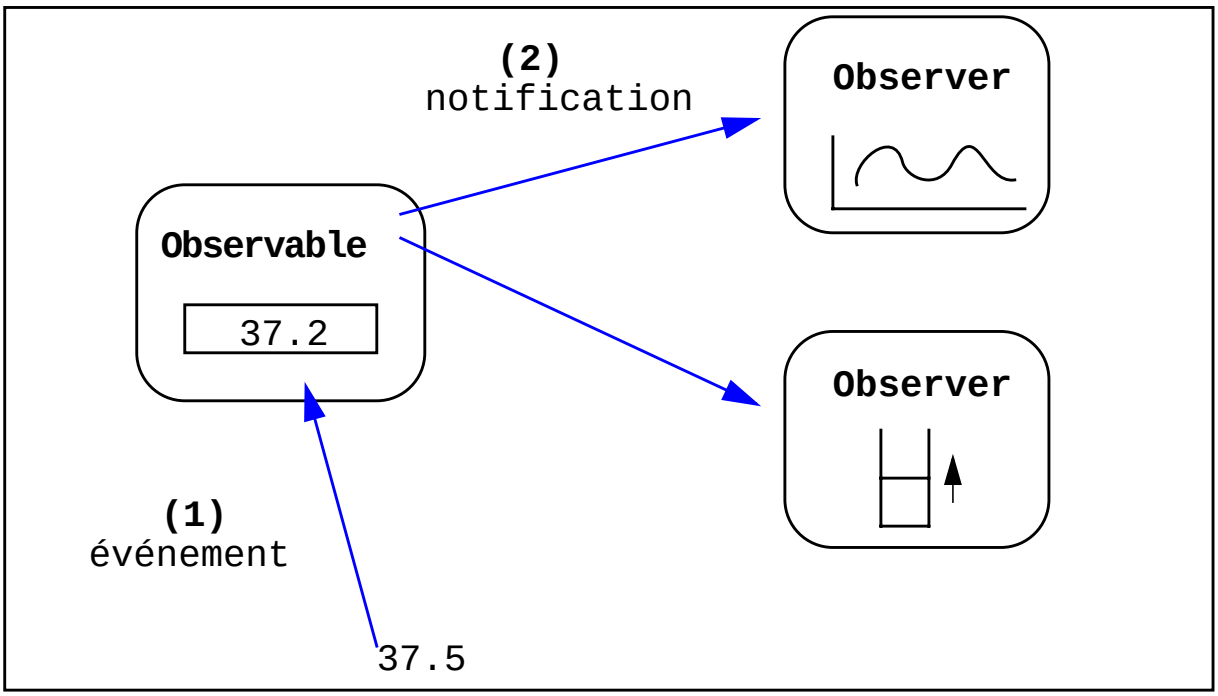
java.util

Utilitaires et structures de données pour le programmeur:

- Classes et interfaces pour les Collections :
  - Ensembles (`Set`): `HashSet`
  - Listes (`List`) : `LinkedList`, `ArrayList`, `Vector`, `Stack`,....
  - Dictionnaires (`Map`) : `HashMap`, `HashTable`, `Properties`....
  - arbres: `TreeMap, TreeSet`
  - Utilitaires pour collections: interfaces de parcours `Enumeration`, `Iterator` . Fonctions de recherche, de tris, etc. sur des collections (`Collections`), des tableaux (`Arrays`) en utilisant les interfaces `Comparator` ou `java.lang.Comparable`.
- Gestion du temps: `Date`, `TimeZone`, classe abstraite `Calendar` et le calendrier standard `GregorianCalendar`.
- “patterns” : `Observer`/`Observable`
- .....

java.util

Modèle Observer/Observable




```
public class CoursBourse extends Observable {
          private HashTable cours ;
          ....
          public void changeCote(String valeur, Money cote){
                    cours.put(valeur,cote) ;
                    notifyObservers(valeur) ;
          }
          ....
}
```

```
public class AfficheCours extends ... implements Observer {
          // A la création s'enregistre auprès de l'Observabl
          ....
          public void update(Observable cours, Object valeur)
                    // Met à jour l'affichage
          }
}
```

Internationalisation (i18n)

L'internationalisation des applications est favorisée par l'emploi du jeu de caractères UNICODE et par des classes et packages comme:

- `java.util.Locale` : permet de désigner des "aires culturelles" hiérarchisées. Par exemple l'aire du Français, sous-ensemble Canadien.

- Les classes dérivées de `java.util.ResourceBundle`: permettent d'organiser des hiérarchies de ressources. Si, par exemple, on ne trouve pas un mot dans le dictionnaire spécifique au Canadien-Français on va le chercher dans le dictionnaire du Français

- Le package `java.text` permet de traiter des problèmes d'adaptation de formats de dates, de formats des nombres, de ternir compte de l'ordre alphabétique local, de mettre en place des messages paramétrables, etc.

- Le package `java.awt` permet de traiter des dispositions dépendantes de l'aire culturelle (par ex. de droite à gauche si besoin est).

Numeriques divers

- `java.math.BigInteger`:
  permet de réaliser des opérations sur des entiers de taille illimitée

- `java.math.BigDecimal`:
  permet de réaliser des calculs avec des nombres décimaux. La taille des nombres n'est pas limitée et on peut spécifier la précision et la manière de réaliser des arrondis. Convient bien pour représenter des valeurs financières.

- `java.util.Random`:
  permet de générer des séries de nombres aléatoires.

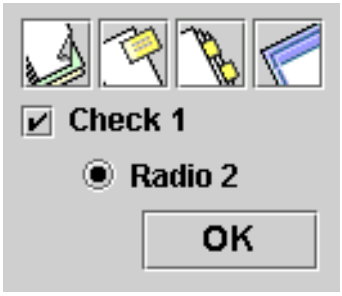
Interactions graphiques portables : AWT, SWING

Il existe deux bibliothèques d'interface graphique : AWT fournit un petit nombre de composants sur une base native, javax.swing fournit des composants pur java (composants "dessinés") plus variés mais plus complexes.

