Programmation avec le langage Java

Norbert KAJLER, Fabien MOUTARDE

CCSI

Mines ParisTech

Norbert.Kajler@mines-paristech.fr ; Fabien.Moutarde@mines-paristech.fr

Plan

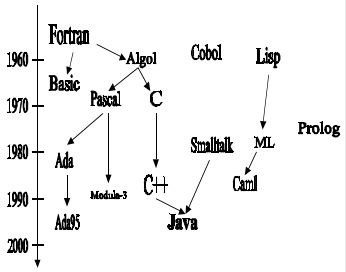
INTRODUCTION

Avertissement :

Ceci n'est PAS un polycopié, mais une simple copie de transparents, prévue pour faciliter le suivi des cours et la prise de notes manuscrites complémentaires.

Les langages de programmation

- Niveaux et catégories de langages :
  - langage binaire
  - assembleur : instructions de base du processeur (transferts entre registres, addition, …)
  - langages impératifs
    Basic, Fortran, Pascal, C, ...
  - langages fonctionnels
    Lisp, Caml, ...
  - langages orientés-objet
    SmallTalk, C++, **Java**,...



Historique de Java

- Initialement, projet de la société Sun (rachetée depuis par Oracle) pour l'électronique grand public (1991)
- Transformé en langage pour le Web, sous le nom de « Java », grâce à sa portabilité (1994/95)
- Lancement officiel en mai 1995
- Après l'engouement pour les applets, Java est progressivement reconnu comme un langage à part entière
- Langage de programmation sans doute le plus utilisé aujourd'hui :
  - plusieurs millions de développeurs Java
  - nombreux outils de développement
  - 6 milliards d'objets avec une « machine virtuelle Java », dont 85% des téléphones portables et 91% des ordinateurs (chiffres 2008)

Historique de Java (2)

- 1996 : Java 1.0
- 1997 : Java 1.1
  - Modification du modèle des événements pour AWT, JDBC, RMI, ...
- 1998 : Java 1.2 (renommé Java2 version 1.2 ou J2SE 1.2)
  - Collections, Swing, Java2D, …
- 2000 : v.1.3 (ou « Kestrel » ou J2SE 1.3)
- 2002 : v.1.4 (ou « Merlin » ou J2SE 1.3)
- 2004 : v.1.5 (ou « Tiger » ou J2SE 5.0 !!)
  - généricité (proche des templates de C++), types énumérés, autoboxing/unboxing des types primitifs dans les collections, web services, ...
- 2006 : v.1.6 (ou « Mustang » ou Java SE 6)
  - améliorations des performances (Swing notamment), interactions avec scripts (PHP, Python, Ruby, JavaScript), Java DB…
- 2011 : v.1.7 (ou « Dolphin » ou Java SE 7)
  - NIO (new I/O), try-with, améliorations de la généricité et des instructions switch, catch, …

Intérêt de Java

- **logiciels portables**
- **programmes fiables**
  (rigueur du langage => peu de bogues)
- **développement rapide**
- pages Web interactives (via les « applets »)
- logiciels (ou briques logicielles) téléchargeables, éventuellement automatiquement
- gestion de la sécurité (par défaut, accès restreint aux ressources locales pour les applets)

Caractéristiques de Java

- Un langage orienté-objet :
  - portable
  - compilé puis interprété (bytecode+JVM)
  - robuste (typage fort, pas de pointeurs, garbage collector)
  - modulaire (packages)
  - intégrant le multi-threading
- une énorme librairie de classes standards

Java et les autres langages

- Java est très proche de C++ (syntaxe de base identique à C et C++)
- Simplifications de Java (par rapport à C++) :
  - pas de manipulation de pointeurs sous forme d'adresse mémoire, gestion mémoire automatique (garbage collector)
  - pas de surcharge des opérateurs
  - pas d'héritage ***multiple***
  - pas de préprocesseur
- Principaux ajouts (par rapport à C++) :
  - tableaux avec test de dépassement de bornes
  - chaînes de caractères sous forme de classe
  - notion d'interface
  - classe racine '`Object`', introspection
  - structuration en paquetages
  - multi-threading incorporé
  - vaste bibliothèque de classes

JRE, Java SE, JDK, Eclipse, …

- Plusieurs implantations disponibles du langage Java, dont celles d'origine ORACLE/SUN (créateur du langage)
- Pour exécuter du code Java : le « **JRE** » (Java Runtime Environment) d'ORACLE/SUN suffit (gratuit et largement disponible)
- Pour créer/compiler du code java : le « **Java SE** » (Standard Edition) dénommée aussi « **JDK** » (Java Development Kit) d'ORACLE/SUN (gratuit) contient :
  - compilateur (`javac`)
  - interpréteur / machine virtuelle (`java`)
  - toute la librairie de classes standards
  - divers outils : génération doc (`javadoc`), visualisation d'applet (`appletviewer`), debugger (`jdb`), ...
- Nombreux outils de développement dont la plateforme « **Eclipse** » (gratuite)
- Les sources de Java sont disponibles gratuitement (code de toutes les classes prédéfinies)

SYNTAXE DE BASE

Premiers exemples

- ***Quelques lignes de code Java :***

```
int i, somme;
somme = 0;
for (i=1; i<=9; i++)
  somme = somme+i;
```

déclarations

affectation

boucle

- ***Un programme en Java :***

```
class Prog {
  public static void main(String [] args){
    int i, somme=0;
    for (i=1; i<=9; i++) {
      somme += i;
    }
    System.out.println(somme);
  }
}
```

structure essentielle : classe

Programme principal

VARIABLES, TYPES, PORTEE, COMMENTAIRES, …

Variables et types

- notion de **variable** :

  nom (identificateur) + type + zone mémoire

- en Java, deux grandes catégories de types :
  - types **« primitifs »** (entiers, flottants, …)
  - **références** à des types « composites » :
    - tableaux
    - énumérations
    - objets
    - interfaces

Types « primitifs »

- **`boolean`**

- **`char` (16-bit, Unicode)**

- `byte` : entier (signé) 8-bit
- `short` : entier (signé) 16-bit
- **`int` : entier (signé) 32-bit**
- `long` : entier (signé) 64-bit

- `float` : flottant (IEEE 754) 32-bit
- **`double` : flottant (IEEE 754) 64-bit**

Booléens : `boolean`

- 2 valeurs : `true` ou `false`
- véritable type
- type retourné par les opérateurs de comparaison
- type attendu dans tous les tests
- ne peut *pas* être converti en entier

Entiers

- littéraux de type entier :
  - en base dix : `139`
  - en octal : `0213`
  - en hexadécimal : `0x8b`
- L ou l pour spécifier un long : `139L`
- valeurs min/max :
  - `byte` = [-128; +127]
  - `short` = [-32768 ; +32767]
  - `int` = [-2.147.483.648 ; +2.147.483.647]
  - `long` = [-9,223... 10^{18} ; +9,223... 10^{18}]
- conversion automatique seulement vers les types entiers plus grands (`int`–>`long`, etc...) et vers les types flottants

Caractères : `char`

- 16-bit ➔ 65536 valeurs : quasiment tous les caractères de toutes les écritures !
- ne sont affichables que si le système possède les polices de caractères adéquates !
- littéraux entre simples quotes : `'a' 'Z'`
- caractères spéciaux : `'\n'` `'\t'` `'\b'` `'\\'` ...
- possibilité d'utiliser la valeur Unicode : `'\u03c0'` pour la lettre **π**
- tests du type :
  `Character.isLetter(c)`, `Character.isDigit(c)`, …
  où `c` est une variable de type `char`
- convertible automatiquement en `int` ou `long` (et manuellement en `byte` ou `short`)
- inversement, `(char)val` est le caractère dont le code Unicode est l'entier `val`

Flottants

- notation « ingénieur » : `2.45e-25`
- littéraux de type `double` par défaut :

```
float x = 2.5;      // Erreur
double y = 2.5;     // OK
```

- valeurs `f` ou `F` pour spécifier un `float` : `float x = 2.5f;`
- valeurs min/max ***de valeur absolue*** (hors 0) :
  - `float` => [1.40239846e-45 ; 3.40282347e+38]
  - `double` => [4.9406...e-324 ; 1.7976...e+308]
- valeurs spéciales : `Infinity`, `-Infinity`, `NaN`
- conversion automatique : seulement `float` –> `double`
- la conversion « manuelle » en entier tronque la partie décimale :

```
float x = -2.8f;
int i = (int) x;    // => i==-2
```


Constantes

- variable dont la valeur ne peut plus être changée une fois fixée

- se déclare avec le mot-clé `final` :

```
final double PI = 3.14159;
```

ce qui interdit d'écrire ensuite…

```
PI = 3.14;    // ERREUR…
```

- possibilité de calculer la valeur de la constante plus tard à l'exécution, et ailleurs qu'au niveau de la déclaration :

```
final int MAX_VAL;
  // OK : constante « blanche »
  // ...
MAX_VAL = lireValeur();
```


Déclaration et portée des variables

- déclaration préalable obligatoire
- identificateurs :
  - caractères Unicode
  - commencent par une lettre, _ ou $
  - ensuite : lettre, chiffre, _ ou $
- exemples de déclaration :

```
int i;
float x, y, z;
char c = 'A', d;
boolean flag = true;
```

- initialisation obligatoire avant usage :

```
int i, j;
j = i;   // ERREUR : i non initialisé
```

- « portée » des variables (= zone de validité de la déclaration) : jusqu'à la fin du *bloc englobant* (voir plus loin)

Conventions de nommage

- pas obligatoires, mais très fortement recommandées (car important pour la *lisibilité* du code et sa *maintenance*)

- identificateurs en minuscules, sauf :
  - majuscule au début des « sous-mots » intérieurs :
    `unExempleSimple`
  - début des noms en majuscule pour classes et interfaces (et elles **seules**) :
    `UnNomDeClasse`

- pour les constantes, majuscules et séparation des mots par _

```
final int VALEUR_MAX;
```

- pas d'emploi de $ (et de caractères non ASCII)

Commentaires

- Sur une ligne (style C++) :
```
//Commentaire -> fin de la ligne
int x; // Ligne partielle
```
- Sur plusieurs lignes (style C) :
```
/* Commentaire qui s'étend
   sur plusieurs lignes */
```
- De documentation :
```
/** Commentaire pour javadoc */
```
- `javadoc` : outil qui analyse le code Java, et produit **automatiquement** une documentation HTML avec la liste des classes, leurs attributs, méthodes… (avec leurs éventuels « commentaires de documentation » respectifs)

Références

- servent à manipuler objets et tableaux
- à voir comme une sorte de « poignée »
- plusieurs références différentes peuvent référencer un même objet ou tableau
- les références sont typées
- une affectation modifie la référence, et non la chose référencée :

```
int t1[] = {1,2,3};
int t2[] = t1;
// t2 : 2° réf. au même tableau
t2 = new int[5];
// t1 reste réf. à {1,2,3}
t1 = null;
// => t1 ne pointe plus vers rien
```

```
int[] t1;
t1 = new int [3];
t1[0] = 1;
t1[1] = 2;
t1[2] = 3;
```

- `null` : valeur d'une référence « vers rien » (pour tous types de références)

Tableaux (1)

- manipulés par des références
- vérification des bornes à l'utilisation
- allocation *dynamique* par `new`
  (=> *taille définissable à l'exécution*)
- taille fixée une fois pour toute
- taille accessible par le « champ » `length`

Tableaux (2)

- **déclaration** :
```
int tab[];
int [] tab2;  // 2° forme autorisée
```
- **création** :
  - par `new` après la déclaration (ou lors de la déclaration) :
```
int tab[]; // déclaration
tab = new int[5]; // création
```
  - ou implicite en cas d'initialisation lors de la création (cf. ci-dessous)
- **initialisation** :
```
// A la déclaration :
int tab[] = {1, 2, 3};

// Lors de la création :
tab = new int[] {1, 2, 3};

// Sinon, cases du tableau initialisées par défaut
//   à 0 ou false ou null (selon le type de base)
```

Tableaux (3)

- accès à la taille :

```
int [] tab = new int[5];
int size = tab.length;
```

- indices : entre `0` et `length-1`
- si tentative d'accès hors des bornes, lancement de l'exception :

```
ArrayIndexOutOfBoundsException
```

- l'affectation ne modifie que la référence :

```
int t1[] = {1, 2, 3};
int t2[] = {4, 5};
t1 = t2;
// => t1 réfère même tableau que t2
```

- espace mémoire libéré automatiquement par le « ramasse-miette » (en anglais : "Garbage Collector") dès que plus désigné par aucune référence
  `ex. : l'espace occupé par {1,2,3} ci-dessus est libéré automatiquement après l'instruction t1=t2`
- recherche, tri, etc. : voir fonctions utilitaires de la classe `Arrays` (cf `java.util`)

Exemple de manipulation de tableau

```
public class ExempleTableau {
 public static void main(String[] args) {
   // DECLARATION
   double[] tab;

   //CREATION ET DIMENSIONNEMENT
   int lg = 1 + (int)( 9*Math.random() );
   tab = new double[lg];

   //AFFECTATION DES ELEMENTS DU TABLEAU
   for (int i=0 ; i<tab.length ; i++)
     tab[i] = 1./(i+1);

   // AFFICHAGE
   for (int i=0 ; i<tab.length ; i++)
     System.out.print(tab[i] + " ");
   System.out.println();
 }
}
```

Taille du tableau

Copie de tableau

- Méthode « manuelle » :
  création d'un nouveau tableau, puis copie case par case
- Copie par appel de `System.arraycopy()` :

```
// copie des 50 premiers éléments de src dans dst à partir de l'indice 2 :
System.arraycopy(src, 0, dst, 2, 50);
```

- Allocation et copie par `Arrays.copyOf()` ou `Arrays.copyOfRange()` :

```
int[] t2 = Arrays.copyOf(t, 10);    // copie les 10 premières cases
int[] t3 = Arrays.copyOfRange(t, 5, 80); // copie les cases 5,6..79
```

- Allocation et copie par clonage :

```
int[] copie = t.clone();
```

- **Attention :** pour un tableau d'*objets*, copie des références
  ➔ les mêmes objets sont ensuite partagés par les 2 tableaux
- Note : déplacement interne dans un tableau possible par `System.arraycopy()` :

```
System.arraycopy(t, 0, t, 1, t.length-1); // décale de 1 vers droite
```


Tableaux multi-dimension

- Tableaux de (références vers) tableaux :

```
// Déclaration et création :
float matrix[][]=new float[2][3];
// Utilisation :
for (int i=0 ; i<matrix.length ; i++)
  for (int j=0 ; j<matrix[i].length ; j++)
    matrix[i][j] = i*j;
```

- Possibilité de forme non rectangulaire :

```
float triangle[][];
int dim;
// ...
triangle = new float[dim][];
for (int i=1 ; i<dim ; i++) {
  triangle[i] = new float[i];
}
```

- Quelques (rares) fonctions à connaître sur les tableaux de tableaux :

```
Arrays.deepEquals();
Arrays.deepToString();
```

Tableaux d'objets

- Des tableaux contenant des (références à des) objets d'une classe donnée peuvent être déclarés et créés ***pour toute classe*** (prédéfinie ou bien écrite par le programmeur).

  Par exemple, un tableau de chaînes de caractères (classe `String` : voir plus loin) se déclare et se dimensionne ainsi :

```
String[] tabChaines;
tabChaines = new String[2];
```

- ATTENTION : le dimensionnement d'un tableau d'objets crée uniquement des « cases » pour stocker des ***références*** aux objets, mais *pas les objets* eux-mêmes. Ces références valent initialement `null`, et il faut donc les faire ensuite pointer vers les objets voulus par des affectations.