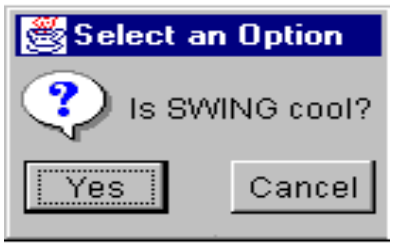
- AWT de base :
  - Bibliothèque de composants simples (disponibles sur toutes plateformes de fenêtrage),
  - Dessin graphique simple et 2D de base,
  - Technique de disposition des composants (auto-adaptation à la taille des fenêtres)
  - Technique de gestion des événements
- Sous-packages spécialisés : coupé/collé (`dnd`, `datatransfer`), 2D (`geom`), `image`,`...`
- "Extensions" livrées en standard : `accessibility`,Swing

Interactions graphiques portables : AWT, SWING

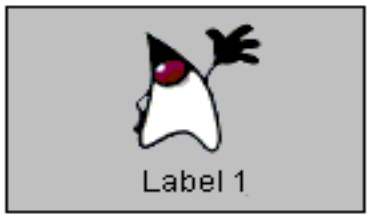
Exemples de composants Swing:



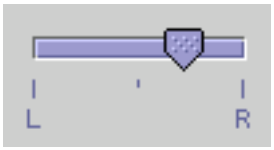
JButton



JOptionPane

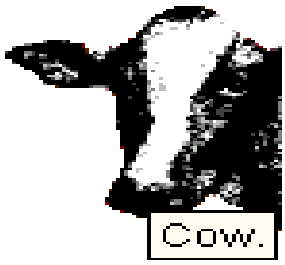


JLabel

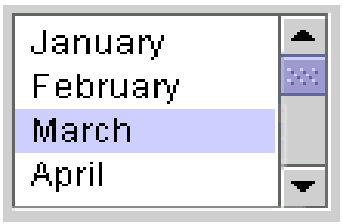


JSlider

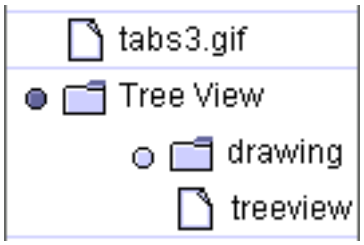
JScrollPane



JToolTip



JList



JTree

JTable

Entrées/sorties

Outre les filtres spécialisés qui permettent de manipuler des données primitives (`DataInput`, `DataOutput`) un aspect très important des E/S est la capacité de lire/écrire des objets sur un flot au moyen des classes `ObjectInputStream` et `ObjectOutputStream`.

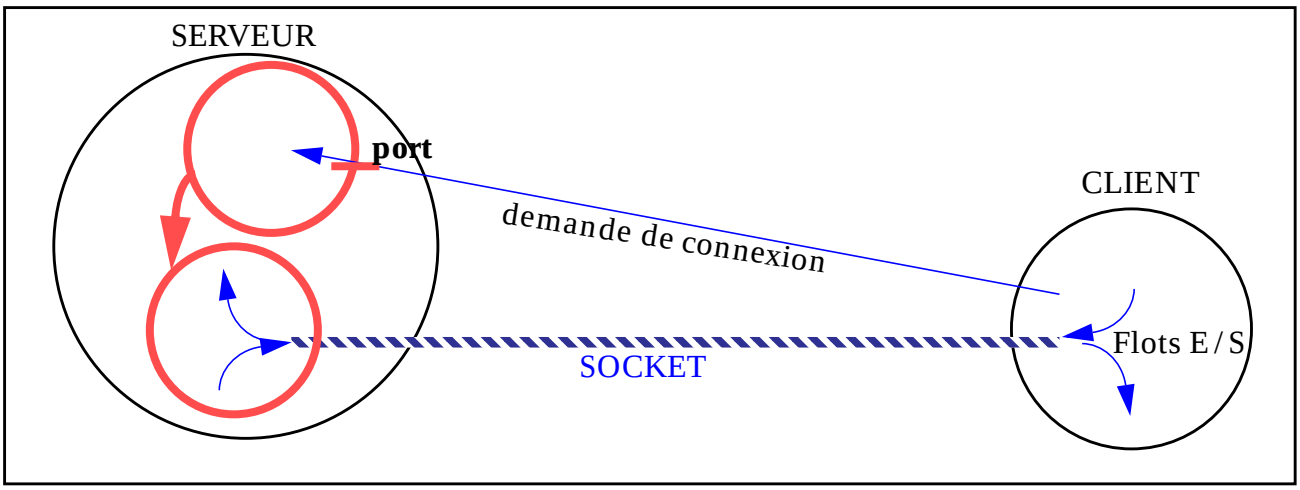
Les objets transférés doivent déclarer la propriété “linéarisable” (`implements Serializable`).

Les mécanismes standard permettent de contrôler la manière dont les objets sont transférés (non-transfert de certaines variables membres, détection des envois multiples de la même instance,...)

Il est également possible de personnaliser la manière dont les instances sont envoyées et reconstituées.

java.net

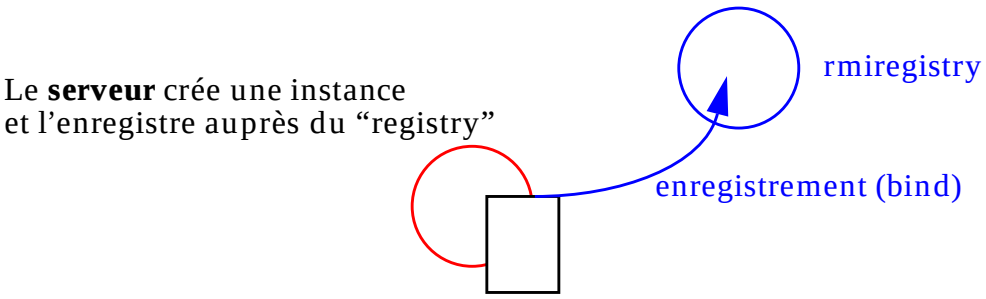
La librairie permet de réaliser des échanges entre des sites distant sur le réseau.

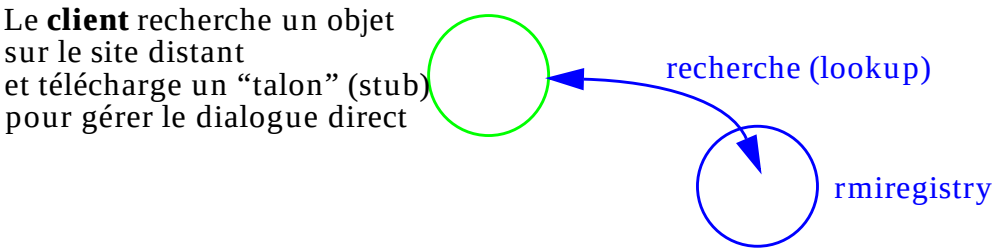


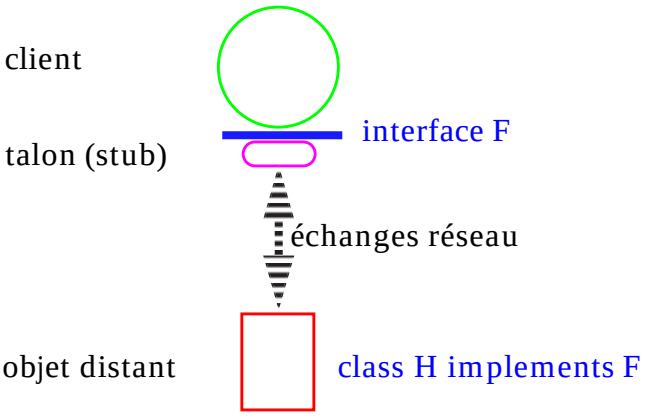
R.M.I

Mécanismes d'appel de méthodes sur des objets distants. Permettent de définir et de mettre en oeuvre des échanges client/serveur entre des programmes Java.