  Par exemple, une suite possible des instructions précédentes est :

```
tabChaines[0] = "OUI";
tabChaines[1] = "NON";
```


Références constantes

- Ce sont des références associées de façon définitive à un objet ou tableau donné

- mot-clé `final` :

```
final double[] tab = new double[10];

// La référence (tab) est figée :
tab = new double[20];  // ici ERREUR

// Le contenu du tableau reste modifiable :
tab[0] = 1.732;  // OK
tab[0] = 3.;     // OK
```


Chaînes de caractères

- Ce sont des objets des classes `String` ou `StringBuffer`
- chaînes littérales : `"une chaîne"`
- concaténation par l'opérateur `+`

- chaînes *non modifiables* : `String`

```
int age = 24;
// Création :
String s1 = "Age : " + age;
// Création par conversion :
float x = 1.618f;
String s2 = String.valueOf(x);
// Utilisation :
int l = s1.length();
char c = s1.charAt(0);
int diff = s1.compareTo(s2);
boolean tst = s2.equals(s1);
s2 = s1;
s2 = "Bonjour";
String[] sub = s1.split(":");
```


Chaînes de caractères (2)

- chaînes *modifiables* : `StringBuffer`

```
// Création :
StringBuffer buf = new StringBuffer("Bonjour");

// Utilisation :
int l = buf.length();
char c = buf.charAt(0);
buf.setCharAt(0, 'L');
buf.insert(3, "gues ");
buf.append("née");
buf.deleteCharAt(6);
String s = buf.toString();
String s2 = buf.substring(7, 11);

// ATTENTION : pour StringBuffer,
//  - pas de compareTo()
//  - equals() ne teste pas l'égalité des chaînes contenues
//  - pas de + entre StringBuffer
//    (par contre buf+s OK si s String, mais produit une String)
```

Types énumérés

- Permet de définir proprement un type ayant un ensemble restreint de « valeurs » possibles
- Mot-clef `enum`
- Exemple :

```
enum CouleurFeu {
  VERT, ORANGE, ROUGE
}
// . . .
CouleurFeu col = CouleurFeu.ROUGE;
```

- Est en fait un type particulier de « classe » (voir plus loin)
- Peut servir dans un `switch` (voir plus loin)

OPERATEURS

Principaux opérateurs

- affectation : `=`
- arithmétiques : `+ - * / %`
- comparaisons : `< <= > >= == !=`
- booléens : `&& || ! ^ & |`
- opérations bit-à-bit (sur les entiers) : `& | ^ ~ << >> >>>`
- opération et affectation simultanées : `+= -= *= /= %= &= |= ^= <<= >>= >>>=`
- pré/post-incrémentation : `++`
- pré/post-décrémentation : `--`
- opérateur ternaire : `?:`
- création tableau ou objet (allocation mémoire) : `new`
- test de type des références : `instanceof`

Opérateurs arithmétiques

Opérateur	Fonction	Usage
+	**addition**	`expr1 + expr2`
-	**soustraction**	`expr1 - expr2`
-	**changement de signe**	`- expr`
*	**multiplication**	`expr1 * expr2`
/	**division**	`expr1 / expr2`
%	**modulo**	`expr1 % expr2`

- Attention aux divisions entre entiers …
- Les fonctions mathématiques sont dans la classe **`Math`** :
  - `Math.pow(x,y)`,
  - `Math.sin(x)`,
  - `Math.log(x)`,
  - …

Opérateurs de comparaison

Opérateur	Fonction
==	**égalité**
!=	**inégalité**
<	**inférieur strict**
<=	**inférieur ou égal**
>	**supérieur strict**
>=	**supérieur ou égal**

- Résultat de type booléen

Opérateurs booléens

Opérateur	Fonction	Usage
&&	**et** (version *optimisée*)	`expr1 && expr2`
||	**ou** (version *optimisée*)	`expr1 || expr2`
^	**ou exclusif** (xor)	`expr1 ^ expr2`
!	**négation**	`! expr`
&	**et** (version *non optimisée*)	`expr1 & expr2`
|	**ou** (version *non optimisée*)	`expr1 | expr2`

- Les opérandes doivent être des expressions à valeurs booléennes
- Remarques :
  - Pour `&&` : le 2e opérande n'est pas évalué si le 1er est faux (pas le cas pour &)
  - Pour `||` : le 2e opérande n'est pas évalué si le 1er est vrai (pas le cas pour |)

Opérateurs bit-à-bit

Opérateur	Fonction	Usage
&	**et**	`op1&op2`
|	**ou**	`op1|op2`
^	**ou exclusif** (xor)	`op1^op2`
~	**négation**	`~op1`
<<	**décalage à gauche** (x2)	`op1<<op2`
>>	**décalage à droite**	`op1>>op2`
>>>	**décalage à droite non signé** (/2)	`op1>>>op2`

- Opérandes entiers uniquement
- Travaillent sur la représentation binaire

Opérateurs d'affectation

- Exemple :

```
int i, j;
i = 0;
j = i;
```

- évaluation de droite à gauche :

```
i = j = 2;    // ➔ i et j valent 2
```

- affectation combinée avec opération arithmétique ou bit-à-bit :

```
+=   -=   *=   &=   …

i += 3;   équivaut à  i = i+3;
```


Opérateurs d'incrémentation et de décrémentation

`i++`	**post-incrémentation**
`++i`	**pré-incrémentation**
`i--`	**post-décrémentation**
`--i`	**pré-incrémentation**

- Opérateurs servant essentiellement à compacter les écritures :

```
n = i++;   équivaut à   n = i;
                        i = i + 1;

n = ++i;   équivaut à   i = i + 1;
                        n = i;
```

Autres opérateurs

- **Ternaire conditionnel** :
  `bool ? expr1 : expr2`
  ➥ vaut `expr1` si `bool` est vrai, et `expr2` sinon

- **Conversion** : `(type)`
  ```
  float x = 1.5f;
  int i = (int) x;
  ```

- **Allocation mémoire** (création de tableaux ou d'objets) : `new`

- **Test de type des références** : `instanceof`
  `(ref instanceof C)` vaut :
  - `true` *si* `ref` référence un objet pouvant être considéré comme une *instance* de la classe `C`
  - `false` *sinon* (y compris si `ref==null`)

Priorité des opérateurs

`[] . (params) expr++ expr--`
`++expr --expr +expr -expr ~ !`
`new (type)expr`
`* / %`
`+ -`
`<< >> >>>`
`< > <= >= instanceof`
`== !=`
`&`
`^`
`|`
`&&`
`||`
`?:`
`= *= /= %= += ...`

- A priorité égale, évaluation de *gauche à droite* pour les opérateurs binaires, *sauf pour les affectations (=, *=, …) qui s'évaluent de droite à gauche*

BLOCS, INSTRUCTIONS DE CONTROLE (TESTS, BOUCLES, ...)

Instructions et blocs

- chaque instruction se termine par un ';'

```
int i;
i = 4*3;
```

- on peut grouper plusieurs instructions en un bloc (délimité par des accolades), exemple :

```
int a=1, b=2;
{ // début de bloc
  int x=a;
  a = b;
  b = x;
} // fin de bloc
x = 3; // ERREUR: x n'est pas connu ici
```

- Remarque : la portée d'une variable va de sa déclaration jusqu'à la fin du bloc où elle est déclarée

Instructions de contrôle

Permettent de modifier l'ordre normal (*séquentiel*) d'exécution des instructions

- exécution conditionnelle :
  - `if … else`
- cas multiples :
  - `switch`
- boucles :
  - `for`
  - `while`
  - `do … while`

Exécution conditionnelle

```
if (boolExpr) instruction;
```

```
if (boolExpr) {
  instruction(s);  // exécuté si VRAI
}
else {
  instruction(s);  // exécuté si FAUX
}
```

```
if (boolExpr) {
  instruction(s);  // exécuté si VRAI
}
else if (boolExpr2) {
  instruction(s);  // ...
} else {
  instruction(s);  // ...
}
```

Cas multiples

```
switch (expr) {
  case cst1:
    // instruction(s) si expr==cst1
  break;
  case cst2:
    // instruction(s) si expr==cst2
  case cst3:
    // instruction(s) si expr==cst3 || expr==cst2
  break;
  default:
    // instruction(s) si aucun des cas prévus
  break;
}
```

- `expr` : de type **entier** ou **`char`** ou bien type **énuméré** (`enum`)
- `cst1`, `cst2`, ... : littéral ou constante (`final`)

Boucles de répétitions

- Deux types de boucles :

```
while (boolExpr) {
  // corps de la boucle :
  instruction(s);
}
```

ou alors

```
// cas avec une et une
// seule instruction
while (boolExpr)
instruction;
```

```
do {
  // corps de la boucle
  // exécuté au moins
  // une fois :
  instruction(s);
} while (boolExpr);
```

ou alors

```
// cas avec une et une
// seule instruction
do instruction;
  while (boolExpr);
```


Boucles d'itération

```
for (initialisations ; boolExpr ; incr) {
  instruction(s);    // corps de la boucle
}
```

- **`initialisations`** : déclaration et/ou affectations, séparées par des virgules
- **`incréments`** : expressions séparées par des virgules

- équivalent à :

```
initialisations;
while (boolExpr) {
  instruction(s);
  incréments;
}
```


Exemples de boucles `for`

```
int somme=0;
for (int i=1 ; i<=n ; i++) {
  somme += i;
}
```

```
for (i=0, j=1 ; i<10 && (i+j)<10 ; i++, j*=2) {
  // . . .
}
```

```
for ( ; ; ) ;
```

Boucles d'itérations simplifiées (« for each »)

```
for (type param : expression) {
  // . . .
}
```

où `expression` est soit un tableau (de type cohérent avec celui de `param`), soit un objet implantant l'interface `Iterable` (voir plus loin).

- Exemples (où `expression` est un tableau) :

```
int[] prems = { 2, 3, 5, 7, 11, 13 };
int somme = 0;
for (int i : prems) {
  somme += i*i;
}
String[] rep = { "oui", "non",  "peut-être" };
for (String s : rep) {
  System.out.println(s);
}
```


Interruptions des boucles

- `break` sortie de la boucle
- `continue` passage à l'itération suivante
- `return` sortie immédiate de la fonction en cours (cf. + loin), donc a fortiori sortie de la boucle en cours par la même occasion

FONCTIONS

Les fonctions

- éléments essentiels de structuration en programmation « impérative »
- en général, une fonction est définie par :
  - un type de retour
  - un nom
  - une liste de paramètres typés en entrée
  - une séquence d'instructions (corps de la fonction)

Eventuellement vide : `public static int f(){//…}`

- en Java, chaque fonction est définie dans une classe (ex. : la classe `Math` regroupe les fonctions mathématiques)
- exemple de fonction :

Type de retour

```
class MaClasse {
  public static int carre(int i ){
    return i*i;
  }
}
```

nom

Fonction : retour

- type de retour : n'importe quel type, ou bien **`void`**
- instructions de retour :
  - `return expression;`
  - `return;` *// possible uniquement pour fonction de type* `void`

```
class MaClasse {
  /** Retourne l'index d'une valeur dans un tableau
   *  (ou -1 si pas trouvée) */
  public static int indexDe(float val, float[] tab) {
    int len = tab.length;
    for (int i=0 ; i<len ; i++) {
      if (tab[i] == val) return i;
    }
    return -1;
  }
}
```


Appel de fonction

- en général, nom de la classe en préfixe :

```
y = Math.sin(x);
k = MaClasse.carre(i);
```

- dans une même classe, par son nom seul :

```
class MaClasse {
  public static int carre(int i) {
    return i*i;
  }
  public static void afficheCarre(int i) {
    int k = carre(i); // On aurait pu aussi bien
                      // écrire : MaClasse.carre(i)
    System.out.println(k);
  }
}
```

Fonction et portée

- Les variables déclarées dans une fonction sont **locales** à cette fonction
- Idem pour les paramètres

```
class Portee {
  public static int calcul(int val){
    int tmp = 3*val;
    return tmp;
  }
  public static void main(String[] args){
    int tmp = 1;
    System.out.println( calcul(9) );
    // Que vaut tmp ici ?
    // Et val ??
  }
}
```

tmp-de-calcul

tmp-de-main

Programme principal

- En Java, l'exécution de programme consiste en la recherche et l'appel d'une *fonction* (*publique*) particulière :
  - nommée `main`
  - prenant en paramètre un `String[]`
  - ne retournant rien
- Exemple, avec utilisation du paramètre :

```
class Echo {
  public static void main(String [] args){
    for (String s : args)
      System.out.print(s + " ");
  }
}

// Exemple d'utilisation :
//    java Echo un deux trois
//    un deux trois
```


Passage des paramètres

- **Par valeur** pour les types « primitifs » :

```
public static void echange(int i, int j){
  int tmp = i;   // i et j : copies locales
  i = j;
  j = tmp;
}
public static void main(String[] args){
  int a=1, b=2;
  echange(a,b);    // a et b ne sont pas modifiés
}
```

- **Par valeur (i.e. recopie)** des références :

```
public static void ech(String s1, String s2) {
  String tmp = s1; // s1 et s2 : copies locales
  s1 = s2;
  s2 = tmp;
}
public static void main(String[] args) {
  String a="oui", b="non";
  ech(a,b);  // a et b ne sont pas modifiés
}
```


Passage des paramètres (2)

- la référence est passée par valeur (i.e. le paramètre est une copie de la référence), **mais le contenu de l'objet référencé peut être modifié par la fonction** (car la copie de référence pointe vers le même objet…) :

```
public static void increm(int t[]) {
  for (int i=0 ; i<t.length ; i++)
    t[i]++;
}

public static void tronquer(StringBuffer b){
  b.deleteCharAt(b.length() - 1);
}

public static void main(String[] args){
  int tab[] = {1, 2, 3};
  increm(tab);
  // tab vaut {2, 3, 4}
  StringBuffer buf = new StringBuffer("oui");
  tronquer(buf);
  // buf vaut "ou"
}
```

Passage de paramètres : autres exemples

```
class Passage {
  public static int fonc(int x){
    x++;
    return x;
  }

  public static void main(String[] args){
    int val = 1;
    int y = fonc(val);
      // Valeur ici de val ?
    int[] tab={1, 2, 3};
    f2(tab);
              // Etat de tab maintenant ?
  }

  public static void f2(int[] t) {
    t = new int[] {2, 2};
  }
}
```


Récursivité

- Une fonction peut s'appeler elle-même
- il faut penser à arrêter la récursion à un moment donné (sinon , récursion infinie…)
- exemple : fonction puissance

```
class ExempleRecursif {
 public static double puiss(double x, int n){
   if ( n==1 )
     return x;
   else
     return x*puiss(x, n-1);
 }
 //...
}
```

L'exemple ci-dessus est-il totalement correct ?

Que se passe-t-il pour n<0 ???

Surcharge

- Possibilité de définir (dans la même classe) plusieurs fonctions de même nom, différenciées par le nombre et/ou le type des arguments :

```
class NomClasse {
  public static int longueur(String s){
    return s.length();
  }
  public static int longueur(int i){
    String si = String.valueOf(i);
    return si.length();
  }
  public static void main(String [] args) {
    int k = 12345;
    int lk = longueur(k);  // lk vaut 5
    String s = "oui";
    int ls = longueur(s);  // ls vaut 3
  }
}
```

- Appel de fonction : les arguments doivent être du type prévu ou d'un type convertible automatiquement vers le type prévu

Fonctions à nombre variable d'arguments

- Possibilité de *terminer* la liste des paramètres d'une fonction par une « ellipse » permettant un appel de fonction avec un nombre variable d'argument(s) *de même type* (sauf si le type est `Object`) :

```
public static double[] f(int exp, double... nb) {
  double[] res = new double[nb.length];
  for (int i=0 ; i<res.length ; i++) {
    res[i] = Math.pow(nb[i], exp);
  }
  return res;
}
//…
double[] t1 = f(2, 2.5);   // nb.length vaudra 1 dans f
double[] t2 = f(2, 1.1, 2.2, 3.3);   // nb.length vaudra 3
double[] t3 = f(2);   // nb.length vaudra 0
```

- Le paramètre avec « ellipse » est traité dans la fonction comme un tableau
- Possibilité d'arguments de types hétérogènes via le type `Object`…

ENTREES-SORTIES STANDARDS

Entrées-sorties standards

- écriture :

```
float z=1.732f;
System.out.print("z=");
System.out.print(z);
System.out.println(",2z=" + (2*z));
```

- lecture :

```
// Lecture d'un caractère à la fois
int car = System.in.read();

// Lecture de 10 caractères
byte buf[] = new byte[10];
int nbLus = System.in.read(buf);
```

NOTA-BENE : utiliser la classe Scanner pour lire autre chose que des caractères ou tableaux de caractères (voir page suivante).

Lecture formatée : classe Scanner

```
import java.util.Scanner;
//…
Scanner sc = new Scanner(System.in);
String ch = sc.next();
int k = sc.nextInt();
double val;
if ( sc.hasNextDouble() )
  val = sc.nextDouble();
String ligne = sc.nextLine();
```

- Il existe une méthode `nextXyz()` et une méthode `hasNextXyz()` pour chaque type primitif `xyz`
- Lance `InputMismatchException` si l'élément suivant n'est pas convertible dans le type demandé
- Possibilité de choisir un séparateur avec `useDelimiter(Pattern)`
- ATTENTION : utilise, par défaut, le format « local » des nombres (par exemple en français : 2,5 et **pas** 2.5) ; possibilité de choisir une « localisation » avec la méthode `useLocale()`

Affichage formaté

- `printf(String, Object…)` est une méthode à nombre d'arguments variable qui facilite l'affichage formaté :

```
double rac2 = Math.sqrt(2);
System.out.printf("sqrt(2)=%.3f", rac2);
// imprime : sqrt(2)=1,414

int i=3, j=30, k=303;
System.out.printf("| %3d | %3d | %3d |", i, j, k);

// imprime : |   3 |  30 | 303 |

double x=2.5, y=-3.755;
System.out.printf(" x:%+.2f \n y:%+.2f", x, y);
// imprime quoi ?
```

- Voir la documentation en ligne pour plus de détails.

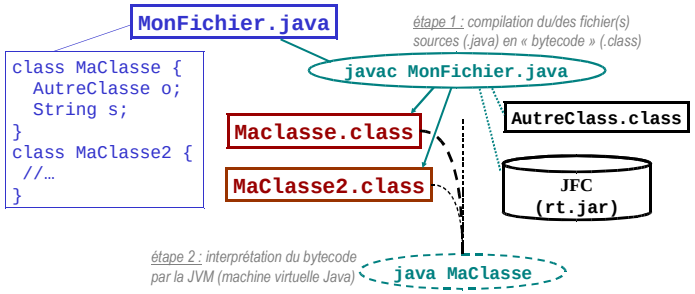
EXEMPLE DE PROGRAMME, COMPILATION, EXECUTION

Exemple de programme

```
// Début du fichier : les « import »
import java.util.Scanner;

// Toutes les autres instructions sont dans une
//   (ou  plusieurs) classe(s)
public class MonProgramme {

  // Un programme « ordinaire » contient toujours une
  // fonction principale (main) comme ci-dessous :
  public static void main(String[] args){
    System.out.print("Entrez un int : ");
    Scanner sc = new Scanner(System.in);
    int n = sc.nextInt();
    for (int i=0; i<n; i++) {
      System.out.println(i);
    } // fin de la boucle for
  } // fin de la fonction main

} // fin de la définition de la classe
```


Compilation et exécution

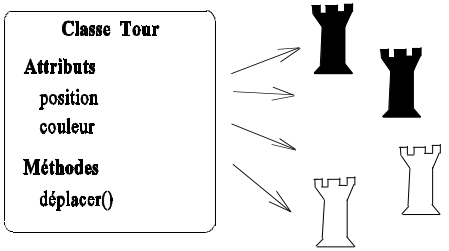
- En général, un fichier `MaClasse.java` doit contenir la classe `MaClasse`, et elle seule (en fait c'est plus compliqué, voir plus loin)
- La compilation de `MaClasse.java` ➔ `MaClasse.class`
- Le fichier `MaClasse.class` peut être exécuté si et seulement si la classe `MaClasse` comporte une fonction « main ».