Coté serveur un programme *rmiregistry* sert d'intermédiaire pour permettre au client de rechercher un objet, de télécharger un protocole de communication avec un objet, etc.





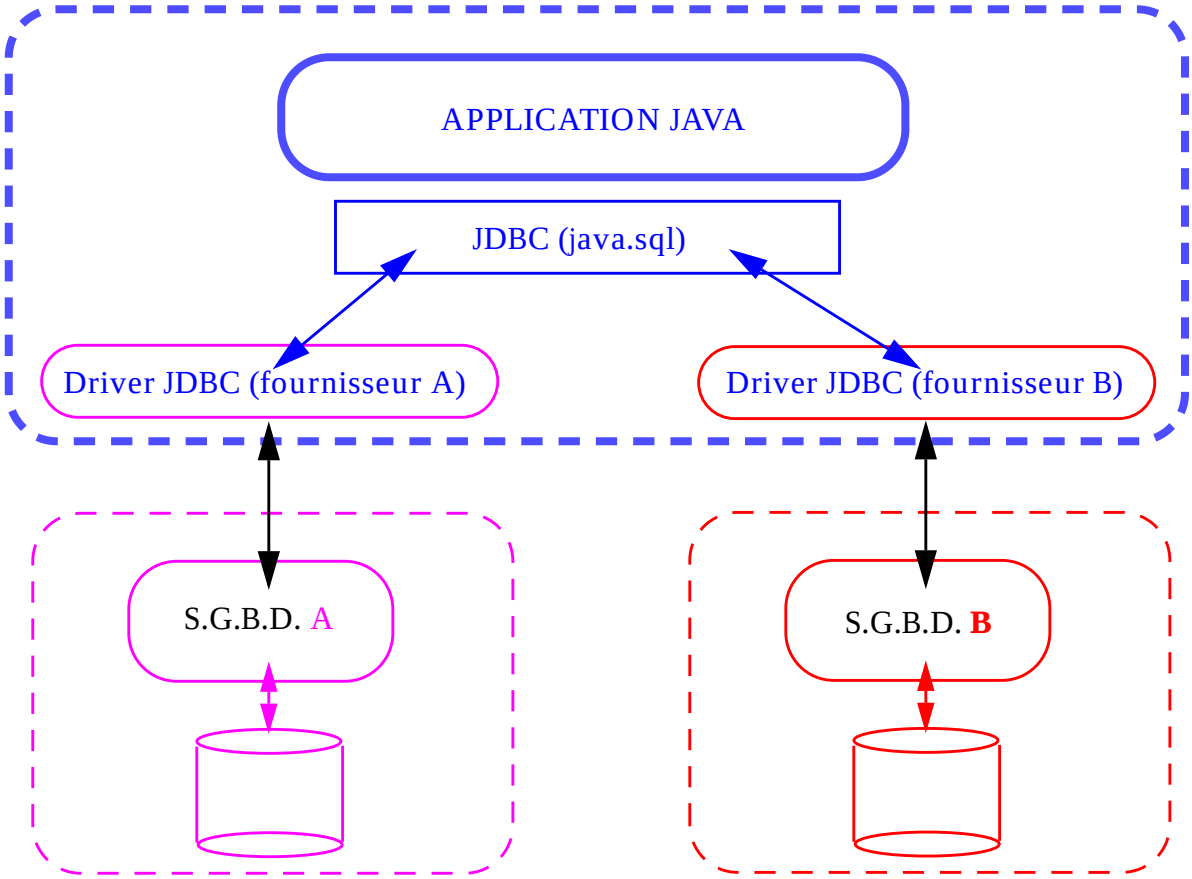


Un autre "démon" (processus système de veille) nommé *rmid* permet d'activer à la demande des objets serveurs qui peuvent être "dormants" (non présents au sein d'une JVM active).

J.D.B.C

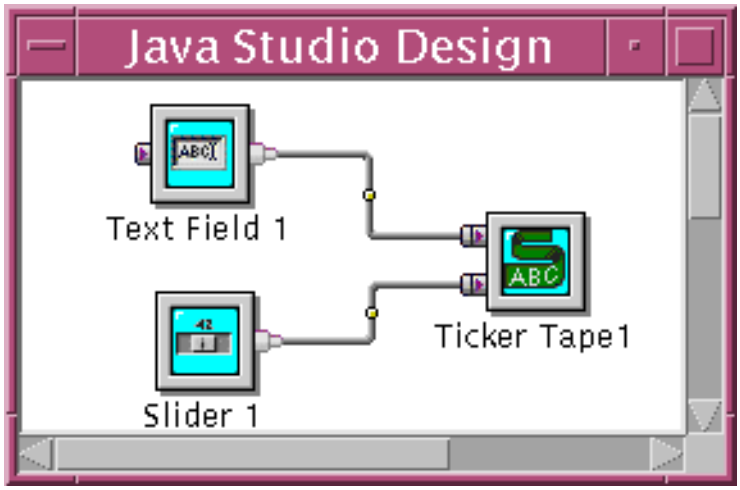
Le package java.sql

Définit essentiellement des interfaces d'accès à des bases de données SQL. Des produits spécifiques (hors JDK) sont chargés d'implanter ces interfaces (drivers JDBC).



Beans

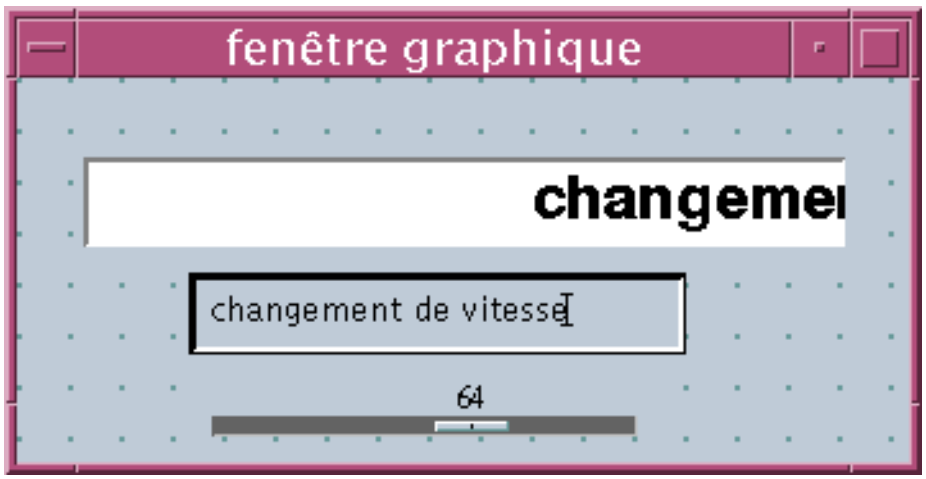
Les “JavaBeans” sont des composants réutilisables destinés à être assemblés dynamiquement (par exemple par un outil interactif “bean builder”).