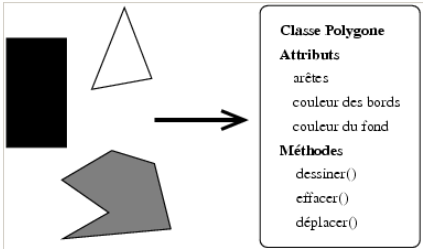
CLASSES

Classes

- ***Classe*** : modèle d'objets ayant les mêmes types de propriétés et de comportements
- Chaque ***instance*** d'une classe possède ses propres valeurs pour chaque attribut
- Exemple :



Concepts « objet » : attributs et méthodes



- **Utilisation :**

```
// DECLARATION
Polygone p;

// CREATION
p = new Polygone(…);

// APPELS DE METHODES
p.dessiner();
p.deplacer(10, 5);
```


Classes

- Une classe est un modèle d'objet défini par ses « membres » (attributs et méthodes)
- Les attributs (état) sont les *données* propres à l'objet
- Les méthodes (comportement) sont les *opérations applicables* à l'objet

- Exemple :

```
class Cercle {
  double rayon=0;
  double calculerSurface() {
    return Math.PI*rayon*rayon;
  }
}
//…
Cercle c = new Cercle();
double surf = c.calculerSurface();
```

Objet

- instance d'une classe
- création uniquement par `new` :

```
Cercle c1;
// c1 : référence non encore initialisée
c1 = new Cercle();
// c1 : référence à une instance de la classe Cercle
```

- destruction automatique par le « ramasse-miettes » (garbage collector) quand il n'y a plus aucune référence vers l'objet :

```
Cercle c2 = new Cercle();
// ...
c2 = c1;
// destruction de l'instance créé par new ci-dessus
// (aucune référence ne pointant plus dessus)t
```


Attribut

- Utilisation dans le cas général → usage de l'opérateur `.` (« point »)

```
Cercle c = new Cercle();
double r = c.rayon;
```

- Utilisation depuis une méthode de la classe → accès direct pour l'instance courante

```
class Cercle {
  // ...
  double comparerA(Cercle autre){
    return rayon - autre.rayon;
  }
}
```


Méthode

- invocation « directe » depuis une autre méthode de la même classe :

```
class Cercle {
  //...
  void afficherAire(){
    double a = calculerSurface();
    System.out.println(a);
  }
}
```

- invocation des méthodes dans les autres cas → usage de l'opérateur `.`

```
Cercle c1 = new Cercle();
//...
double s = c1.calculerSurface();
```

- « **surcharge** » possible : plusieurs méthodes de même nom, mais de *signatures* différentes (i.e. avec types de paramètres différents)

La référence « `this` »

- possibilité dans les méthodes de désigner *explicitement* l'instance courante :
  - pour accéder à un attribut « masqué » par un paramètre :

```
class Cercle {
  double rayon=0;
  // ...
  double comparerA(double rayon){
    return this.rayon-rayon;
  }
}
```

  - pour appeler une fonction (ou méthode) avec l'instance courante en paramètre :

```
class Cercle {
  void dessinerSur(Ecran e){
    // ...
  }
}
class Ecran {
  // ...
  void tracer(Cercle c){
    c.dessinerSur(this);
  }
}
```

Encapsulation

- cacher le détail d'implantation, et ne laisser voir que l'interface fonctionnelle
- pour chaque attribut et méthode, visibilité possible :
  - `public`
  - `private`
  - `protected`
  - par défaut, accès « package »
- encapsulation « usuelle » : attributs privés, et méthodes publiques (sauf celles à usage interne) :

```
class Cercle {
  private double rayon;
  public double calculerSurface() {
    return Math.PI*rayon*rayon;
  }
  // ...
}
```


Constructeur

- Contient les instructions à exécuter à la création des instances (en particulier : l'initialisation des attributs) :

```
class Cercle {
  private double rayon;
  public double calculerSurface(){
    return Math.PI*rayon*rayon;
  }
  public Cercle(double r) {
    rayon = r;
  }
}
```

- Est appelé à chaque création d'instance (par `new`) :

```
Cercle c = new Cercle(2.5);
```

- NOTA-BENE : ***jamais*** de type de retour (ce n'est pas une méthode)

Constructeur (2)

- Il peut y avoir plusieurs constructeurs (mais les signatures doivent être différentes)
- On peut invoquer un constructeur depuis un autre avec `this(...)` :

```
class Cercle {
  // ...
  public Cercle(double r) {
    rayon = r;
  }
  public Cercle() {
    this(0);   // invoque Cercle(0)
  }
}
```

- Conseils :
  - écrire un constructeur « complet » avec tous les paramètres possibles ;
  - ensuite, faire en sorte que tous les autres constructeurs (en général un constructeur *sans argument* et un *pour construire par recopie d'un objet de même type*) appellent le constructeur « complet »

Destructeur

- appelé par le « ramasse-miettes » (garbage collector ou GC) avant de supprimer l'objet
- donc habituellement jamais appelé explicitement
- utile essentiellement pour fermer des fichiers associés à l'instance et susceptibles d'être encore ouverts à sa disparition
- `protected void finalize() {...}`
- appelé une seule fois par instance (donc pas ré-appelée par le GC si appelé « à la main » avant)

Tableau d'objets

- ATTENTION : un tableau d'objets est en fait un tableau de ***références*** vers objet :

```
Cercle [] tabC = new Cercle[10];
// tabC[0]==null
```

- Il faut donc allouer les objets eux-mêmes ensuite :

```
for (int i=0 ; i<10 ; i++) {
  tabC[i] = new Cercle();
}
```

- Par exemple, soient les 2 instructions suivantes :

```
String [] tabChaine;
tabChaine = new String[2];
```

Que contient alors `tabChaines` ??

Et si on fait maintenant :

```
tabChaine[0] = "oui";
tabChaine[1] = "non";
```

Comment cela se présente-t-il en mémoire ?

Attribut « statique »

- variable partagée par *toutes* les instances de la classe (nommé aussi attribut « **de classe** »)
- mot-clef : `static`

```
class Cercle {
  //...
  static int nbCercles = 0;
  static public double[] defRayon;
  static {  // initialiseur statique
    defRayon = new double[10];
    for (int i=0 ; i<10 ; i++) {
      defRayon[i] = 3*i;
    }
  }
  public Cercle(double r){
    nbCercles++;
    // ...
  }
}
```


Constantes de classe

- attribut à la fois `static` et `final`

- exemple :

```
class Math {
  //...
  static final double PI = 3.1415926;
}
```


Méthode « statique »

- type de méthode ne s'appliquant pas à une instance particulière de la classe (appelée aussi méthode « **de classe** »)
- équivalent des fonctions « ordinaires » des langages non-objet
- mot-clef : `static`
- exemples :
  - fonctions mathématiques prédéfinies telles que : `Math.random(), Math.sin(double), …`
  - fonction principale (`main`)
  - `class Cercle {`

```
// ...
static private int epaisseur = 1;
static public void setTrait(int e){
  epaisseur = e; }
}
```

Appel de méthode « statique »

- appel depuis une autre méthode de la même classe :

```
class Cercle {
  // ...
  public void bidule(){
    setTrait(0);
  }
}
```

- appel depuis l'extérieur de la classe → préfixer par le nom de la classe :

```
Cercle.setTrait(2);
```


Cas particulier de classe : type énuméré

- Classe ayant un **nombre FINI** d'instances, toutes prédéfinies
- Mot-clef : `enum` (au lieu de `class`)
- Exemple d'énumération « simple » :

```
enum Reponse {
  OUI, NON, PEUT_ETRE
}
//…
Reponse r1 = Reponse.OUI;
```


Type énuméré « élaboré »

- Un `enum` est une classe, il peut avoir des attributs et méthodes :

```
enum Jeune {
  BEBE("Bebe", 0, 3),
  ENFANT ("Enfant", 3, 12),
  ADO("Adolescent", 12, 18);
  private String type;
  private int ageMin;
  private int ageMax;
  Jeune(String type, int ageMin, int ageMax){
    this.type = type;
    this.ageMin = ageMin;
    this.ageMax = ageMax;
  }
  public String getType(){ return type; }
  public int getAgeMin(){ return ageMin; }
  public int getAgeMax(){ return ageMax; }
}
//…
if ( age < Jeune.BEBE.getAgeMax() ) {
  Jeune j1 = Jeune.BEBE;
  // …
```

PAQUETAGE, IMPORT, JavaDoc, classes internes…

Paquetage

- entité de structuration regroupant plusieurs classes (et/ou interfaces) et/ou sous-paquetages
- le paquetage d'appartenance est indiqué au début du fichier source par :
  ```
  package nomPackage;
  ```
- les fichiers `.class` de chaque paquetage doivent être dans un répertoire ayant le nom exact du paquetage
- les paquetages (et classes) sont recherchés dans une liste de répertoires (et/ou de fichiers zip) fixée par la variable d'environnement `CLASSPATH`

Paquetage et visibilité

- par défaut, les classes et interfaces ne sont accessibles que dans leur paquetage : seules celles qui sont déclarées `public` pourront être importées dans d'autres paquetages
- les membres de classes sans accès précisé (i.e. ni `public`, ni `protected`, ni `private`) sont visibles dans tout le paquetage de leur classe
- fichier hors-paquetage => classes et interfaces dans le « paquetage anonyme »

Classe publique

- classe utilisable à l'extérieur de son paquetage
- mot-clef : `public`
  ```
  public class Cercle {
    //...
  }
  ```
- par défaut, une classe n'est utilisable que dans son paquetage (éventuellement le « paquetage anonyme » si pas de paquetage précisé en début de fichier)

Organisation en fichiers, compilation et exécution

- au maximum une classe ou interface *publique* par fichier source (auquel cas, le nom du fichier doit impérativement être celui de la classe ou interface publique contenue dans le fichier)
- il peut y avoir d'autres classes et interfaces non publiques dans le même fichier
- la compilation du fichier `PubClasse.java` :

  `javac PubClasse.java`

  produit autant de fichiers suffixé par .class qu'il y a de classes dans le fichier .java
- exécution du main de la classe `NomClasse` :

  `java NomClasse`

  ou alors (si on est dans le *répertoire-père* de `nomPackage`) :

  `java nomPackage.NomClasse`

Paquetage : importation

- Nom (complet) d'une classe à l'extérieur de son paquetage :

  `nomPackage.NomClasse`

  (sauf si la classe est importée)
- Importation : permet d'utiliser une classe d'un autre paquetage avec juste son nom (sans le préfixer par son package)
- Importation de classe (publique) par :

```
import nomPackage.NomClasse;
```

- Importation « à la demande » de toute classe publique du package :

```
import nomPackage.*;
```


import « statique »

- Permet d'importer tout ou partie des membres (attributs ou méthodes) statiques d'une classe donnée
- Exemple :

```
import java.lang.Math.PI;
import java.lang.Math.sin();
import java.lang.Integer.*;
//…
final double PI2 = PI*PI;  // (PI désigne Math.PI)
double x = sin(PI/3);  // sin() désigne Math.sin()

long val;
//…
if ( val > MAX_VALUE ) {
  // MAX_VALUE désigne Integer.MAX_VALUE
  // …
```


Création de paquetage

- attention au choix du nom (parlant, mais évitant conflit de nom)
- suggestion de nommage : type hostname inversé

  `fr.societe.service.nom_paquetage`
- bien penser la hiérarchisation si plusieurs paquetages liés

Documentation automatique

- outil `javadoc` du JDK
- écrire commentaires spécifiques
  `/** bla-bla ... */`
  juste avant la déclaration :
  - de chaque classe (ou interface)
  - de chaque méthode
  - de chaque attribut
- la commande `javadoc NomClasse.java` produit automatiquement des fichiers HTML décrivant la classe et intégrant ces commentaires
- par défaut, seuls les éléments publics et protégés apparaissent (car ce sont les seuls qu'ont à connaître les utilisateurs de la classe)

Documentation automatique (2)

- On peut insérer dans les commentaires de documentation des tags qui seront formatés de manière spécifique.

- Ils commencent en début de ligne par @, et vont jusqu'à fin de ligne :
  - `@author nom` (pour une classe ou interface)
  - `@param nom description` (pour une méthode: commentaire sur un de ses paramètres)
  - `@return description` (pour une méthode : commentaire sur ce qui est retourné)
  - `@exception nom description` (pour une méthode : commentaire sur un type d'exception potentiellement émise)
  - `@see NomClasse` (lien hypertexte vers la documentation de la classe `NomClasse`)
  - `@see NomClasse#nomMethode` (idem, mais lien vers l'endroit où est décrite la méthode `nomMethode`)

Classes ou interfaces internes (ou « nichées »)

- classes ou interfaces définies *à l'intérieur* d'une autre (au même niveau d'imbrication que les attributs et méthodes)
- intérêt : classes (ou interfaces) utilitaires très liées à la classe englobante
- pour les classes internes, 2 catégories :
  - static : classes « normales » mais fortement liées à l'englobante
  - membres : associées à chaque instance => peuvent accéder directement aux attributs privés de classe englobante
-
```
public class A {
  private int x;
  static public class Liee {  //...  }
  public class Membre {
    void meth(){ x = 1; //...}
  }
}
A.Liee l = new A.Liee();
A a = new A();
A.Membre m = new a.Membre();
```


Classes locales

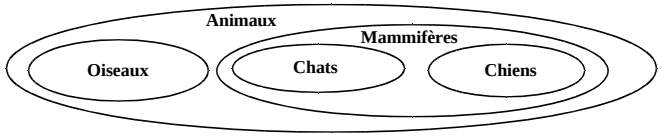
- classes membres définies à l'intérieur du corps d'une méthode (d'une autre classe)
- classes anonymes : classes locales sans nom, définie juste à l'intérieur d'un `new` (pour créer une instance spécialisée ou une implantation)

```
class MonApplet{
  private ActionListener al;
  private String message;
  public void init() {
    //...
    al = new ActionListener() {
      public void actionPerformed() {
        message = "DONE";
      }
    }
  }
}
```

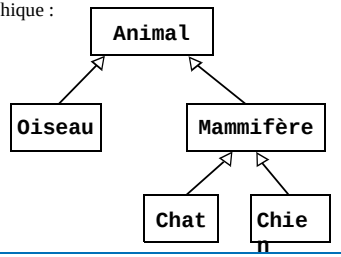
HERITAGE, classe racine, introspection

Notion de hiérarchie de classes

Vision ensembliste :



Vision hiérarchique :



Héritage

- Pour réutiliser une classe existante en l'adaptant, et/ou factoriser des choses communes à plusieurs classes
- Mot-clef : **`extends`**

```
class Figure{
  private String nom;
  protected Position pos;
  public Figure(Position p){ pos = p; }
  public void afficher() { pos.afficher(); }
  public void deplacer(int dx, int dy) {
    pos.ajouter(dx,dy);
  }
}
class Cercle extends Figure{
 // Cas particulier de Figure, hérite de tous ses attributs/méthodes
 // ...
}
```

- En Java, héritage ***simple*** uniquement (pas plus d'une classe mère)

Héritage, attributs et méthodes

- la sous-classe hérite de tous les attributs (y compris « `static` ») et méthodes membres de la classe mère (sauf les méthodes privées et les constructeurs) :

```
Cercle c = new Cercle();
// appel d'une méthode héritée
c.deplacer(2, -1);
```

- La sous-classe peut évidemment avoir des attribus et méthodes supplémentaires correspondant à sa spécificité :

```
class Cercle extends Figure{
  private double rayon;
  public double circonference() {
    return 2*Math.PI*rayon;
    //...
  }
}
```

Héritage et références

- Toute référence vers une instance de la classe fille peut être vue aussi comme une référence vers la classe mère (conversion automatique fille -> mère) :

```
Figure f = new Cercle(); // OK
// Affectation d'un Cercle dans une Figure

Cercle c = new Cercle(1);
if ( f.contient(c) ) { // …
  // OK : passage d'un Cercle en paramètre
  // à  une méthode attendant une Figure
```


Héritage et visibilité

- Une classe fille ne peut accéder qu'aux membres (attributs ou méthodes) « *publics* » ou « *protégés* » de sa classe mère (ainsi qu'aux attributs et méthodes « *package* » ssi elle appartient au même paquetage que sa classe mère) :

```
class Cercle extends Figure {
  //...
  void essai(){
    // essai d'accès à un champ privé
    // de la classe mère
    String s = nom; // ERREUR !
  }
}
```


Héritage et constructeurs

- les constructeurs ne sont pas hérités, mais on peut appeler ceux de la classe mère avec `super(…)` :

```
class Cercle extends Figure{
  //...
  Cercle(float r, Position pos) {
    // appel du constructeur Figure(pos)
    super(pos);
    rayon = r;
  }
}
```

- l'appel `super(…)` doit ***impérativement*** être la 1ère instruction du constructeur
- si la 1ère instruction n'appelle ni le constructeur de la classe mère, ni un autre constructeur de la classe fille, alors appel automatique de `super()` sans argument
- ordre des opérations :
  1/ appel du constructeur de la classe mère
  2/ initialiseurs et blocs d'initialisation
  3/ corps du constructeur fille

Spécialisation de méthode

- on peut définir un comportement différent pour les méthodes héritées (et accessibles) en redéfinissant une méthode de même nom et même prototype (signature) :

```
class Cercle extends Figure{
  private float rayon;
  //...
  public void afficher() {
    // appel de la méthode de la mère
    super.afficher();
    System.out.println("rayon=" + rayon);
  }
}
```

- ... sauf pour les méthodes déclarées `final` dans la classe mère
- la visibilité de la méthode redéfinie peut être différente (mais seulement augmentée)

Polymorphisme dynamique

- Quand on manipule un objet via une référence à une classe mère, ce sont toujours les méthodes (non statiques) de la classe *effective* de l'objet qui sont appelées :

```
Figure f = new Cercle();
// Appel de afficher() de Cercle
//   (bien que référence de type Figure) :
f.afficher();
```


Méthode abstraite

- méthode non implantée (on ne fait que spécifier son nom, les types de ses paramètres et son type de retour)
- mot-clef : `abstract`

```
class Figure {
  //...
  public abstract void colorier();
}
```

- destinée à être (re)définie dans les classes filles
- ne peut exister que dans une classe elle-même déclarée abstraite (cf. ci-après)
- impossible pour les méthodes « statiques »

Classe abstraite

- classe non instanciable (sert uniquement de classe mère)
- mot-clef : `abstract`

```
abstract class Figure {
  //...
}
```

- toute classe qui définit une méthode abstraite (ou qui en hérite et ne la redéfinit pas) doit obligatoirement être déclarée abstraite

Classe non dérivable

- classe qui ne pourra pas servir de classe mère
- mot-clef : `final`

```
final class ClasseTerminee {
   //...
}
```

- intérêt : sécurité
- exemple : beaucoup de classes du paquetage `java.lang`

Méthodes non redéfinissables

- Méthode qui ne pourra pas être redéfinie dans les classes filles
- Mot-clef : `final`
- Exemple :

```
abstract class Figure {
  Position pos;
  final public deplacer(int dx, int dy){
    pos.ajouter(dx, dy);
  }
}
```

- Comme classes non dérivables, mais plus flexible

Héritage et tableau

- si `Fille` dérive de `Mere`, alors `Fille[]` est considéré comme un sous-type de `Mere[]` :

```
Fille[] tabF = new Fille[3];
Mere[] tabM = tabF;    // OK
```

- **typage dynamique** : même manipulé via un `Mere[]`, un tableau de `Fille` ne peut contenir que des références à `Fille` :

```
tabM[0] = new Mere();  // Erreur
```


Classe racine `Object`

- Ancêtre de toutes les classes
- Définit donc des méthodes héritées par *toutes* les classes :
  - `public boolean equals(Object obj)`
    par défaut, retourne this==obj (compare les références), mais prévue pour être redéfinie pour comparer les contenus (ex : classe String)
  - `public String toString()`
    par défaut, retourne "NomClasse@"+hashCode(), mais à redéfinir en une représentation textuelle de l'objet pertinente dans un System.out.print()
  - `public int hashCode()`
  - `protected Object clone()`
  - `public Class getClass()`

  ...

Méthode `clone()`

- duplique l'objet auquel elle est appliquée (copie des attributs)
- tout objet (et tableau) en hérite
- utilisable directement pour les tableaux « ordinaires » :

```
int[] tab = { 1, 2, 3, 4 };
int[] tab2 = tab.clone();
```

- attention si tableau multi-dim ou d'objets, car copie des références
- pour les objets que l'on veut clonables :
  1/ déclarer que la classe implante l'interface `Cloneable`,
  2/ redéfinir la méthode `clone()` comme `public`, et soit retournant `super.clone()`, soit adaptée à la classe

Programmation « générique » *non-typée*, via la classe `Object`

- La classe `Object` permet aussi de faire de la programmation générique, i.e. des classes ou fonctions peuvant fonctionner avec des instances de n'importe quelles classes
- Exemple de fonction générique :

```
int chercher(Object o, Object[] tab) {
  for (int i=0 ; i<tab.length ; i++) {
    if (o.equals(tab[i])) {
      return i;
    }
    return -1;
  }
}
```

- Exemple de fonction « générique » prédéfinie :
  `java.util.Arrays.sort(Object[] tab)`
- NOTA-BENE : depuis Java 1.5, la programmation « générique » se fait plutôt sous forme ***paramétrée*** (syntaxe proche des « templates » de C++, voir plus loin)

Introspection : classes `Class, Method, Field,...`

- chaque classe (et interface) est représentée par une instance de `Class`
- permet notamment d'instancier une classe à partir de son nom :

```
Class cl;
cl = Class.forName("NomDeClasse");
Object o = cl.newInstance();
```

- permet de tout savoir sur la classe (introspection) :

```
Method[] getDeclaredMethods()
Field[] getDeclaredFields()
Constructors[] getDeclaredConstructors()
Class[] getInterfaces()
Class getSuperClass()
boolean isInterface()
boolean isArray()
boolean isPrimitive()
```

INTERFACES

Nature et intérêt des interfaces

- définition abstraite d'un service, indépendamment de la façon dont il est implanté (« type abstrait de données »)
- concrètement, ensemble de méthodes publiques abstraites (et de constantes de classe)
- facilite la programmation générique
- permet un héritage multiple restreint

Définition d'interfaces

- Exemples :

```
interface Redimensionnable {
  void grossir(int facteur);
  void reduire(int facteur);
}

interface Coloriable {
  Couleur ROUGE = new Couleur("rouge");
  //...
  void colorier(Couleur c);
}

interface Pile {
  Object sommet();
  void empiler(Object o);
  void depiler();
  boolean estVide();
}
```


Utilisation des interfaces

- utilisation : toute classe peut *implanter* une (ou plusieurs) interface(s) :

```
class Cercle implements Coloriable, Redimensionnable {
  //...
}
```

- une classe qui implante une interface doit redéfinir ***toutes*** les méthodes de l'interface (ou bien être abstraite) :

```
class Cercle implements Coloriable, Redimensionnable {
  int rayon;
  //...
  public void grossir(int facteur) {
    rayon *= facteur;
  }
  public void reduire(int facteur) {...}
  public void colorier(Couleur c) {...}
}

abstract class Figure implements Coloriable {
  // classe abstraite ==> pas indispensable de redéfinir
  // la méthode colorier()
  // ...
}
```

Références de type interface

- Chaque interface définit un type de référence
- Une référence à une interface peut désigner toute instance de toute classe qui implante l'interface en question :

```
Redimensionnable redim;
Coloriable col;
Cercle cercle = new Cercle();
redim = cercle;
col = cercle;

// cercle, col et redim forment 3 vues différentes
// du même objet...
// ...et les méthodes applicables dépendent
// de la vue utilisée :

col.colorier(Coloriable.ROUGE); // OK
col.grossir(2);                 // ERREUR
redim.grossir(2);               // OK
cercle.colorier(Coloriable.ROUGE); // OK
```


Programmation « générique » avec les interfaces

- Si une méthode/fonction prend en paramètre une référence à interface, on peut lui passer en argument une référence à toute instance de toute classe qui implante cette interface :

```
interface Comparable {
  int compareTo(Object o);
}
class Tri {
  public static void trier(Comparable[] tab) {
    //...
    if (tab[i].compareTo(tab[j]) > 0)
      //...
  }
}
class W implements Comparable {  //...
  public int compareTo(Object o) { ... }
}
W[] tabW = new W[10];
// ...
Tri.trier(tabW);
```

- NOTA-BENE : depuis Java 1.5, la programmation « générique » se fait plutôt sous forme ***paramétrée*** (proche des « templates » de C++ : voir plus loin)

Les interfaces comme paramètres de type fonction

- Une interface peut jouer le rôle d'une sorte de « référence à fonction » :

```
interface FonctionAUneVariable{
  double valeur(double);
}
class Integration {
  public static double integrer(FonctionAUneVariable f,
                                double deb, double fin, double pas){
     double s=0., x=deb;
     long n = (long) ((fin-deb)/pas)+1;
     for (long k=0 ; k<n ; k++, x+=pas)
       s += f.valeur(x);
     return s*pas;
  }
}
class Exp implements FonctionAUneVariable {
  double valeur(double d) {
    return Math.exp(d);
  }
}
double d = Integration.integrer(new Exp(), 0., 1., 0.001);
```


Interface et héritage

- Une interface peut dériver d'une (ou plusieurs) autres interfaces :

```
interface X {...}
interface Y {...}
interface Z extends X,Y{...}
```

- si une classe `Mere` implante une interface, alors toutes ses classes filles héritent de cette propriété :

```
class Mere implements Z {...}
class Fille extends Mere {...}
//...
Z z = new Fille(); // OK
X x = new Fille(); // OK
// OK car Fille implante (par héritage)
// l'interface Z (et donc aussi l'intreface X)
```

Interfaces « marqueurs »

- Une interface peut ne contenir aucune méthode ni constante, et servir juste de « marqueur » pour indiquer une propriété des classes qui l'implantent
- Exemples :
  - interface `Cloneable` pour identifier les classes aux instances desquelles on peut appliquer la méthode de duplication `clone()`
  - interface `Serializable` pour identifier les classes dont on peut sauvegarder des instances sur fichier (cf `java.io`)

EXCEPTIONS

Principe des exceptions

- Mécanisme pour traiter les anomalies se produisant à l'exécution
- Principe fondamental = séparer *détection* et *traitement* des anomalies :
  - signaler tout problème dès sa détection
  - mais regrouper le traitement des problèmes ailleurs, en fonction de leur type

Exemple (1 / 3)

- Que fait-cette méthode ?

```
static void divTab(int [] tab) {
  Scanner sc = new Scanner(System.in);
  int index = sc.nextInt();
  int div = tab[index];
  for (int j=0 ; j<tab.length ; j++) {
    tab[j] /= div;
  }
}
```


Exemple (2 / 3)

- Gestion des erreurs sans utiliser d'exceptions :

```
static void divTab(int [] tab) {
  if( tab == null )
    System.err.println("tab==null");
  else {
    Scanner sc = new Scanner(System.in);
    int index = sc.nextInt();
    if( index<0 || index>=tab.length )
      System.err.println("index incorrect");
    else {
      int div = tab[index];
      if( div == 0 ) {
        System.err.println("diviseur nul");
        else {
          for (int j=0 ; j<tab.length ; j++)
          tab[j] /= div;
        }
      }
    }
  }
}
```

Exemple (3 / 3)

- Utilisation des exceptions de Java :
  - toute exception générée provoque l'arrêt brutal du programme ;
  - pour éviter cela, prévoir la *capture* et le *traitement* des exceptions ;
  - mots-clefs : `try`, `catch` et `finally`

```
static void divTab2(int[] tab) {
  try {
    Scanner sc = new Scanner(System.in);
    int index = sc.nextInt();
    int div = tab[index];
    for (int j = 0 ; j < tab.length ; j++)
      tab[j] /= div;
  } catch (NullPointerException e) {
     System.err.println("tab==null");
  } catch (ArrayIndexOutOfBoundsException e) {
     System.err.println("index incorrect");
  } catch (ArithmeticException e) {
     System.err.println("diviseur nul");
  } finally { // on passe toujours ici }
}
```


Traitement des exceptions

- Si une exception est générée dans le bloc `try{ … }` :
  1. on passe immédiatement dans le premier gestionnaire `catch` compatible avec l'exception (i.e. attrapant exactement le type d'exception généré, ou bien un type « ancêtre » au sens de l'héritage)
  2. on exécute le `catch`
  3. puis on continue APRÈS l'ensemble `try{…}catch(…){…}`
- Si pas de `catch` adéquat, on remonte au `try` « englobant » le plus proche, on cherche un `catch` correspondant, etc...
  (et arrêt programme si fin de la pile d'appels)
- Le `finally` est facultatif ; il est exécuté à la fin du bloc `try` quoiqu'il arrive (fin normale, sortie par return, sortie après catch, ou avec exception non traitée)

Nature des exceptions

- Concrètement, une exception est un objet instance de classes spécifiques
- Divers types d'exceptions (classes) sont prédéfinies, et susceptibles d'être générées automatiquement à l'exécution, par exemple :
  - `ArithmeticException`
  - `ArrayIndexOutOfBoundsException`
  - `NullPointerException`
  - `FileNotFoundException`
- On peut aussi définir de nouveaux types d'exceptions

Catégories d'exceptions

- classe mère : `Throwable`
- erreurs « système » : classe `Error` (dérivée de `Throwable`)
- autres anomalies : classe `Exception` (dérivée de `Throwable`) et ses sous-classes
- cas particuliers : classe `RuntimeException` (dérivée de `Exception`) et ses sous-classes = exceptions « non contrôlées » (voir plus loin)
- méthodes communes :
  - `NomException(String)` : constructeur avec message explicatif sur la cause de l'exception
  - `String getMessage()` : renvoie le message explicatif en question
  - `String toString()` : renvoie le type de l'exception (nom de sa classe) et le message explicatif
  - `void printStackTrace()` : affiche la pile d'appel jusqu'au point de lancement de l'exception

Exceptions contrôlées ou non

- Les exceptions de type `RuntimeException` et ses sous-classes (par exemple `ArithmeticException`, `NullPointerException`, `ArrayIndexOutOfBoundsException`) sont dites « non contrôlées »
- contrairement aux autres exceptions (dites « contrôlées »), on peut en lancer une, ou appeler une méthode risquant d'en lancer une, sans être **obligé** :
  - de faire cet appel dans un `try{}` avec le `catch()` correspondant
  - soit de mettre dans la méthode englobante la clause `throws` adéquate (cf. + loin)

Types d'exception usuels

- `Exception`
  - `RuntimeException`
    - `ArithmeticException`
    - `NullPointerException`
    - `NegativeArraySizeException`
    - `ArrayIndexOutOfBoundsException`
    - `StringIndexOutOfBoundsException`
    - `IllegalArgumentException`
      - `NumberFormatException`
    - `UnsupportedOperationException`
  - `IOException`
    - `FileNotFoundException`
    - `EOFException`
  - `InterruptedException`

Lancement d'exception

- Possibilité de lancer « manuellement des exceptions » (en plus de celles générées automatiquement)
- mot-clef : `throw`
- Exemple :

```
if (test_anomalie)
  throw new Exception("blabla");
if (autre_test)
  throw new IllegalArgumentException("bla");
```

- interrompt immédiatement le cours normal du programme pour rechercher un gestionnaire adéquat englobant (cf. traitement des exceptions)
- lancer de préférence une exception d'un type spécifique à l'anomalie et contenant un texte précisant l'anomalie

Création de types d'exception

- Nouveaux types d'exception : il suffit de créer une sous-classe de `Exception` ou de `RuntimeException` (ou d'une de leurs sous-classes prédéfinies) :

```
class MonException extends Exception {
  MonException(String s) {
    super(s);
    //…
  }
  //…
}

// ... puis plus loin :
if (test_anomalie) {
  throw new MonException("commentaire");
}
```

Clause `throws`

- toutes les exceptions (sauf celle dérivant de `RuntimeException`) sont dites « contrôlées »
- une méthode doit déclarer les exceptions « contrôlées » :
  - qu'elle envoie elle-même
  - qu'elle laisse passer (i.e. émises par méthodes appelées, et non traitées)

```
// ...
void lire() throws MonException, IOException {
  // lancement explicite d'une exception si pb. :
  if (testAnomalie() == true) {
    throw new MonException("bla");
  }
  // appel d'une méthode susceptible de générer
  // une exception contrôlée de type IOException :
  int car = System.in.read();
}
```

PROGRAMMATION GENERIQUE
PARAMETREE
(“generics”)

Motivation pour la généricité

```
class UtilTab {
 public static String[] concatenerTableauxChaines
                  (String[] t1, String[] t2) {
  String[] res = new String[t1.length + t2.length];
  System.arraycopy(t1, 0, res, 0, t1.length);
  System.arraycopy(t2, 0, res, t1.length, t2.length);
  return res;
 }

 static public void main(String[] args) {
  String[] ts1 = { "un", "deux", "trois" };
  String[] ts2 = { "quatre", "cinq" };
  String[] concat = concatenerTableauxChaines(ts1, ts2);
 }
}
```

- Autant de fonctions à écrire que de type de tableaux que l’on souhaite pouvoir concaténer !!

Motivation pour la généricité (2)

```
class CoupleDeChaines {
  private String premier, deuxieme;
  public CoupleDeChaines(String s1,String s2){
    premier = s1;
    deuxieme = s2;
  }
  public void afficher () {
    System.out.println( "(" + premier + " ; " + deuxieme + " )" );
  }
  public static void main(String[] args) {
    CoupleDeChaines c = new CoupleDeChaines("un", "deux");
    c.intervertir();
    String s = c.premier();
  }
  public String premier() { return premier; }
  public String deuxieme() { return deuxieme; }
  public void intervertir() {
    String tmp=premier;
    premier = deuxieme;
    deuxieme = tmp;
  }
}
```


Inconvénients de la généricité « non-typée »

```
class UtilTab {
  public static Object[] concatenerTableauxObj
                                  (Object[] t1, Object[] t2) {
    Object[] res = new Object[t1.length + t2.length];
    System.arraycopy(t1, 0, res, 0, t1.length);
    System.arraycopy(t2, 0, res, t1.length, t2.length);
    return res;
  }
  static public void main(String[] args) {
    String[] ts1 = { "un", "deux", "trois" };
    String[] ts2 = { "quatre", "cinq" };
    Object[] concat = concatenerTableauxObj(ts1, ts2);
    Integer[] ti1 = new Integer[] { 1, 2, 3 };
    Integer[] ti2 = new Integer[] { 4, 5 };
    concat = concatenerTableauxObj(ti1, ti2);
  }
}
```

- Type de retour indépendant du type ***réel*** des tableaux passés en arguments
- Impossible de définir des dépendances requises entre les types des paramètres (et de retour)

Inconvénients de la généricité « non-typée » (2)

```
class CoupleObjets {
  private Object premier, deuxieme;
  public CoupleObjs(Object o1, Object o2) {
    premier = o1; deuxieme = o2;
  }
  public Object premier() {
    return premier;
  }
  // …
  static public void main(String[] args) {
    CoupleObjs cs = new CoupleObjs("un", "deux");
    String s = (String) (cs.premier());
    cs = new CoupleObjs("un", new Integer(11));
    CoupleObjs ci = new CoupleObjs(new Integer(1), new Integer(2));
    cs = ci;  // OK (même type)
  }
}
```

Types de retour des méthodes indépendants du type ***réel*** des objets du couple créé (➔ nécessité de conversions à l'utilisation…)

Impossible :

- de définir des couples ayant ***une certaine homogénéité de type*** (la vérification n'est possible qu'à l'exécution) ;
- de distinguer ***au niveau type*** des couples ayant des contenus de types différents.