JAVA STUDIO
un ex. de “bean builder”

assemblage logique des beans

assemblage résultant



La spécification JavaBean permet d’écrire des classes Java selon des conventions qui permettent la “découverte” de certaines capacités (production/consommation de données, etc.).

Un environnement de manipulation de Beans permet donc d'inspecter les capacités d'un classe (*introspection*), de personnaliser un composant en jouant sur ses paramètres, d'assembler des composants en les faisant communiquer aux moyen d'événements, de sauvegarder et de ranimer des instances, et de créer donc de nouveaux composants à partir de composants existants.

Le package **java.beans.beancontext** permet de définir des "containers" qui constituent un environnement d'exécution pour des beans.

Annexe: les composants AWT 15

Contenu

Cette annexe donne un aperçu sur les composants AWT courants et sur leur manipulation.

Button

C’est un composant d’interface utilisateur de base de type "appuyer pour activer". Il peut être construit avec un étiquetage texte précisant son rôle .

```
Button bt = new Button(“Sample”);
bt.addActionListener(....);
```

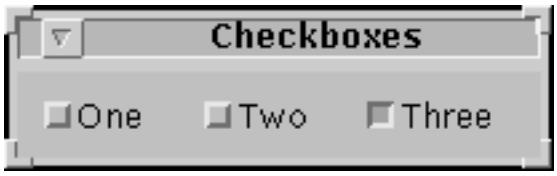
L’interface `ActionListener` doit pouvoir traiter un clic d’un bouton de souris. La méthode `getActionCommand()` de l’événement action (`ActionEvent`) activé lorsqu’on appuie sur le bouton rend par défaut la chaîne d’étiquetage (pour une meilleure internationalisation récupérer plutôt une chaîne positionnée par `setActionCommand()`)

Pour des besoins de repérage programmatique une chaîne symbolique peut-être rattachée à tout composant pour l’identifier: utiliser `setName()` et `getName()`.

Checkbox

La case à cocher fournit un dispositif d'entrée "actif/inactif" accompagné d'une étiquette texte.

```
Checkbox one = new Checkbox("One", false);
Checkbox two = new Checkbox("Two", false);
Checkbox three = new Checkbox("Three", true);
one.addItemListener(new Handler());
two.addItemListener(new Handler());
three.addItemListener(new Handler());
```



La sélection ou désélection d'une case à cocher est notifiée à un objet rélaisant l'interface `ItemListener`. Pour détecter une opération de sélection ou de déselection, il faut utiliser la méthode `getStateChange()` sur l'objet `ItemEvent`. Cette méthode renvoie l'une des constantes ItemEvent.`DESELECTED` ou `ItemEvent.SELECTED`, selon le cas. La méthode `getItem()` renvoie un objet de type chaîne (`String`) qui représente la chaîne de l'étiquette de la case à cocher considérée.

```
class Handler implements ItemListener {
        public void itemStateChanged(ItemEvent ev) {
                String state = "deselected";
                if (ev.getStateChange() == ItemEvent.SELECTED){
                        state = "selected";
                }
                System.out.println(ev.getItem() + " " + state);
        }
}
```

CheckboxGroup

On peut créer des cases à cocher à l'aide d'un constructeur spécial qui utilise un argument supplémentaire de type CheckboxGroup. Si on procéde ainsi, l'aspect des cases à cocher est modifié et toutes les cases à cocher liées au même groupe adoptent un comportement de "bouton radio".

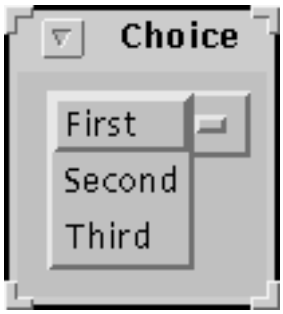
```
CheckboxGroup cbg = new CheckboxGroup();
Checkbox one = new Checkbox("One", cbg, false);
Checkbox two = new Checkbox("Two", cbg, false);
Checkbox three = new Checkbox("Three", cbg, true);
```

Choice

Le composant `Choice` fournit une outil simple de saisie de type "sélectionner un élément dans cette liste".

```
 Choice c = new Choice();
c.addItem("First");
c.addItem("Second");
c.addItem("Third");
c.addItemListener(. . .);
```

Lorsqu'un composant `Choice` est activé il affiche la liste des éléments qui lui ont été ajoutés. Notez que les éléments ajoutés sont des objets de type chaîne (`String`).



L'interface `ItemListener` sert à observer les modifications de ce choix. Les détails sont les mêmes que pour la case à cocher. La méthode `getSelectedIndex()` de Choice permet de connaître l'index selectionné.

List

Une liste permet de présenter à l'utilisateur des options de texte affichées dans une zone où plusieurs éléments peuvent être visualisés simultanément. Il est possible de naviguer dans la liste et d'y sélectionner un ou plusieurs éléments simultanément (mode de sélection simple ou multiple).

```
List l = new List(4, true);
```

L'argument numérique transmis au constructeur définit le nombre d'items visibles. L'argument booléen indique si la liste doit permettre à l'utilisateur d'effectuer des sélections multiples.

Un `ActionEvent`, géré par l'intermédiaire de l'interface `ActionListener`, est généré par la liste dans les modes de sélection simple et multiple. Les éléments sont sélectionnés dans la liste conformément aux conventions de la plate-forme. Pour un environnement Unix / Motif, cela signifie qu'un simple clic met en valeur une entrée dans la liste, mais qu'un double-clic déclenche l'action correspondante.

Récupération: voir méthodes `getSelectedObjects()`, `getSelectedItems()`

Label

Un label affiche une seule ligne de texte. Le programme peut modifier le texte. Aucune bordure ou autre décoration particulière n'est utilisée pour délimiter un label.

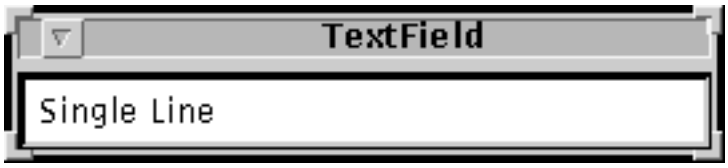
```
Label lab = new Label("Hello");
```

En général, on ne s'attend pas à ce que les `Labels` traitent des événements, pourtant on peut effectuer cette opération de la même façon que pour un `Canvas` (ou une classe utlisateur dérivée de Component). Dans ce cas, on ne peut capter les événements clavier de façon fiable qu'en faisant appel à `requestFocus()`.