Les « generics » de Java

- Java 1.5 a introduit dans le langage une syntaxe (désignée sous le terme de « generics ») permettant une programmation générique paramétrée assez similaire (au premier abord) aux « templates » de C++

- Permet d'ajouter à une classe, à une interface, ou à une méthode, un (ou plusieurs) paramètre(s) formel(s) de type sous la forme : **`<NomDeTypeFormel1, NomDeTypeFormel2, …>`**

  [en pratique, par souci de lisibilité, on utilise généralement pour le(s) type(s) formel(s) un (des) nom(s) d'une seule lettre majuscule telle que T,S,U,V ou E]

Méthode générique paramétrée

```
class UtilTab {
  public static <T> T[] concatenerTableaux(T[] t1, T[] t2) {
    T[] res = (T[]) java.lang.reflect.Array.newInstance(t1.getClass()
    getComponentType(), t1.length + t2.length);
    System.arraycopy(t1, 0, res, 0, t1.length);
    System.arraycopy(t2, 0, res, t1.length, t2.length);
    return res;
  }

  static public void main(String[] args) {
    String[] ts1 = { "aa", "bb", "cc" }, ts2 = { "dd", "ee" };
    String[] concat = concatenerTableaux(ts1, ts2); // T<->String
    Integer[] ti1 = new Integer[] { 1, 2, 3 };
    Integer[] ti2 = new Integer[] { 4, 5 };
    Integer[] ti = concatenerTableaux(ti1, ti2);// T<->Integer
  }
}
```

- <T> définit un ***paramètre formel de type*** pouvant être remplacé par n'importe quelle ***classe***
- Valeur de T déterminée à la compilation (classe la plus spécifique possible telle que la signature des méthodes soit compatible avec les types des arguments d'appel)

Classe générique paramétrée

```
class CoupleGenerique<T> {
  private T premier, deuxieme;
  public CoupleGenerique(T o1, T o2) {
    premier = o1;  deuxieme = o2;
  }
  public T premier()  { return premier; }
  public T deuxieme() { return deuxieme;}
  public void intervertir() {
    T tmp = premier; premier = deuxieme; deuxieme = tmp;
  }
  static public void main(String[] args) {
    CoupleGenerique<String> cs
      = new CoupleGenerique<String>("un", "deux");
    cs.intervertir();
    String s = cs.premier();
    CoupleGenerique<Integer> ci
      = new CoupleGenerique<Integer>(new Integer(3), new Integer(5));
    Integer val = ci.deuxieme();
    // cs = ci; IMPOSSIBLE (types differents)
  }
}
```

- Valeur du type T définie explicitement par le programmeur à chaque usage de la classe

Classe générique paramétrée et héritage

- Une classe générique paramétrée `GenClasse<T>`:
  - peut dériver d'une classe non-générique
    ```
    class GenClasse<T> extends Number
    ```
    (d'ailleurs `Object` est ancêtre de `GenClasse<T>`, comme de n'importe quelle classe Java)
  - peut servir de mère :
    - à d'autres classes génériques
      ```
      class FilleGenClasse<T> extends GenClasse<T>
      class GenClasse2<T,V> extends GenClasse<T>
      ```
    - à des classes non-génériques
      ```
      class FilleNonGen extends GenClasse<String>
      ```

- ATTENTION : il n'y a ***aucun lien d'héritage*** entre `GenClasse<Fille>` et `GenClasse<Mere>` (où `Fille` dérive de `Mere`)

- On peut spécifier des contraintes (« bornes ») sur les valeurs possibles du type `T` :
```
class SpecifGenClasse<T extends Animal>
//…
SpecifGenClasse<Chien> gc;
// OK : Chien est un « descendant » de Animal
SpecifGenClasse<String> gs;
// ERREUR : String « hors bornes de type »
```


Interfaces et généricité paramétrée

- On peut naturellement définir aussi des ***interfaces*** génériques paramétrées. Exemple :
```
interface Pile<T> {
  void empiler(T obj);
  T sommet();
  void depiler();
  boolean estVide();
}
```

- Une classe générique paramétrée peut être déclarée comme implantant une (ou des) interface(s), générique(s) ou non :
```
class GenClasse<T> implements Cloneable
class GeCla<T> implements Cloneable, Pile<T>,Comparable<GeCla<T>>
```

- Des interfaces peuvent intervenir dans la contrainte sur le type T d'une classe générique :
```
class Clonage<T extends Cloneable> {//…}
class PoulaillerUniforme<T extends Animal & Cloneable & Ovipare> {
  //…
}
class Groupe<T extends Comparable<T>> {//…}
```


Types paramétrés et « wildcard »

- Chaque valeur correcte de T pour une classe générique `GenClasse<T>` définit un type différent et incompatible. Exemple :
  `CoupleGenerique<String>` et `CoupleGenerique<Integer>`

- On peut cependant avoir besoin de manipuler de manière commune des instances de différentes variantes d'une même classe générique, ce qui est possible grâce au « wildcard » (noté ?) :
```
static public void intervertir(CoupleGenerique<?> c){
    c.intervertir();
}
// ...
CoupleGenerique<String> cs = newCoupleGenerique<String>("un","deux")
CoupleGenerique<Integer> ci = new CoupleGenerique<Integer>(3, 4);
intervertir(cs);
intervertir(ci);
```
Note : on peut faire pointer un **`CoupleGenerique<?>`** vers n'importe quel **`CoupleGenerique<T>`** , par contre les méthodes ayant un paramètre de type T seront inaccessibles via cette référence…

Wildcard borné

- On peut aussi avoir besoin d'être plus spécifique sur les valeurs acceptables pour le type T :
  `GenClasse<? extends ClasseOuInterface>`
  ➔ OK pour tout T héritant de (ou implantant) `ClasseOuInterface`

  Exemple :
```
static public double diff(CoupleGenerique<? extends Number> c){
  double res = c.premier().doubleValue();
  res -= c.deuxieme.doubleValue();
  return res;
}
//…
CoupleGenerique<Integer> ci = new CoupleGenerique<Integer>(3,10);
CoupleGenerique<Double> cd = new CoupleGenerique<Double>(5.5,3.3);
double x = diff(ci);
double y = diff(cd);
```

- Note : possible, inversement, de définir une borne « ***inférieure*** » pour le type T, par exemple : **`Pile<? super Number>`**

Classes et interfaces génériques prédéfinies

- De nombreuses classes et interfaces prédéfinies en Java sont génériques.
- C'est en particulier le cas de toutes les « collections » de `java.util` (voir cette section)

THREADS

Programmation multi-threads

- un « thread » est une séquence d'instructions exécutées séquentiellement
- en Java, un même programme peut lancer plusieurs threads
  => exécution « en parallèle » de plusieurs processus (partageant les mêmes données)
- Intérêts : intéractivité, réactivité (essentiellement pour les interfaces homme-machine), et plus léger que multi-process Unix
- Inconvénients : problèmes de synchronisation, de gestion des ressources partagées, risque de « deadlock », caractère non déterministe

Thread

- un thread Java est un fil d'exécution interne au programme, représenté par un objet (instance de la classe `Thread`) qui possède :
  - une instruction courante
  - un état (actif, inactif, en attente, ...)
  - un nom (facultatif)
- il faut donc d'abord le créer, et ENSUITE le démarrer

Classe `Thread`

- démarrage par la méthode `start()`, qui exécute la méthode `run()`
- arrêt quand on sort de `run()` par fin normale ou exception
- mise en sommeil du thread courant : méthode statique `Thread.sleep(long milliseconds)`
- changement de thread courant (pour laisser la main aux autres threads) : méthode statique `Thread.yield()`

Création de threads

- Deux approches pour créer un thread :
  - soit créer une classe dérivée de `Thread`, et redéfinir sa méthode `run()`
  - ou bien, créer une classe implantant l'interface `Runnable`, donc ayant une méthode `run()`, puis créer un `Thread` avec ce `Runnable` en paramètre du constructeur

Suspension/arrêt de thread

- Pour pouvoir suspendre/arrêter un thread, faire en sorte qu'il teste périodiquement un (ou des) drapeau(x) lui indiquant s'il doit être suspendu/redémarré/arrêté. Exemple :

```
private Thread t;
boolean suspendThread = false;
boolean stopThread = false;
public void run() {
  while ( !stopThread ) {
    if ( !suspendThread ) {
      // ...}
  }
}
public static void main(String[] args) {
  t = new Thread(this); t.start();
  // …
  suspendThread = true;  //suspension
  // …
  suspendThread = false; //redémarrage
  // …
  stopThread = true;     //arrêt définitif
}
```


Exemple de thread

```
class Compteur extends Thread {
  private int compteur = 0;
  private boolean pause = false;
  private boolean stop = false;
  /** Méthode lancée automatiquement au démarrage du thread par start()
   */

  public void run() {
    while (!stop) {
      if (!pause) compteur++;
      try {
        sleep(50);
      }
      catch (InterruptedException e) {}
    }
  }
  public int valeur(){
    return compteur;
  }
  public void suspendre()  { pause = true; }
  public void redemarrer() { pause = false;}
  public void tuer() { stop = true; }
}
```


Synchronisation des données

- gestion des conflits de modification de données par « lock » sur les méthodes déclarées `synchronized` :
  durant toute l'exécution par un thread d'une méthode synchronisée, aucun autre thread ne peut appeler simultanément une autre méthode synchronisée du même objet
- possibilité de faire un « bloc synchronisé » exigeant un lock sur un objet donné :

```
synchronized (obj) {
  //...
}
```

- si modification asynchrone d'un attribut, le déclarer `volatile` (pour forcer compilateur à rechercher valeur courante à chaque accès)

Synchronisation des exécutions

- attente de condition remplie : appel de la méthode `wait()` de l'objet sur lequel le thread travaille

```
class File {
  Element tete,queue;
  public synchronized Element suivant(){
    try{
      while (tete==null)
        wait();
    }
    catch(InterruptedException e) { return null; }
    return tete;
}
```

- déblocage de threads en attente sur l'objet courant : méthodes `notify()` et `notifyAll()`de l'objet

```
public synchronized arrivee(Element e){
  // ajout du nouvel element dans la file
  // ...
  notifyAll();
}
```


Synchronisation des exécutions (2)

- attente de fin d'un autre thread (rendez-vous) : appel de la méthode `join()` de ce thread

```
// Calcul est supposée implanter Runnable
Calcul c;
Thread calc = new Thread(c);
calc.start();
//...
try {
  // attente de la fin de calc
  calc.join();
} catch (InterruptedException e){
  // ...

}
```


Blocages

- la programmation avec threads nécessite de faire très attention à ce qu'aucun « deadlock » ne puisse se produire

- cas typiques de blocage :
  - t1 attend la fin de t2 et réciproquement
  - t1 attend la fin de t2 alors qu'il a un « lock » sur un objet sur lequel t2 attend de pouvoir mettre un « lock »
  - t1 suspend t2 pendant qu'il a un « lock » sur o, puis essaie de prendre un « lock » sur le même o

Priorités des threads

- plus sa priorité grande, plus le thread dispose d'une part importante du temps d'exécution (mais aucune garantie que supérieur strict)
- par défaut, `Thread.NORM_PRIORITY`
- `getPriority()` retourne la valeur
- `setPriority(int p)` avec p entre `Thread.MIN_PRIORITY` et `Thread.MAX_PRIORITY`
- `sleep(long milliseconds)` met en sommeil le thread courant
- `yield()` interrompt le thread courant pour permettre à un autre de prendre la main

Groupement de threads

- classe `ThreadGroup`
- permet de regrouper des threads que l'on veut pouvoir traiter de la même façon (par exemple les suspendre tous ensemble, modifier la priorité de tous en même temps, ...)

PAQUETAGES STANDARDS

Paquetages standards

- `java.lang` : bases
- `java.io` : entrée-sorties
- `java.util` : utilitaires, structures de données
- `java.text` : internationalisation
- `java.awt` : graphisme (Abstract Window Toolkit)
- `java.applet` : appliquettes
- `javax.swing` : graphisme « 2ème génération »
- `java.beans` : composants
- `java.rmi` : objets distribués (Remote Method Invocation)
- `java.net` : réseau
- `java.math` : précision arbitraire, ...
- `java.sql` : bases de données (JDBC)
- `java.security` : cryptage, authentification, …
- ...

Paquetage `java.lang`

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

- Nota-Bene : ce paquetage est « importé » automatiquement

classe `String`

- constructeurs : `String()`, `String(String)`, ...
- longueur : `length()`
- accès à un caractère : `char charAt(int)`
- recherche dans la chaine :
  `int indexOf(char), int indexOf(String)`
- comparaison de contenu (ordre lexico.) : `int compareTo(String)`
- test égalité de contenu : `boolean equals(Object)`
- test égalité partielle :

```
boolean regionMatches(
    int start, String other, int oStart, int len)
```

- test de début/fin de la chaîne :
  `boolean startsWith(String)`,
  `boolean endsWith(String)`,...

classe `String` (2)

- concaténation (renvoie une nouvelle chaîne avec le résultat) :
  `String concat(String)`
- sous-chaîne :
  `String substring(int deb, int fin)`
  `void getChars(int deb, int fin, char[] dest, int destDeb)`
- découpage :
  `String[] split(String delim)`
- suppression des blancs de début et fin :
  `String trim()`
- changement de classe :
  `String toLowerCase() ; String toUpperCase() ; ...`
- substitution de caractère :
  `String replace(char old, char new)`
- conversion en chaîne des types de base (méthodes statiques) :
  `String.valueOf(int) ; String.valueOf(float) ; ...`

classe `StringBuffer`

- constructeurs :
  `StringBuffer()`, `StringBuffer(String)`, `StringBuffer(int)`, ...
- concaténation :
  `StringBuffer append(String)`, `StringBuffer append(int)`, ...
- modification de caractère :
  `StringBuffer setCharAt(int index, char c)`
- insertion : `StringBuffer insert(int index, char c)`, ...
- suppression :
  `deleteCharAt(int index)`, `delete(int debut, int fin)`
- troncature : `void setLength(int)`
- miroir : `StringBuffer reverse()`
- conversion en String : `String toString()`
- capacité : `int capacity()`, ...
- comme String : `charAt(int)`, `substring()`, `getChars()`

Classe `StringBuilder`

- Variante de la classe `StringBuffer` (mêmes fonctionnalités), avec des implantations plus efficaces des méthodes, mais à ne pas utiliser en cas de multi-threading (méthodes non « synchronisées »)

Classe racine (`Object`)

- ancêtre de toutes les classes

- comparaison : `public boolean equals(Object obj)`
- conversion en chaîne : `public String toString()`
- duplication : `protected Object Clone()`
- code de hachage : `public int hashCode()`
- destruction : `protected void finalize()`
- synchronisation threads : `notify()`, `notifyAll()`, `wait()`
- classe instanciée : `public Class getClass()`

Classe Enum

- Classe abstraite ancêtre de tous les types énumérés (i.e. définis avec le mot-clef `enum`)
- Implante l'interface de comparaison `Comparable` (voir + loin), de sorte que toute valeur d'un type énuméré est automatiquement comparable à toute autre valeur d'un ***même*** type énuméré ; ***l'ordre « naturel » utilisé*** par la méthode `compareTo(Enum autre)` est ***l'ordre de déclaration des constantes*** dans le type énuméré

Classe Class

- représentation des classes et interfaces (et tableaux)
- obtention via une instance (`o.getClass()`), ou par le nom : `Class.forName("NomCompletClasse")`
- littéraux : `NomClasse.class`
  (ex: `String.class` désigne la classe `String`)
- instantiation : `Object newInstance()`
- récupération des membres de la classe
  - `Method[] getDeclaredMethods()`
  - méthodes publiques: `Method[] getMethods()`
  - `Field[] getDeclaredFields()` et `getFields()`
  - `Constructors[] getDeclaredConstructors()`, `getConstructors()`
  - `Class[] getInterfaces()`
  - `Class getSuperClass()`
  - classes internes : `Class[] getClasses()`

classe Class (2)

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

classe Field

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

classe `Method`

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

classes `Integer`, `Float`, ...