TextField

Le `TextField` est un dispositif de saisie de texte sur une seule ligne.

```
TextField tf = new TextField(“Single line”, 30);
```



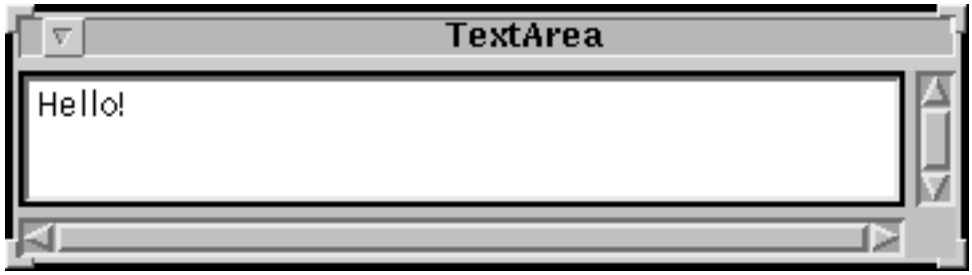
Du fait qu’une seule ligne soit possible, un veilleur d’événement Action (`ActionListener`) peut être informé, lorsque la touche <Entrée> ou <Retour> est activée.

Le champ texte peut être en lecture seule. Certains constructeurs prennent en argument un nombre de caractères visibles. Un TextField n’affiche pas de barres de défilement dans l’une ou l’autre direction mais permet, si besoin est, un défilement de gauche à droite d’un texte trop long.

TextArea

Le `TextArea` est un dispositif de saisie de texte multi-lignes, multi-colonnes. On peut le rendre non éditable par l'intermédiaire de la méthode `setEditable(boolean)`. Il affiche des barres de défilement horizontales et verticales.

```
TextArea tx = new TextArea("Hello!", 4, 30);
;
```



On peut ajouter des veilleurs d'événements de divers type sur un `TextArea`.

Le texte étant multi-lignes, le fait d'appuyer sur <Entrée> place seulement un autre caractère dans la mémoire tampon. Si on a besoin de savoir à quel moment une saisie est terminée, on peut placer un bouton de validation à côté d'un TextArea pour permettre à l'utilisateur de fournir cette information.

Un veilleur `KeyListener` permet de traiter chaque caractère entré en association avec la méthode `getKeyChar()`, `getKeyCode()` de la classe `KeyEvent`.

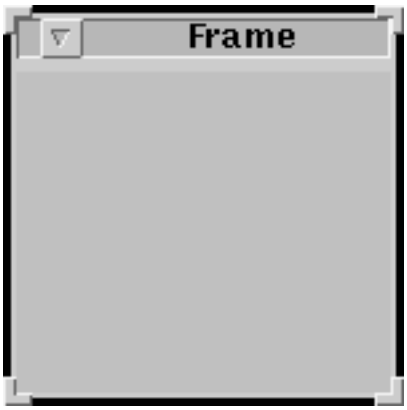
TextComponent

La classe `TextComponent` dont dérivent `TextField` et `TextArea` fourni un grand nombre de méthodes.On a vu que les constructeurs des classes `TextArea` et `TextField` permettent de définir un nombre de colonnes pour l'affichage. Le nombre de colonnes est interprété en fonction de la largeur moyenne des caractères dans la police utilisée. Le nombre de caractères effectivement affichés peut varier radicalement en cas d'utilisation d'une police à chasse proportionnellel.

Frame

C’est la fenêtre générale de "plus haut niveau". Elle possède des attributs tels que : barre de titre et zones de contrôle du redimensionnement.

```
Frame fr = new Frame(“Frame”);
```



La taille d’un `Frame` peut être définie à l’aide de la méthode `setSize()` ou avec la méthode `pack()`. Dans ce cas le gestionnaire de disposition calcule une taille englobant tous les composants du `Frame` et définit la taille de ce dernier en conséquence.

Les événements du Frame peuvent être surveillés à l’aide de tous les gestionnaires d’événements applicables aux composants généraux. `WindowListener` peut être utilisé pour réagir, via la méthode `windowClosing()`, lorsque le bouton Quit a été activé dans le menu du gestionnaire de fenêtres.

Il n’est pas conseillé d’écouter des événements clavier directement à partir d’un `Frame`. Bien que la technique décrite pour les composants de type `Canvas` et `Label`, à savoir l’appel de `requestFocus()`, fonctionne parfois, elle n’est pas fiable. Si on a besoin de suivre des événements clavier, il est plutôt recommandé d’ajouter au `Frame` un `Canvas`, `Panel`, etc., et d’associer le gestionnaire d’événement à ce dernier.

Dialog

Un `Dialog` est une fenêtre (qui, comme `Frame`, hérite de `Window`) elle diffère toutefois d'un `Frame` :

- elle est destinée à afficher des messsages
- elle peut être modale: elle recevra systématiquement toutes les saisies jusqu'à fermeture.


```
Dialog dlg = new Dialog(fen, "Dialog", false);
dlg.add(new Label("Hello, I'm a Dialog"),
                        BorderLayout.CENTER);
dlg.pack();
```

Un dialog dépend d'une Frame : cette Frame apparait comme premier argument dans les constructeurs de la classe Dialog.

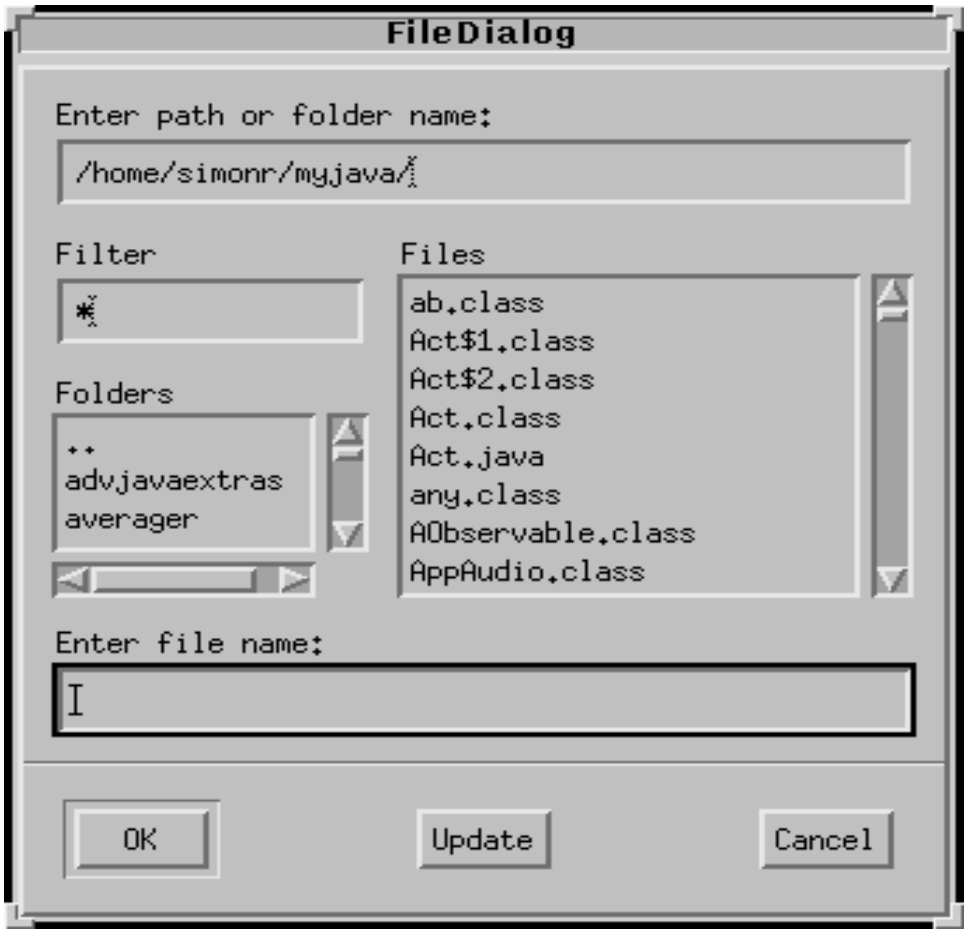
Les boîtes de dialogue ne sont pas visibles lors de leur création (utiliser `setVisible(true)`). Elles s'affichent plutôt en réponse à une autre action au sein de l'interface utilisateur, comme le fait d'appuyer sur un bouton.