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

Auto-boxing / unboxing des types primitifs

- Les conversions dans les deux sens entre chaque type primitif et sa classe « enveloppe » correspondante se font ***automatiquement*** :

```
Integer objEntier = 4;
// OK : équivalent de = new Integer(4);

Double objDouble =
      Double.valueOf("1.618");
double x = 10.*objDouble;
// OK : équivalent de = 10.*objDouble.doubleValue();

Object[] tab= new Object[3];
tab[0] = true;
tab[1] = 'W';
tab[2] = 33;
// OK : équivalent de tab[0]= new Boolean(true);
// tab[1]= new Character('W');
// tab[2]= new Integer(33);
```


classe `Number` et classes filles

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

classes `Integer`, `Byte`, `Short`, `Long`

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

classes `Float` et `Double`

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

classe `Character`

- ne contient quasiment que des fonctions (méthodes statiques)
- test de la nature d'un caractère :
  `boolean Character.isLetter(char), Character.isDigit(char), Character.isWhiteSpace(char)`, ...
- Accès au type précis d'un caractère :
  `int Character.getType(char ch)` renvoie un identifiant parmi : `Character.LOWERCASE_LETTER, Character.DECIMAL_DIGIT_NUMBER`
  ...
- changement de casse :
  `char Character.toLowerCase(char)`, ...
- ...

classes `Throwable`, `Exception`, ...

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

classe System

- entrée-sorties standards :
  `InputStream System.in`,
  `PrintStream System.out` et `System.err`
- redirection des I/O : `System.setIn(InputStream)`, ...
- lancement forcé du Garbage Collector : `System.gc()`
- fin d'exécution : `System.exit(int status)`
- accès aux variables d'environnement :
  `String System.getProperty(Sring name)`...
- réglage sécurité :
  `System.setSecurityManager(SecurityManager)`
- heure (en ms depuis 1/1/1970) :
  `long System.currentTimeMillis()`
- copie rapide de tableau :
  `System.arraycopy(src,deb,dest,debD,len)`

classe Runtime

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

classe Process

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

classe Thread

- classe pour gérer les « sous-processus » internes à un programme Java
- constructeurs : `Thread(String)`, `Thread(Runnable)`
- thread courant : `Thread.currentThread()`
- démarrage : `start()`
- envoyer message : `interrupt()`
  => `isInterrupted()` devient `true` (à exploiter dans la méthode `run`)
- état : `boolean isAlive()`
  `== true` depuis `start()` jusqu'à fin d'exécution
- attente de la fin du thread t : `t.join()`
- contrôle du thread actif courant (méth. statiques) :
  - mise en sommeil : `Thread.sleep(long millisec)`
  - rendre la main : `Thread.yield()`
- infos: `getName()`,`getPriority()`,`getThreadGroup()`
- priorité : `setPriority(int p)`, p entre `Thread.MIN_PRIORITY` et `Thread.MAX_PRIORITY`

classe `Math`

- Constantes : `Math.PI`, `Math.E`
- Fonctions mathématiques usuelles (toutes *static*, avec `double` en entrée et en sortie) :
  `sin()`, `cos()`, `tan()`, `acos()`, `... sqrt()`, `exp()`, `log()`, `pow()` , `ceil()`, `floor()`, `rint()`, `...`
- Autres fonctions :
  - `int round(float)`, `long round(double)`: arrondis à l'entier le plus proche
  - `abs(a)`, `min(a,b)`, `max(a,b)` pour a et b : int, long, float et double
  - `double random()` : nombre compris dans l'intervalle [0. … 1. [et tiré – par défaut – d'une série pseudo-aléatoire nouvelle à chaque exécution du programme

classe `StrictMath`

- Identique à la classe `Math` en termes de fonctionnalités mais avec des caractéristiques différentes en termes de performances et de portabilité :
  - utilise des algorithmes standards définis dans la bibliothèque mathématique **fdlibm** « *Freely Distributable Math Library* »
  - résultats des calculs strictement identiques quelque soit le matériel utilisé
  - performances *possiblement* inférieures comparées à celles de la classe `Math` (lorsque le hardware utilisé fourni des routines optimisées).

Interfaces diverses

- `Appendable` : implantée par classes prédéfinies permettant l'append de caractères (`StringBuffer`, `FileWriter`, …)
- `CharSequence`
- `Cloneable` : voir description de méthode `clone()` dans section sur classes/objets
- `Comparable` : interface de comparaison (voir section sur `java.util`)
- `Iterable<T>` : interface déclarant l'unique méthode `Iterator<T> iterator()`, et caractérisant les types permettant les « itérations simplifiées » (voir section sur les instructions de contrôle)
- `Readable` : source de caractères (implantée notamment par `FileReader`, `StringReader`, …)
- `Runnable` : voir section sur les thread

ENTREES-SORTIES :

paquetage `java.io`

- lecture/écriture séquentielle (flux)
- gestion fichiers

Les 4 catégories de flux

- flux d'octets : classes abstraites **`InputStream`** et **`OutputStream`**
- flux de caractères : classes abstraites **`Reader`** et **`Writer`**
- principales méthodes de lecture :
  - `int read()` : prochain élément du flux (ou -1 si « fin de flux »)
  - `int read(byte[] buf)` pour `InputStream` et `int read(char[] buf)` pour `Reader`
  - `long skip(long nb)` : saute nb éléments
  - `void close()`
- principales méthodes d'écriture :
  - `void write(int c)`
  - `void write (byte[] / char[] buf)`
  - `pour Writer : void write(String s)`
  - `void flush()`
  - `void close()`

Les flux « concrets »

- lecture/écriture (séquentielle) dans fichiers :
  `FileInputStream/FileOutputStream`,
  `FileReader/FileWriter`
  nom du fichier spécifié dans le constructeur : `new FileReader(filename)`
- lecture/écriture dans tableau en mémoire :
  `ByteArrayInput(/Output)Stream`
  `CharArrayReader/CharArrayWriter`
  tableau donné dans le constructeur : `new CharArrayReader(tab)`,
  récupéré par `toByteArray()` (resp. `toCharArray()`)
- lecture/écriture dans une chaîne :
  `StringReader/StringWriter`
  chaîne donnée dans le constructeur (`new StringReader(ch)`),
  ou récupérée par `toString()`
- enchaînements de flux (« pipes ») :
  `PipedInputStream/PipedOutputstream`,
  `PipedReader/PipedWriter`

Conversions pour flux binaire

- pour lire/écrire *autre chose* que des octets ou caractères, on utilise des classes avec d'autres méthodes, qui font la conversion avec les flux « concrets » de bas niveau :
- flux données binaires (types primitifs) :
  `DataInputStream`/`DataOutputStream`
  - couche au-dessus d'un flux d'octets (`new DataOutputStream(outStream)`, ...)
  - méthodes de lecture : `readFloat()`, `readInt()`,... et écriture : `writeFloat(x)`, `writeInt(i)`,...
- flux données binaires (objets persistants) :
  `ObjectInputStream` / `ObjectOutputStream`
  - couche au-dessus d'un flux d'octets
  - mêmes méthodes que `DataXXXStream` + `Object readObject()` (resp. `writeObject(o)`)
  - objets doivent être de classe implantant l'interface `Serializable`

Autres conversions de flux

- flux « avec tampon » (pour éviter des accès disque ou réseau à chaque lecture/écriture) :
  `BufferedInputStream / BufferedOutputStream`,
  `BufferedReader / BufferredWriter`
- flux pour écriture formatée en mode texte (comme pratiqué sur `System.out`) : classe `PrintWriter`
  avec comme méthodes utiles : `print()`, `println()`, `printf()` ...
- flux compressés : paquetage `java.util.zip`
  (classes `ZipInputStream`, `ZipOutputStream`,...)
- conversion flux octets / flux caractères :
  `InputStreamReader/OutputStreamWriter`

En pratique : écriture d'un fichier « texte »

1/ Importer les classes nécessaires :
```
import java.io.*;
```
2/ Ouvrir un flux de caractères vers un fichier :
```
FileWriter out = new FileWriter("nomFichier");
```
3/ Ecrire les caractères et/ou chaînes de caractères sur ce flux :
```
out.write(ch); // ch peut être :
 //                  - un char
 //                  - une String
```
4/ Fermeture du flux :
```
out.close();
```
Remarque : chaque étape (sauf l'import) est susceptible de lancer une `IOException` donc mettre les instructions à risque dans un bloc `try/catch` :
```
try {
   //...
}
catch(IOException e) {
   //...
}
```
(ou bien, utiliser une clause `throws`) .

En pratique : lecture d'un fichier « texte »

1/ Importer les classes nécessaires :
```
import java.io.*;
```
2/ Ouvrir un flux de caractères depuis un fichier :
```
FileReader in = new FileReader("nomFichier");
```
3/ Lire caractère par caractère sur ce flux :
```
ch = in.read();
// ch vaut -1 en fin de fichier
```
4/ Fermeture du flux :
```
in.close();
```
Remarques :

- Chaque étape (sauf l'import) peut lancer une `IOException` ➔ bloc `try/catch` (ou `throws`) .
- Pour lire des nombres ou toutes donnés structurées écrites sous forme texte, utiliser la classe `Scanner` :

```
import java.util.Scanner;
Scanner sc = new Scanner(in);
double val = sc.nextDouble();
...
```


En pratique : écriture d'un fichier « binaire »

1/ Importer : `import java.io.*;`

2/ Ouvrir un flux d'octets vers un fichier :
```
FileOutputStream out =
                    new FileOutputStream("nomFichier");
```
3/ Brancher un flux d'écriture d'objets sur le flux d'octets :
```
ObjectOutputStream objOut =
                      new ObjectOutputStream(out);
```
4/ Ecrire les nombres et/ou objets sur ce flux :
```
objOut.writeInt(i);   // i entier
objOut.writeFloat(x); // x flottant
// ...
objOut.writeObject(o);
// o objet d'une classe qui implante l'interface Serializable
```
5/ Fermeture des flux : `objOut.close();`

Remarque : chaque étape (sauf l'import) peut lancer une `IOException` ➔ bloc `try/catch` (ou clause `throws`) .

En pratique : lecture d'un fichier « binaire »

1/ Importer : `import java.io.*;`

2/ Ouvrir un flux d'octets depuis un fichier :
```
FileInputStream in = new FileInputStream("nomFichier");
```

3/ Brancher un flux de lecture d'objets et de données sur le flux d'octets :
```
ObjectInputStream objIn = new ObjectInputStream(in);
```

4/ Lire les nombres et/ou objets sur ce flux :
```
int i = objIn.readInt();
float x = objIn.readFloat();
// …
Object o = objIn.readObject();
```

5/ Fermeture des flux : `objIn.close();`

Remarques :

- Chaque étape (sauf l'import) peut lancer une `IOException` ➔ bloc `try/catch` (ou `throws`).
- Penser à convertir ce que retourne `readObject()` dans le bon type (celui de l'objet écrit à cet endroit du fichier).

Lecture/ecriture simultanées et non-séquentielles sur fichier

- Possible via classe **`RandomAccessFile`**
- constructeur : `RandomAccessFile(filename,mode)`
  `mode` valant `"r"` (readonly) ou `"rw"` (read/write)
- écrire sur le fichier :
```
raf.writeInt(i);  // i entier
raf.writeFloat(x);// x flottant
raf.writeUTF(s);  // s String
//…
```
- lire sur le fichier :
```
int i = raf.readInt();
float x = raf.readFloat();
String s = raf.readUTF();
String l = raf.readLine();
//…
```
- taille : `int len = raf.length()`
- déplacement dans le fichier : `raf.seek(pos);`

Classes utilitaires pour entrées-sorties

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

- **`File`** : pour manipuler fichiers/répertoires (accès à la taille, listage du contenu, suppression, renommage, …)
  - constructeurs :
    `File(String name), File(String path, String name), …`
  - méthodes :
    - `boolean exists(), long length(), File getParent()`
    - `boolean isDirectory(), String[] list()`
    - `void delete(), void mkdir(),`
    - `boolean renameTo(File f), ...`

Paquetage `java.util` :

COLLECTIONS, DATES, ...

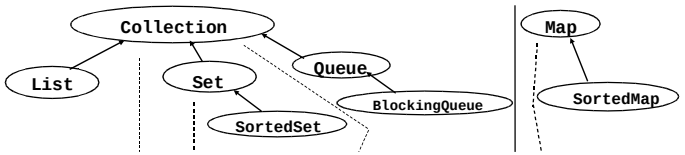
- Ensembles, Listes, Piles, Tables de hachage,...
- interface d'Itérateurs
- gestion des dates/heures
- internationalisation

Les collections

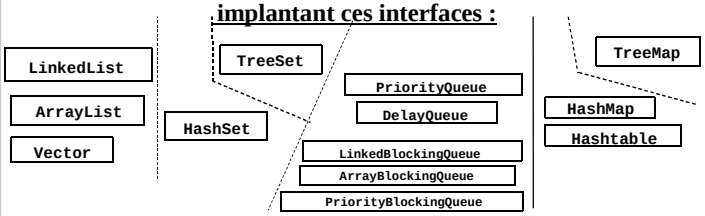
- une collection = un objet regroupant plusieurs éléments (ex. : tableau, liste, …)
- Java propose, en plus des tableaux, plusieurs types de collections :
  - ensemble : aucun élément dupliqué, pas d'ordre intrinsèque
  - liste : collection ordonnée, avec répétition possible du même élément
  - table : série de couples (clef –> valeur)
  - …
- Depuis la version 1.5, toutes les classes et interfaces de ces collections sont « generics », ce qui permet de définir et manipuler des collections d'objets ***d'un type donné***

Les principales collections Java

Principales interfaces :



Principales classes implantant ces interfaces :



L'interface `Collection`

- L' interface `Collection` *(ou plutôt l'interface* `Collection<E>`*, où* `E` *est le type des éléments)* permet de manipuler toutes les collections de façon unifiée, et regroupe les méthodes communes à toutes les collections :
  - taille : `int size()`
  - test si vide : `boolean isEmpty()`
  - ajout : `boolean add(E element)`
    [Note : quand `Collection` n'est pas paramétrée par `<E>` , le prototype est `boolean add(Object element)`]
  - suppression : `boolean remove(Object elt)`
  - suppression de tous les éléments : `void clear()`
  - recherche : `boolean contains(Object elmt)`
  - parcours : `Iterator<E> iterator()`
  - conversion en tableau : `Object[] toArray()`
  - …

L'interface Iterator

- Permet de parcourir n'importe quelle collection
- habituellement, paramétrée par le type `E` des éléments de la collection : `Iterator<E>`
- s'obtient par appel de `iterator()` sur la collection à parcourir
- contient les méthodes :
  - `boolean hasNext()`
  - `E next()`
    [ou `Object next()` quand Iterator n'est pas paramétrée par `<E>`]
  - `void remove()`
- **ATTENTION :** il faut en recréer un nouveau (par appel de `iterator()`) dès que la collection a été modifiée par appel à `add()`, `remove()`,… directement sur la collection

Fonctions utilitaires pour les `Collection` (et les tableaux)

- La classe `Collections` contient toutes sortes de fonctions permettant d'effectuer des manipulations courantes sur les `Collection` :
  - tri avec les fonctions `sort()`
  - recherche du plus petit/grand élément avec les fonctions `min()` et `max()`
  - inversion de l'ordre d'une `List` (fonction `reverse()`)
  - réarrangement aléatoire d'une `List` (fonction `shuffle()`)
  - transformation en `Collection` non modifiable
  - …
- La classe `Arrays` contient le même genre de fonctions pour les tableaux : `binarySearch()`, `sort()`, `fill()`, …

Interfaces de comparaison : `Comparable` & `Comparator`

- L'interface `Comparable<T>` (du paquetage `java.lang`) spécifie que toute instance de la classe qui l'implante peut être comparée à toute instance de la classe `T` ; elle déclare une seule méthode :

  ```
  public int compareTo(T autre);
  ```

  (<0 si `this` "<" `autre`, >0 si `this` ">" `autre`, ou 0 si les deux sont "égaux")

  Si `MaClasse` implante `Comparable<MaClasse>`, ceci permet notamment de trier un `MaClasse[] tab` avec `Arrays.sort(tab)` ou encore de trier une `Collection<MaClasse> coll` avec `Collections.sort(coll)`

- L'interface `Comparator<T>` (du paquetage `java.util`) déclare principalement :

  ```
  public int compare(T t1, T t2);
  ```

  (négatif, positif, ou nul selon l'ordre `t1 t2`)

  elle permet de définir des « comparateurs d'instances » de la classe T, notamment pour `Arrays.sort(T[],Comparator)` et `Collections.sort(List<T>,Comparator)`

Les listes (interface `List`)

- *Habituellement paramétrée par le type `E` des éléments de la liste : `List<E>`*
- idem que `Collection`, avec en plus des méthodes pour l'accès par position, et itérateur spécifique :
  - `E get(int index)`
  - `E set(int index, E element)`
  - `void add(int index, E element)`
  - `E remove(int index)`
  - `int indexOf(Object elt)`
  - `ListIterator<E> listIterator(int debut)`
  - `ListIterator<E> listIterator()`
  - …

  [remplacer `E` par `Object` quand `List` n'est pas paramétrée par `<E>`]
- trois implantations de l'interface `List<E>` :
  - classe `LinkedList<E>`
  - classe `ArrayList<E>`
  - classe `Vector<E>` (implantation similaire à `ArrayList`)

L'interface `ListIterator`

- permet de parcourir une `List` (*habituellement paramétrée par le type `E` des éléments de la liste : `ListIterator<E>`)*
- s'obtient par appel de `listIterator()` sur la liste à parcourir
- étend l'interface `Iterator<E>` en ajoutant des méthodes :
  - pour parcourir en sens inverse (`previous()`, `hasPrevious()`, …)
  - pour insérer (`add(E o)`)
  - pour modifier (`set(E o)`)
  - …
- **ATTENTION :** comme pour les `Iterator`, il faut en recréer un nouveau (par appel de `listIterator()`) dès que la liste a été modifiée par appel à `add()`/`remove()`/… directement sur la liste

Exemple d'utilisation de liste [sans paramétrage de type]

```
import java.util.*;
public class ExempleList {
  public static void main(String[] args) {
    List list; // DECLARATION D'UNE LISTE(sans préciser implantation)
    list = new LinkedList();   //  CREATION D'UNE LISTE *CHAINEE*
    list.add("Paul");  // INSERTION D'UN OBJET DANS LA LISTE (A LA FIN)
    list.add(0, "Léa");   // INSERTION D'UN OBJET EN *DEBUT* DE LISTE
    list.add("Julie");    // INSERTION D'UN OBJET EN *FIN* DE LISTE
    System.out.println(list);  // AFFICHAGE DE LA LISTE
    int n = list.size();       // ACCES A LA TAILLE DE LA LISTE
    System.out.println("Nombre d'elements = " + n);
    list.set(1, "Marc");       // REMPLACEMENT D'UN ELEMENT DE LA LISTE
    // ACCES A UN DES ELEMENTS DE LA LISTE :
    String nom = (String) list.get(1);   // ->"Marc"
    System.out.println(nom);
    // SUPPRESSION D'UN ELEMENT DE LA LISTE :
    list.remove(1);   // retire "Marc"
    // ...
```


Exemple d'utilisation de liste [sans paramétrage de type] (suite)

```
  // ...
  // PARCOURS DE LA LISTE AVEC UN ITERATEUR
  ListIterator iter = list.listIterator();
  while (iter.hasNext()){
    // TANT QU'IL RESTE UN ELEMENT, ACCES A L'ELEMENT SUIVANT
    String s = (String) (iter.next());
    // (penser à convertir dans le bon type)
    System.out.println(s);
    if (s.equals("Léa")) {
      iter.add("Jean");  // ajout durant le parcours
    }
  }

  String s = (String) iter.previous();  // retour en arriere
  iter.remove();           // suppression durant le parcours

  list.add("Alain");   // modif sans passer par l'iterateur !
  System.out.println(list);

  // ATTENTION, après modif, l'itérateur n'est plus valide :
  System.out.println(iter.next());
  // -> ConcurrentModificationException
```


Utilisation de liste en tirant parti de la généricité

- Si le type des éléments est uniforme, mieux vaut utiliser les collections en mode « *générique* »
- Dans l'exemple précédent, il faut donc écrire :
  - déclaration : `List<String> list;`
    (on précise le type des éléments de la liste)
  - création : `list = new LinkedList<String>();`
  - accès à un élément : `String nom = list.get(1);`
    (plus besoin de convertir ici, car le type de retour de `get()` est adapté à celui des éléments)
  - création itérateur :
    `ListIterator<String> iter = list.listIterator();`
  - utilisation itérateur : ***type de retour de `iter.next()` et `iter.previous()` adaptés ➔ plus besoin de conversion des références***
    `(String s = iter.next();)`

Types et classes des collections paramétrées

- Remarque préalable : ***les collections ne peuvent contenir que des objets (pas de types primitifs)*** ; à part cela, le type E des éléments peut-être n'importe quelle ***classe ou interface*** (éventuellement paramétrée)
- Les types `ArrayList<String>` et `ArrayList<Integer>` (par exemple) sont ***distincts et incompatibles*** (bien qu'il n'y ait en fait qu'une unique classe `ArrayList`)
- ATTENTION : il est impossible de créer des tableaux dont les éléments sont d'un type paramétré (`new ArrayList<String>[3]` est illégal !)
  ➔ utiliser à la place des collections de collections, e.g. :

```
List<List<String>> lili ;
lili = new ArrayList<List<String>>();
```


Références vers collections de divers types

- On peut déclarer et utiliser des références capables de pointer vers les diverses variantes d'une même classe ou interface : ainsi, ***`List<?> list;` peut pointer vers toute liste indépendamment du type des éléments***
- On peut aussi restreindre les types possibles d'éléments :
  `List<? extends Truc> li;` (où `Truc` est une classe ou interface)
  puis `li = new ArrayList<Machin>();`
  avec `Machin extends` (ou `implements`) `Truc`
- **ATTENTION :** ces références de type `Xyz<?>` ou `Xyz<? extends Truc>` ne permettent PAS l'utilisation des méthodes de `Xyz` pour lesquels un des paramètres est de type `E` (ou `<E>` est le paramètre de type de `Xyz`)…

Classe `Vector`

- *Habituellement paramétrée par le type `E` de seséléments : `Vector<E>`*
- Similaire à la classe ArrayList en termes de fonctionnalités
- Principale différence : Vector est « synchronized »

Classe `Stack`

- *Habituellement paramétrée par le type `E` de ses éléments : `Stack<E>`*
- Implantation par dérivation de la classe `Vector`, en ajoutant les méthodes :
  - empiler : `push(E item)`
  - sommet : `E peek()`
  - dépiler : `E pop()`

[remplacer `E` par `Object` quand `Stack` n'est pas paramétrée par `<E>`]

Les ensembles (interfaces Set et SortedSet)

- `Set<E>` : exactement les mêmes méthodes que `Collection<E>`, mais impossible de mettre deux fois le même élément dans un même `Set`
- `SortedSet<E>` y ajoute des méthodes pour :
  - accéder au plus grand ou au plus petit : `last()`, `first()`
  - extraire un sous-ensemble : `subSet(E min, E max)`, ...

- Deux implantations :
  - classe `HashSet<E>` pour `Set<E>` (utilisant une table de hachage `HashMap<E,Object>`)
  - classe `TreeSet<E>` pour `SortedSet<E>` (utilisant un « arbre bicolore » `TreeMap<E,Object>`)

- Opérations sur les ensembles :
  - Test d'inclusion : `e1.containsAll(e2)`
  - Union : `e1.addAll(e2)` [modifie `e1` !]
  - Intersection: `e1.retainAll(e2)` [modifie `e1` !]

Les files d'attente (interface Queue)

- Cette nouvelle sous-interface `Queue<E>` de `Collection<E>` offre les fonctionnalités d'une file d'attente.

- En plus des méthodes de `Collection`, on trouve :
  - essai d'ajout : `boolean offer(E elt)` : [idem `add(elt)`, sauf que la réussite de l'ajout dans la file n'est pas garanti (cas d'une file pleine…)]
  - récupération et suppression de l'élément en tête (si file non vide) : `E poll()`
  - `E remove()` : idem `poll()`, mais lance `NoSuchElementException` si la file est vide
  - accès à la tête de file : `E peek()`
  - `E element()` : idem `peek()` mais lance `NoSuchElementException` si la file est vide

- La sous-interface `BlockingQueue<E>` bloque jusqu'à ce que la file soit non-vide, en cas d'essai d'accès à la tête.

Implantations de Queue

- **`PriorityQueue<E>`** : les éléments sont traités selon leur ordre « naturel »

Autres implantations, dans le sous-paquetage `java.util.concurrent` :

- **`LinkedBlockingQueue<E>`** : une FIFO (« First In First Out ») bloquante.
- **`ArrayBlockingQueue<E>`** : une FIFO (« First In First Out ») bloquante à capacité limitée (stockage dans un tableau).
- **`PriorityBlockingQueue<E>`** : idem `PriorityQueue<E>` mais en version bloquante
- **`DelayQueue<E>`** : les éléments doivent implanter l'interface `Delayed`, et ne sont traités qu'une fois leur « délai » expiré, en commençant par ceux ayant expiré les premiers

Les tables (interface Map)

- Série de couples (clef –> valeur)
  - toute clef est présente au plus une fois
  - une clef présente est associée à une **unique** valeur

  [*paramétrée par les types respectifs K et V des clefs et des valeurs :* `Map<K,V>`]

- Principales méthodes :
  - ajout : `put(K clef, V valeur)`
  - valeur associée à une clef : `V get(Object clef)`
  - suppression : `V remove(Object clef)`
  - taille : `boolean isEmpty()` et `int size()`
  - `boolean containsKey(Object clef)`
  - `boolean containsValue(Object val)`
  - collection des valeurs : `Collection<V> values()`
  - ensemble des clefs : `Set<K> keySet()`
  - ensemble des couples clef-valeur : `Set<Map.Entry<K,V>> entrySet()`
    (chaque couple est une instance de `Map.Entry<K,V>`)

Table triée : interface SortedMap

- `SortedMap<K,V>` est un cas particulier de `Map<K,V>` qui garantit que les couples sont stockés ***par ordre croissant de clef***

- Méthodes supplémentaires par rapport à `Map` :
  - `K firstKey()`
  - `K lastKey()`
  - `SortedMap headMap(K clefLimite)`
  - `SortedMap tailMap(K clefLimite)`
  - `SortedMap subMap(K clefMin, K clefMax)`

[remplacer `K` et `V` par `Object` dans les prototypes quand `SortedMap` n'est pas paramétrée par `<K,V>`]

Principales implantations de table

- Deux implantations principales :
  - `HashMap<K,V>` = implantation de `Map<K,V>` par hachage
  - `TreeMap<K,V>` = implantation de `SortedMap<K,V>` par « arbre bicolore »

- **NOTE :** Il existe aussi une autre implantation de `Map`, très voisine de `HashMap` : la classe `Hashtable`. En plus des méthodes standards de `Map`, elle possède les méthodes suivantes :
  - liste des éléments : `Enumeration elements()`
  - liste des clef : `Enumeration keys()`

Collections et enum

- La classe **`EnumSet`**`<E extends Enum<E>>` est une implantation de `Set` spécialisée (et optimisée) pour le ***stockage de valeurs d'un même type énuméré*** (`enum`)
  - pas de constructeur : la création se fait par une des méthodes « static » de la classe ; exemple :
    ```
    Set<MonEnum> s;
    s = EnumSet.allOf(MonEnum.class);
    ```
  - Manipulation avec les méthodes standard d'un `Set`

- `EnumMap<K extends Enum<K>,V>` est une classe offrant une implantation de `Map` spécialisée et optimisée pour des tables dont les ***clefs*** sont des valeurs d'un même type énuméré.

Enumeration

- `Enumeration<E>` contient deux méthodes :
  - élément suivant : `E nextElement()`
  - test de fin : `boolean hasMoreElements()`

« parsing » : classes Scanner et Pattern

- Un `Scanner` permet de « parser » tout ce qui est « Readable » (flux, Strings, etc.) :

```
import java.util.Scanner;
// . . .
Scanner scFlux = new Scanner(System.in);
String tst = "bleu 12 rouge 1,618";
Scanner scString = new Scanner(tst);
String s = scString.next();
int k = scString.nextInt();
double x = scFlux.nextDouble();
if (scFlux.hasNext()) { /* ... */ }
if (scFlux.hasNextDouble()) { /* ... */ }
```

- La classe `Pattern` permet de définir et rechercher des « expressions régulières » :

```
if ( Pattern.matches("[a-d].*[0-9]", chaine) ) { /* ... */ }
Pattern pat = Pattern.compile("[^aeiouy]*\s");
```

- Usage conjoint avec `Scanner` :

```
if (scFlux.hasNext(pat)){ /* ... */ }
```


classe Scanner : exemples

- Lecture depuis le clavier (sans gestion d'erreurs) :

```
Scanner scin = new Scanner(System.in);
System.out.println("Entrez une chaine : ");
String s = scin.next();
System.out.println("Entrez un entier : ");
int n = scin.nextInt();
System.out.println("s=" + s + ", n=" + n);
```

- Lecture d'un fichier ligne par ligne et affichage de son contenu à l'écran :

```
try {
  Scanner sc = new Scanner( new FileReaderfichier.txt") );
  String ligne;
  while (sc.hasNextLine()) {
    ligne = sc.nextLine();
    System.out.println(ligne);
  }
}
catch (FileNotFoundException e) {
  System.err.println(e);
}
```


classe Scanner : exemple

- Lecture depuis le clavier avec pattern et gestion d'erreurs :

```
Scanner sc = new Scanner(System.in);

Pattern p = Pattern.compile("[Yy]"));

try {
  String s1 = sc.next(p);
     // accepte uniquement Y ou y
}
catch(InputMismatchException e){
  System.out.println("Y ou y attendu");
}

. . .
```


Divers : BitSet, StringTokenizer, Random

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

- Découpage de chaîne en tronçons : classe `StringTokenizer`
  - constructeur : `StringTokenizer(String aDecouper)`, …
  - `boolean hasNextToken()`
  - `String nextToken();`
  - `String nextToken(String delim);`
    [`delim` : chaîne contenant l'ensemble des délimiteurs à prendre en compte]

- Classe `Random` : pour la génération de séries (pseudo-)aléatoires de nombres

Dates, heure, etc…

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

INTERFACES HOMME-MACHINE (IHM) GRAPHIQUES

IHM graphiques

- Java inclut diverses classes facilitant la création d'Interfaces Homme-Machine (IHM) graphiques PORTABLES
- cette IHM peut être :
  - soit une applet intégrée dans une page Web,
  - soit une application indépendante
- principaux paquetages :
  - `java.awt` : bases communes et composants de « 1ère génération »
  - `java.awt.event` : gestion des événements
  - `java.applet` : applets non Swing
  - `javax.swing` : complément et alternative aux composants AWT

Applet v.s. application

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

tag HTML <APPLET>

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

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

où `w` et `h` = dimension réservée à l'applet (en pixels)

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

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

pour passer des paramètres récupérables dans l'applet

Applets, versions de Java et navigateurs

- Les navigateurs usuels (Firefox, IE) utilisent pour exécuter les applets leur propre JVM qui peut correspondre à une version plus ou moins récente du langage
- Solution #1 : ne pas utiliser dans les applets les nouveautés les plus récentes de Java
- Solution #2 : exiger la présence d'un plug-in Java dans le navigateur :
  - côté utilisateur, il faut donc installer un « Java Plug-in » dans le navigateur ;
  - côté développeur, faire en sorte que la référence à l'applet dans le fichier HTML utilise la balise <OBJECT> au lieu de la balise <APPLET>

Ecrire une Applet

Pas de `main`(), mais une sous-classe de `Applet`, redéfinissant tout ou partie des méthodes suivantes (appelées par le browser, ou par `appletviewer`) :

- `init()` : exécutée au chargement de l'applet
- `start()` : exécutée au démarrage de l'applet
  [doit rendre la main => lancer des threads si besoin d'une boucle continue]
- `paint(Graphics)` : dessine l'applet (est normalement appelée automatiquement par `update()` quand nécessaire)
- `update(Graphics)` : appelée automatiquement quand il faut re-dessiner l'applet (par défaut : efface, puis appelle `paint()`)
- `print(Graphics)` : pour imprimer, si différent de affichage
- `stop()` : exécutée quand l'applet devient temporairement invisible (=> suspendre les threads, ...)
- `destroy()` : exécutée au déchargement de l'applet (=> libérer les ressources)

Ecrire une Applet (2)

- méthodes essentielles souvent utilisées :
  - `repaint()` : demande de ré-affichage (exécutée par le système dès que possible)
  - `Image getImage(URL base, String name)`: chargement en mémoire d'un fichier image
  - `AudioClip getAudioClip(Url base, String name)` : chargement d'un fichier son
  - `URL getDocumentBase()` : récupération de l'URL du document web incluant l'applet
  - `URL getCodeBase()` : récupération de l'URL du répertoire de l'applet
  - `String getParameter(String)` : pour récupérer valeur d'un des <PARAM>
- affichage d'une image : `g.drawImage(img, x,y,this);`
  [dans la méthode `paint(Graphics g)`]
- utilisation d'un son `Audioclip ac` :
  `ac.play(), ac.loop(), ac.stop()`

Une applet élémentaire

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

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

Une applet avec image et son

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

```
import java.applet.*;
import java.awt.*;
import java.net.*;
      // pour la classe URL

public class ImageSonApplet
         extends Applet {
  private Image img;
  private int xIm, yIm;
  private AudioClip son;

  // Chargement image et son :
  public void init() {
  URL base = getDocumentBase();
  img=getImage(base, "im.gif");
  son=getAudioClip(base, "s.au");
  xIm=10; yIm=20;
  }

  // Jouer le son :
  public void start(){
    son.loop();
  }

  // Afficher image :
  public void paint(Graphics g){
    g.drawImage
        (img,xIm,yIm,this);
  }

  // Arrêter son, modifier la
  // position :
  public void stop() {
    son.stop();
    xIm+=20; yIm+=20;
  }
}
```


Une applet animée simple (sans thread)

Affiche un disque de rayon qui varie en permanence (croît/décroit/croît/…)

```
import java.applet.*;
import java.awt.*;
public class AnimApplet extends Applet {
  private int x=50, y=50, r=5, dr=1;
  private final int MAX_DIM = 40;
  public void paint(Graphics g) {
    g.fillOval(x, y, 2*r, 2*r);
    iteration();
    repaint(); // demande le ré-affichage du dessin
  }
  public void iteration() {  // effectue une itération
    if ( r>MAX_DIM || r<1 ) {
      dr *= -1; // changement du sens de variation
    }
    r += dr;    // modification du rayon
 }
}
```


Une applet animée par un thread

Affiche un disque de rayon variable

```
import java.applet.*;
import java.awt.*;
public class AnimApplet2 extends Applet implements Runnable {
  private int x=50, y=50, dt=100, r=5, dr=1;
  private final int MAX_DIM = 40;
  private Thread anim;
  private boolean suspendAnim=false;
  public void init(){ anim=new Thread(this); }
  public void start() {
    if (!anim.isAlive()) anim.start(); else suspendAnim=false;
  }
  public void paint(Graphics g) { g.fillOval(x,y,2*r,2*r); }
  public void run() {  // corps du thread d'animation
    while (true) {
      if (!suspendAnim) { if (r>MAX_DIM||r<1) dr*=-1; r+=dr; }
      try { Thread.sleep(dt); }
      catch(InterruptedException e){}
      repaint();
     }
  }
  public void stop() { suspendAnim=true; }
}
```