Il est recommandé de considérer une boîte de dialogue comme un dispositif réutilisable. Ainsi, vous ne devez pas détruire l'objet individuel lorsqu'il est effacé de l'écran, mais le conserver pour une réutilisation ultérieure.

FileDialog

C'est une implantation d'un dispositif de sélection de fichier. Elle comporte sa propre fenêtre autonome et permet à l'utilisateur de parcourir le système de fichiers et de sélectionner un fichier spécifique pour des opérations ultérieures.

```
FileDialog d = new FileDialog(f, "FileDialog");
d.setVisible(true);
String fname = d.getFile();
```



En général, il n'est pas nécessaire de gérer des événements à partir de la boîte de dialogue de fichiers. L'appel de `setVisible(true)` se bloque jusqu'à ce que l'utilisateur sélectionne OK. Le fichier sélectionné est renvoyé sous forme de chaîne .Voir `getDirectory(), getFile()`

Panel

C’est le conteneur de base. Il ne peut pas être utilisé de façon isolée comme les Frames, les fenêtres et les boîtes de dialogue.

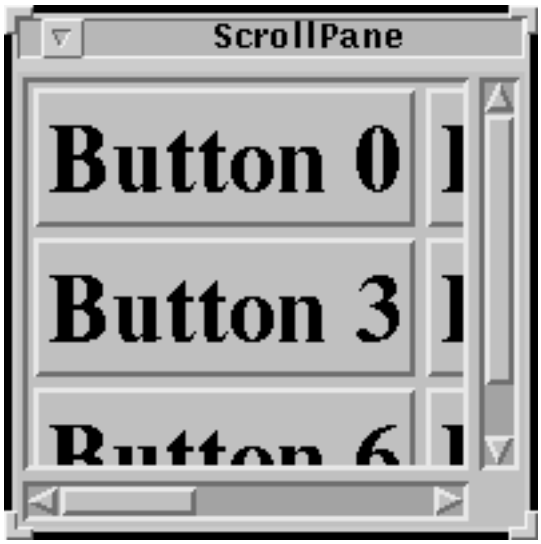
```
Panel p = new Panel();
```

Les Panels peuvent gérer les événements (rappel : le focus clavier doit être demandé explicitement).

ScrollPane

Fournit un conteneur général ne pouvant pas être utilisé de façon autonome. Il fournit une vue sur une zone plus large et des barres de défilement pour manipuler cette vue.

```
Frame fr = new Frame("ScrollPane");
Panel pan = new Panel();
ScrollPane sp = new ScrollPane();
pan.setLayout(new GridLayout(3, 4));
....
sp.add(pan);
fen.add(sp);
fen.setSize(200, 200);
fen.setVisible(true);
```



Le `ScrollPane` crée et gère les barres de défilement selon les besoins. Il contient un seul composant et on ne peut pas influer sur le gestionnaire de disposition qu'il utilise. Au lieu de cela, on doit lui ajouter un Panel, configurer le gestionnaire de disposition de ce Panel et placer les composants à l'intérieur de ce dernier.

En général, on ne gère pas d'événements dans un `ScrollPane`, mais on le fait dans les composants qu'il contient.

Canvas

Un canvas fournit un espace vide (arrière-plan coloré). Sa taille par défaut étant zéro par zéro, on doit généralement s'assurer que le gestionnaire de disposition lui affectera une taille non nulle.

Cet espace peut être utilisé pour dessiner, recevoir du texte ou des saisies en provenance du clavier ou de la souris.

Le canvas est généralement utilisé tel quel pour fournir un espace de dessin général.

Le canvas peut écouter tous les événements applicables à un composant général. On peut, en particulier, lui associer des objets `KeyListener`, `MouseMotionListener` ou `MouseListener` pour lui permettre de répondre d'une façon ou d'une autre à une interaction utilisateur.

Pour réaliser un tel espace "libre" on peut aussi créer un composant dérivé de Component ou de Container: dans ce cas le "fond" sera transparent. Pour donner une taille à un tel composant on définira ses méthodes `get***Size()` .

Menus

Les menus diffèrent des autres composants par un aspect essentiel. En général, on ne peut pas ajouter de menus à des conteneurs ordinaires et laisser le gestionnaire de disposition les gérer. On peut seulement ajouter des menus à des éléments spécifiques appelés conteneurs de menus. Généralement, on ne peut démarrer une "arborescence de menu" qu'en plaçant une barre de menus dans un Frame via la méthode `setMenuBar()`. A partir de là, on peut ajouter des menus à la barre de menus et incorporer des menus ou éléments de menu à ces menus.

L'exception est le menu `PopUpMenu` qui peut être ajouté à n'importe quel composant, mais dans ce cas précis, il n'est pas question de disposition à proprement parler.

Menu Aide

Une caractéristique particulière de la barre de menus est que l'on peut désigner un menu comme le menu Aide. Cette opération s'effectue par l'intermédiaire de la méthode `setHelpMenu(Menu)`. Le menu à considérer comme le menu Aide doit avoir été ajouté à la barre de menus, et il sera ensuite traité de la façon appropriée pour un menu Aide sur la plate-forme locale. Pour les systèmes de type X/Motif, cela consiste à décaler l'entrée de menu à l'extrémité droite de la barre de menus.

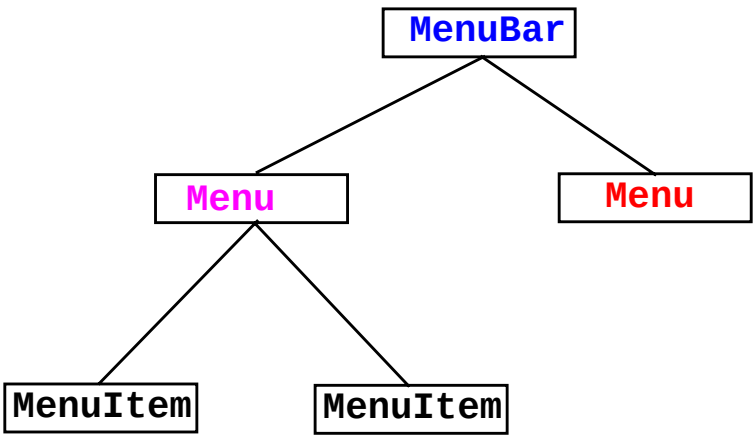
MenuBar

C'est la barre de menu horizontale. Elle peut seulement être ajouté à l'objet `Frame` et constitue la racine de toutes les arborescences de menus.

```
Frame fr = new Frame("MenuBar");
MenuBar mb = new MenuBar();
fr.setMenuBar(mb);
```

On n'abonne pas de veilleur d'événements à une `MenuBar` tout est automatique à ce niveau.

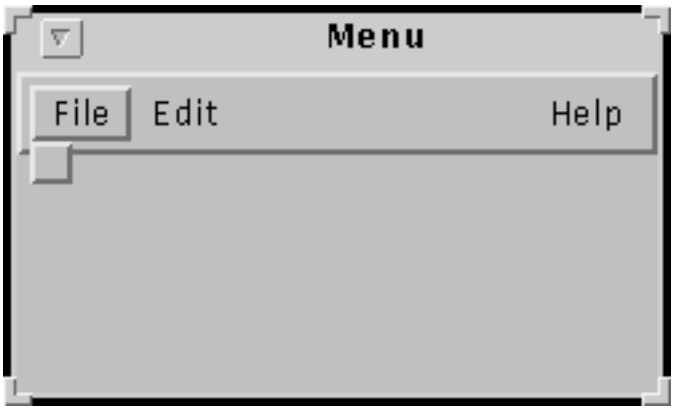
Hiérarchie d'accrochage des éléments de menu:



Menu

La classe `Menu` fournit le menu déroulant de base. Elle peut être ajoutée à une barre de menus ou à un autre menu.

```
MenuBar mb = new MenuBar();
Menu m1 = new Menu("File");
Menu m2 = new Menu("Edit");
Menu m3 = new Menu("Help");
mb.add(m1);
mb.add(m2);
mb.add(m3);
mb.setHelpMenu(m3);
```



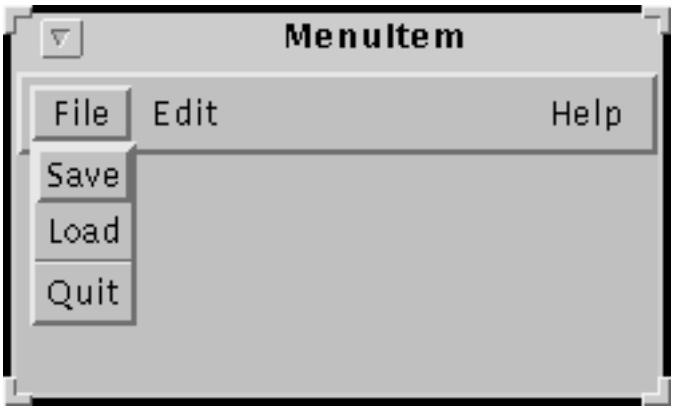
Les menus présentés ici sont vides ce qui explique l'aspect du menu File.

On peut ajouter un *`ActionListener`* à un objet Menu, mais c'est assez inhabituel. Normalement, les menus servent seulement à disposer des `MenuItem` décrits plus loin.

MenuItem

Les éléments de menu `MenuItem` sont les “feuilles” d’une arborescence de menu.

```
Menu m1 = new Menu("File");
MenuItem mi1 = new MenuItem("Save");
MenuItem mi2 = new MenuItem("Load");
MenuItem mi3 = new MenuItem("Quit");
m1.add(mi1);
m1.add(mi2);
m1.addSeparator();
m1.add(mi3);
```

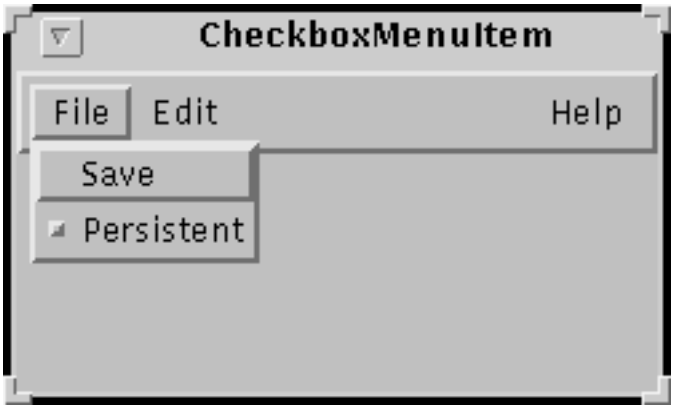


En règle générale, on associe un `ActionListener` aux objets `MenuItem` afin d’attribuer des comportements aux éléments de menu.

CheckboxMenuItem

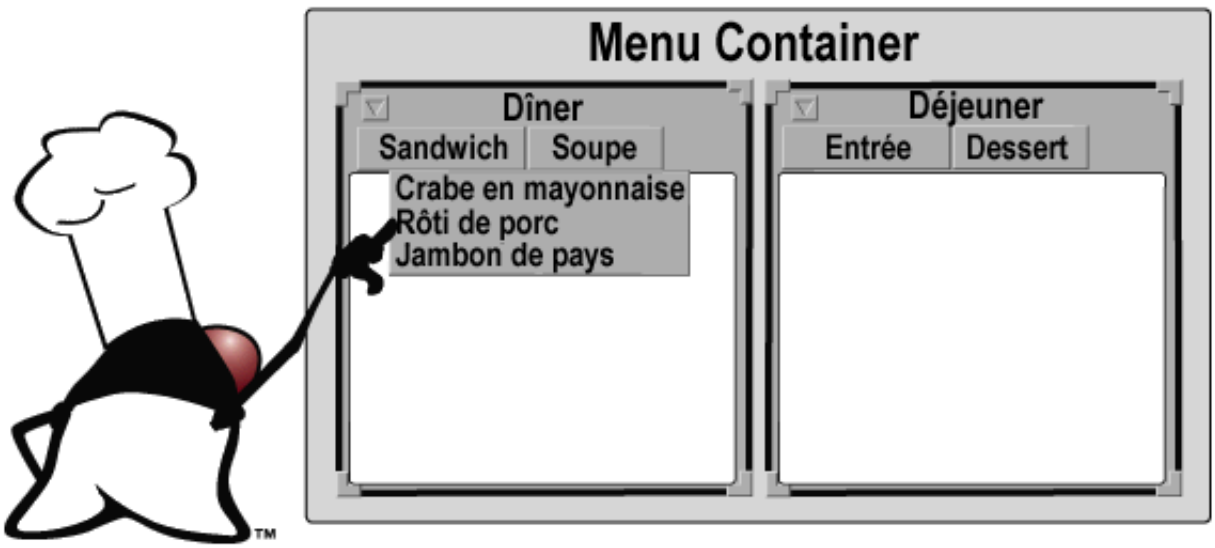
Les éléments de menu à cocher permettent de proposer des sélections (activé / désactivé) dans les menus.

```
Menu m1 = new Menu("File");
MenuItem mi1 = new MenuItem("Save");
CheckboxMenuItem mi2 =
          new CheckboxMenuItem("Persistent");
m1.add(mi1);
m1.add(mi2);
```



L'élément de menu à cocher doit être surveillé via l'interface ItemListener. La méthode `itemStateChanged()` est appelée lorsque l'état de l'élément à cocher est modifié.

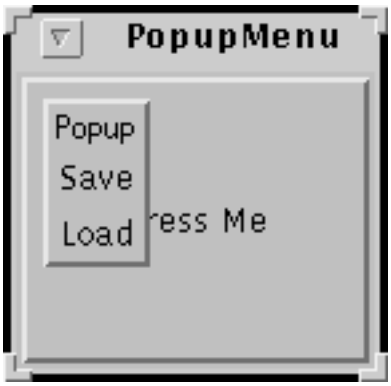
Menus


Menu Container
Dîner
Sandwich
Soupe
Crabe en mayonnaise
Rôti de porc
Jambon de pays
Déjeuner
Entrée
Dessert

PopupMenu

Fournit un menu autonome pouvant s'afficher instantanément sur un autre composant. On peut ajouter des menus ou éléments de menu à un menu instantané.

```
Frame fr = new Frame("PopupMenu");
Button bt = new Button("Press Me");
bt.addActionListener(...);

PopupMenu pp = new PopupMenu("Popup");
MenuItem sit = new MenuItem("Save");
MenuItem lit = new MenuItem("Load");

sit.addActionListener(...);
lit.addActionListener(...);

fr.add("Center", bt);
pp.add(sit);
pp.add(lit);
fr.add(pp);
```



PopupMenu (suite)

Le menu PopUp doit être ajouté à un composant "parent". Cette opération diffère de l'ajout de composants ordinaires à des conteneurs. Dans l'exemple suivant, le menu instantané a été ajouté au Frame englobant.

Pour provoquer l'affichage du menu instantané, on doit appeler la méthode show. L'affichage nécessite qu'une référence à un composant joue le rôle d'origine pour les coordonnées x et y.

Dans cet exemple c'est le composant bt qui sert de référence.

```
public void actionPerformed(ActionEvent ev) {
          pp.show(bt, 10, 10);
}
```

Composant de référence pour l'affichage du popup menu

Coordonnées % composantes d'origines

Le composant d'origine doit être sous (ou contenu dans) le composant parent dans la hiérarchie des composants.

Contrôle des aspects visuels

On peut contrôler l'apparence des composants AWT en matière de couleur de fond et de premier plan ainsi que de police utilisée pour le texte.

Couleurs

Deux méthodes permettent de définir les couleurs d'un composant :

- setForeground(...)
- setBackground(...)

Ces deux méthodes utilisent un argument qui est une instance de la classe `java.awt.Color`. On peut utiliser des couleurs de constante désignées par `Color.red` `Color.blue` etc. La gamme complète de couleurs prédéfinies est documentée dans la page relative à la classe `Color`.

Qui plus est, on peut créer une couleur spécifique de la façon suivante :

```
int r = 255, g = 255, b = 0;
Color c = new Color(r, g, b);
```

Un tel constructeur crée une couleur d'après les intensités de rouge, vert et bleu spécifiées sur une échelle allant de 0 à 255 pour chacune.

Polices

La police utilisée pour afficher du texte dans un composant peut être définie à l'aide de la méthode `setFont()`. L'argument utilisé pour cette méthode doit être une instance de la classe `java.awt.Font`.

Aucune constante n'est définie pour les polices, mais on peut créer une police en indiquant son nom, son style et sa taille en points.

```
Font f = new Font("TimesRoman", Font.PLAIN, 14);
```

Si la portabilité est recherchée il vaut mieux utiliser des noms de polices abstraites :

- Dialog, DialogInput
- SansSerif (remplace Helvetica)
- Serif (remplace TimesRoman)
- Monospaced (remplace Courier)
- Symbol

On peut obtenir la liste complète des polices en appelant la méthode `getFontList()` de la classe `Toolkit`. La boîte à outils (toolkit) peut être obtenue à partir du composant, une fois ce dernier affiché on appelle la méthode `getToolkit()`. On peut aussi utiliser le ToolKit par défaut obtenu par `Toolkit.getDefaultToolkit()`.

Les constantes de style de police sont en réalité des valeurs entières (int), parmi celles citées ci-après :

- `Font.BOLD`
- `Font.ITALIC`
- `Font.PLAIN`
- `Font.BOLD + Font.ITALIC`

Les tailles en points doivent être définies avec une valeur entière.

Impression

L'impression de documents est très liée à la représentation graphique.

Pour une documentation à jour voir l'URL: http://java.sun.com/printing et, en particulier, http://java.sun.com/printing/jdk1.2/printing-chapter-v2.txt

INDEX

Numerics

A

B

C

D

E

F

G

H

I

J

K

L

M

N

O

P

R

S

T

U

V

W

Z

Sun Microsystems France

La couverture des agences de Sun France permet de répondre à l'ensemble des besoins de nos clients sur le territoire.

Table 15.1 Liste des agences Sun Microsystems en france

Sun Microsystems France S.A
13, avenue Morane Saulnier
BP 53
78142 VELIZY Cedex
Tél : 01.30.67.50.00
Fax : 01.30.67.53.00

Agence de Issy les Moulineaux
Le Lombard
143, avenue de Verdun
92442 ISSY-LES-MOULINEAUX Cedex
Tél : 01.41.33.17.00
Fax : 01.41.33.17.20

Agence de Lille
Tour Crédit Lyonnais
140 Boulevard de Turin
59777 EURALILLE
Tél : 03.20.74.79.79
Fax : 03.20.74.79.80

Agence de Rennes
Immeuble Atalis
Z.A. du Vieux Pont
1, rue de Paris
35510 CESSON-SEVIGNE
Tél : 02.99.83.46.46
Fax : 02.99.83.42.22

Bureau de Grenoble
32, chemin du Vieux Chêne
38240 MEYLAN
Tél : 04.76.41.42.43
Fax : 04.76.41.42.41

Agence d'Aix-en-Provence
Parc Club du Golf
Avenue G. de La Lauzière
Zone Industrielle - Bât 22
13856 AIX-EN-PROVENCE
Tél : 04.42.97.77.77
Fax : 04.42.39.71.52

Agence de Lyon
Immeuble Lips
151, boulevard de Stalingrad
69100 VILLEURBANNE
Tél : 04.72.43.53.53
Fax : 04.72.43.53.40

Agence de Toulouse
Immeuble Les Triades
Bâtiment C - B.P. 456
31315 LABEGE Cedex
Tél : 05.61.39.80.05
Fax : 05.61.39.83.43

Agence de Strasbourg
Parc des Tanneries
1, allée des Rossignols
Bâtiment F - B.P. 20
67831 TANNERIES Cedex
Tél : 03.88.10.47.00
Fax : 03.88.76.53.63