Paquetage `java.awt` (et sous-paquetages)

- composants graphiques élémentaires (boutons, menus, ...) de première génération
- conteneurs (fenêtres, ...) de première génération
- primitives graphiques pour dessiner
- gestion des images et des sons
- gestion des événements

Composants graphiques élémentaires

- Briques de base à assembler pour fabriquer l'IHM
- Classe abstraite `Component` dont héritent les sous-classes :
  - `Button` : bouton
  - `Label` : texte affiché par le programme
  - `TextField` et `TextArea` : pour entrer du texte
  - `Checkbox` (et `CheckboxGroup`) : case à cocher/décocher
    [éventuellement choix parmi un ensemble mutuellement exclusif]
  - `Choice` : menu déroulant permettant un choix exclusif entre les valeurs proposées
  - `List` : ensemble de choix cumulables, présentés dans une fenêtre scrollable
  - `MenuBar, Menu, MenuItem` : pour faire menus déroulants usuels
  - `PopupMenu` : menu « contextuel » lié à un autre composant au-dessus duquel il peut apparaître
  - `ScrollBar` : barre de défilement
  - `Canvas` : zone pour dessiner, ou base pour créer de nouveaux composants par dérivation
  - `Container` (classe abstraite) : pour grouper des composants selon un certain positionnement

Types de Conteneurs

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

Placement dans les conteneurs

- A chaque conteneur est associé un `LayoutManager` qui définit la façon de positionner ses composants
- Interface `LayoutManager`, implantée par les classes :
  - `FlowLayout` : de gauche à droite et de haut en bas, dans ordre d'ajout (ou en fonction de position demandée)
  - `GridLayout` : nb. de lignes et colonnes définis à création, puis idem `FlowLayout`, sauf nb. de colonnes reste fixe lors de redimensionnements (=> composants retaillés en conséquence)
  - `BorderLayout` : 5 composants seulement, placés en haut, bas, droite, gauche ou centre (cf. constantes de classe `NORTH, SOUTH, EAST, WEST, CENTER`)
  - `CardLayout` : superposé
  - `GridBagLayout` : basé grille, mais très paramétrable via `GridBagConstraints`

Fonctionnement des conteneurs

- Gestion des composants :
  - `add(Component c)` : ajout en première place vide (au centre pour `BorderLayout`)
  - `add(Component c, Object contrainte)` : ajout en précisant une contrainte (`BorderLayout.NORTH` pour container en mode `BorderLayout`, ...)
  - `add(Component c, int pos)` : ajout à la position pos entre 0 et la place du dernier composant +1 (ne marche pas pour BorderLayout)
  - `int getComponentCount()` : nb. de composants
  - `Component [] getComponents()` : tableau des composants
  - `Component getComponent(int pos)` : composant situé à la position pos
  - `void remove(int pos)` : retrait du composant situé à position pos
  - `void removeAll()`
  - ...
- Gestion du type de placement :
  - void `setLayout(LayoutManager)`
  - `LayoutManager getLayout()`
  - ...

Les Menus

- créer une `MenuBar mb`
- l'insérer dans un `Frame f` :
  `f.setMenuBar(mb);`
- créer les menus `m` et les ajouter à la `MenuBar` :
  `mb.add(m);`
- créer les `CheckboxMenuItem` et/ou `MenuItem` et les ajouter aux menus

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

Dessiner

- Pour dessiner sur un composant, il faut un « contexte graphique » (objet `Graphics`) associé
- on obtient une copie du `Graphics` courant pour un composant par appel de sa méthode `getGraphics()` [possible a priori uniquement pour les conteneurs et les composants de type `Canvas`]
- méthodes de `Graphics` pour dessiner :
  - `drawString(String texte, int x, int y)`
  - `drawLine(int x1, int y1, int x2, int y2)`
  - `drawRect(int x, int y, int l, int h)`
  - `drawOval(int x, int y, int l, int h)`
  - `drawArc(int x, int y, int l, int h, int angleDeb, int angleArc)`
  - `drawPolygone(int[] tabX, int[] tabY,int nbPoints)` et `drawPolygon(Polygone)`
  - `drawPolyline(int[] tabX, int[] tabY, int nbPoints)`
  - Aussi, méthodes de remplissage associées : `fillRect(), fillArc(), ...`

Couleurs

- les dessins sur un `Graphics` se font avec la « couleur courante », consultable et modifiable par ses méthodes respectives :
  - `Color getColor()`
  - `void setColor(Color)`
- les couleurs d'avant-plan et d'arrière-plan d'un composant sont aussi modifiables via les méthodes de `Component` :
  - `Color getForeground()`
    `void setForeground(Color)`
  - `Color getBackground()`
    `void setBackground(Color)`
- classe `Color` :
  - constructeur : `Color(int rouge, int vert, int bleu)`
  - couleurs usuelles : constantes de classe `Color.black, Color.red, Color.blue, ...`

Polices de caractères

- Chaque `Graphics` et `Component` a une « font courante » consultable / modifiable par ses méthodes :
  - `Font getFont()`
  - `setFont(Font)`
- Polices existant sur toutes plates-formes : Serif, SansSerif, Monospaced, Dialog et DialogInput
- Liste complète locale par appel à :
  `String[] Toolkit.getDefaultToolkit().getFontList()`
- Classe `Font` :
  - constructeur `Font(String nom, int style, int taille)` où `nom` est du type « Helvetica », `style` est une des constantes `Font.BOLD`, `Font.ITALIC`, `Font.PLAIN` (ou une combinaison par |), et `taille` est la dimension en « points »

Images

- classe `Image`
- chargement par la méthode `getImage(url_base,name)` de `Applet`, ou bien, hors des applets, par la méthode `getImage(name)` de la classe `Toolkit` (on obtient un Toolkit par `getToolkit()`, ou bien par `Toolkit.getDefaultToolkit()`)
- affichage dans un `Graphics` :
  `boolean drawImage(Image i, int x, int y, ImageObserver o)`
- méthodes de la classe Image :
  - `int getHeight(ImageObserver)`
  - `int getWidth(ImageObserver)`
  - `Image getScaledInstance(int l, int h, int method)`
  - `Graphics getGraphics()` // pour pouvoir dessiner dessus
  - ...

Création d'images

Dans certains cas (double buffering pour animation, ...) on veut dessiner dans un objet `Image` non visible avant de l'afficher. Pour faire cela :

- créer une image par :
  ```
  Image img = createImage(l,h);
  // méthode de Component
  ```
- récupérer un `Graphics` associé :
  ```
  Graphics g = img.getGraphics();
  ```
- dessiner sur `g`
- quand on le souhaite, afficher l'image dans le composant voulu par appel de `drawImage()` sur un `Graphics` associé au composant (et non à l'image)

Autres méthodes de `Graphics`

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

Dimensions de l'écran et des composants

- pour connaître les dimensions de l'écran en pixels (par exemple pour adapter celles des fenêtres) :
  `Toolkit.getDefaultToolkit().getScreenSize()`
  renvoie une instance de la classe `Dimension` qui regroupe dans des attributs publics `width` et `height`

- pour connaître (respectivement modifier) les dimensions d'un composant (en fait, de son rectangle englobant) :
  - `Dimension getSize()`, ou bien `int getWidth()`et `int getHeight()`
  - `setSize(w,h)` ou `setSize(Dimension d)`

Position des composants

- position relative (coin en haut à gauche du rectangle englobant le composant) :
  - `Point getLocation()`
    ou bien `int getX()`et `int getY()`
  - `setLocation(x, y), setLocation(Point p)`
- position absolue (sur l'écran) :
  - `Point getLocationOnScreen()`
- bornes (position + dimension) :
  - `Rectangle getBounds()`, où `Rectangle` regroupe `x, y, width, height`, et diverses méthodes
- test d'inclusion d'un point dans bornes :
  - `boolean contains(x, y)`, `boolean contains(Point p)`
- position de la souris : classe `MouseInfo`

Curseur

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

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

  - méthodes statiques :
    `Cursor.getDefaultCursor()` `Cursor.getPredefinedCursor(int)`

Ecrire une IHM hors applet

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

Exemple d'application graphique élémentaire

```
import java.awt.*;
class Appli extends Frame {
  private Label texte;
  private Image img;
  public Appli() { // constructeur :
    // initialisation des composants
    texte = new Label("UNE IMAGE :");
    add(texte, BorderLayout.NORTH);
    img = getToolkit().getImage("img.gif");
  }
  public void paint(Graphics g) {  // affichage et dessin
    g.setColor(Color.green);
    int d = 10;
    g.fillRect(d, d, getWidth()-2*d, getHeight()-2*d);
    g.drawImage(img, 100, 100, this);
  }
  public static void main(String[] args){
    Appli a = new Appli();
    Dimension d;
    d = Toolkit.getDefaultToolkit().getScreenSize();
    a.setSize(d.width/2, d.height/3);
    a.setLocation(150, 200);
    a.setVisible(true);
  }
}
```

Gestion des événements

- c'est le mécanisme qui permet l'intéraction avec l'utilisateur via l'interface graphique
- Modèle émetteur/récepteur, avec séparation claire entre
  - les éléments d'interface qui émettent les événements,
  - et des objets récepteurs qui « écoutent » les événements et agissent en conséquence
- classes dans le paquetage `java.awt.event`

Modèle d'événement

- chaque composant peut générer des événements (classe abstraite `AWTEvent` et ses sous-classes `MouseEvent, ActionEvent`, ...)
- tout objet `o` qui doit réagir quand un type d'événement se produit dans un certain composant `c` doit :
  - implanter l'interface adéquate (`MouseListener, ActionListener`, ...)
  - être enregistré dans la liste des objets intéressés par ce type d'événements issus de ce composant (`c.addMouseListener(o)`, `c.addActionListener(o)`, ...)
- quand un événement se produit sur le composant `c`, il est transmis à tous les récepteurs enregistrés chez lui pour ce type d'événement, ceci par appel de sa méthode correspondante (e.g., pour appui sur bouton souris, `o.mousePressed(evt)` pour tous les `o` concernés, et où `evt` est l'événement)

Principaux types d'événements

- `WindowEvent` : apparition, iconification, (dé-)masquage, fermeture, ...
- `ComponentEvent` : redimensionnement, déplacement, ...
- `FocusEvent` : début/fin de sélection comme destinataire des inputs (clavier et souris)
- `KeyEvent` : clavier (touche appuyée, ...)
- `MouseEvent` : souris (boutons, déplacement, ...)
- `ActionEvent` : appui sur `Button`, double-clic sur item de `List`, appui sur Return dans un `TextField`, choix d'un `MenuItem`, ...
- `ItemEvent` : (dé-)sélection d'une `Checkbox`, d'un item de `List` ou de `Choice`, passage sur un `MenuItem`, ...
- `TextEvent` : modification de texte
- `ContainerEvent` : ajout/suppression de composant
- `AdjustementEvent` : défilement de `Scrollbar`

Sources des événements

`tous les Component`	`ComponentEvent`
	`FocusEvent`
	`KeyEvent`
	`MouseEvent`

`toutes les Window`	`WindowEvent`
`Button` `MenuItem`	`ActionEvent`
`List`	`ActionEvent` `ItemEvent`

`CheckBox`
`CheckBoxMenuItem`
`Choice`
	`ItemEvent`

`TextField`	`ActionEvent` `TextEvent`
`TextArea et autres TextComponent`	`TextEvent`
`tous les Container`	`ContainerEvent`
`ScrollBar`	`AdjustmentEvent`

Interfaces récepteurs

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

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

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

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

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

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

Interfaces récepteurs (2)

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

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

Classes Adaptateurs

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

Où écrire une méthode de gestion d'événement ?

- Plusieurs solutions possibles :
  - la plus simple et lisible : faire implanter l'interface récepteur adéquate par l'applet, la fenêtre principale, ou la principale classe de l'interface graphique
  - la plus lourde et la moins commode : définir une classe dédiée qui implante l'interface récepteur adéquate
  - créer « à la volée » (juste au moment de l'enregistrer comme écouteur) une classe qui implante l'interface récepteur adéquate

Comment écrire une méthode de gestion d'événement ?

Se servir des méthodes des événements pour avoir des détails sur ce qui s'est passé :

- méthodes communes :
  `Object getSource()` ret. référence au composant qui a émis l'événement
  `int getID()` donne la nature précise de l'événement

- `ComponentEvent` : `getComponent()`
- `InputEvent` : `long getWhen()`, `int getModifiers()`, `void consume()`, `boolean isConsumed()`
- `MouseEvent` : `int getX()`, `int getY()`
  donnent les coordonnées de la souris (relatives à la source écoutée)
- `KeyEvent` : `char getKeyChar()` ou `int getKeyCode()`
- `WindowEvent` : `getWindow()`
- `ActionEvent` : `String getActionCommand()`
- `ItemEvent` : `Object getItem()` et `int getStateChange()`
- `ContainerEvent` : `Component getChild()`

Exemple de gestion d'événements par la principale classe de l'IHM

```
import java.applet.*;
import java.awt.*;
import java.awt.event.*;
// L'applet implante ActionListener
public class ButtonApplet extends Applet implements ActionListener {
  private Label lab;
  private int k = 0;
  public void init() {
    Button b = new Button("++");
    // Enregistrement de l'applet elle-même comme récepteur des évts
du bouton :
    b.addActionListener(this);
    add(b);
    lab = new Label(String.valueOf(k));
    add(lab);
  }
  /* Méthode correspondant à l'interface ActionListener : ici on
     incrémente le label (à chaque appui sur le bouton). */
  public void actionPerformed(ActionEvent e) {
    lab.setText(String.valueOf(++k));
    lab.setSize(lab.getMinimumSize());
  }
}
```

- **Note :** on peut faire la même chose avec une `Frame` au lieu d'une `Applet`

Exemple de gestion d'événements par classe dédiée

```
import java.awt.*;
import java.awt.event.*;
public class CloseableFrame extends Frame {
  private Label lab;
  public CloseableFrame() {
    lab = new Label("Fermez-moi !");
    add(lab);
    // Enregistrement d'un objet séparé comme récepteur des évts :
    addWindowListener(new EcouteurFenetre());
  }
}


/* Classe dédiée à l'écoute de l'événement "fermeture fenêtre"
    (elle  implante WindowListener, en héritant de WindowAdapter
     pour éviter d'écrire les méthodes ne contenant rien). */
class EcouteurFenetre extends WindowAdapter {
  public void windowClosing(WindowEvent e){
    // Arrêt du programmefermeture
    System.exit(0);
  }
}
```


Exemple de gestion d'événements par création d'écouteur « à la volée »

```
import java.applet.*;
import java.awt.*;
import java.awt.event.*;
public class MouseApplet extends Applet {
  private Label lab;
  private int k = 0;
  public void init() {
    lab = new Label(String.valueOf(k));
    add(lab);
    // Création « à la volée » d'un écouteur des clics souris :
    //    on crée une instance de classe anonyme, héritant de
    //    MouseAdapter pour éviter d'écrire les méthodes
    //    ne contenant rien.
    addMouseListener(
      new MouseAdapter() {
        public void mouseClicked(MouseEvent e) {
          k = e.getX();
          lab.setText(String.valueOf(k));
        }
    );
}
```

Paquetage `javax.swing`

- Complément et alternative aux composants de AWT permettant de ne pas être limité au « plus grand dénominateur commun » des composants Windows/Mac/X11-Motif : fonctionnalités plus riches et plus grande fiabilité, mais maniement un peu plus complexe, et vitesse d'affichage un peu plus lente (car les composants Swing sont « dessinés » au lieu de faire intervenir les composants de la plate-forme d'exécution)

- Contient :
  - des composants graphiques (boutons, menus, ...) de « 2-ème génération »
  - des applets et fenêtres de « 2-ème génération »

Les applets Swing

- Indispensables si on souhaite utiliser des composants Swing dans une applet

- classe `JApplet`
  - sous-classe de `Applet` de AWT
  - fonctionne donc de la même façon, sauf (principalement) existence d'une« contentPane » (comme dans tous les conteneurs "top-level" de Swing) ce qui implique :
    - tout composant (ou sous-Panel) `c` doit être ajouté à la contentPane par `getContentPane().add(c);`
    - le « `LayoutManager` » est celui de la contentPane (par défaut, c'est un `BorderLayout`) et se modifie donc par `getContentPane().setLayout(…);`
    - on peut mettre une barre de menu par `setJMenuBar(…);`

Les fenêtres Swing

- Indispensables pour utiliser des composants Swing dans une fenêtre

- classe `JFrame`
  - sous-classe de `Frame` de AWT
  - fonctionne donc de la même façon, sauf principalement :
    - existence de « contentPane » donc :
      - tout composant (ou sous-Panel) `c` doit être ajouté à la contentPane par `getContentPane().add(c);`
      - le « `LayoutManager` » est celui de la contentPane (`BorderLayout` par défaut) et se modifie donc par `getContentPane().setLayout(…);`
    - possibilité de définir un comportement par défaut quand on la ferme (exécuté APRES les `windowClosing(…)` des `WindowListener` enregistrés) par `setDefaultCloseOperation(…);`

Equivalents Swing des autres principaux composants AWT

La plupart des composants graphiques usuels de AWT ont leur équivalent Swing, qui a le même nom avec un J devant :

- `JComponent` : ancêtre de tous les composants Swing (sauf `JApplet`, `JDialog`, `JFrame`)
- `JButton` : bouton usuel
- `JCheckBox` : case cochable (« indépendante »)
- `JLabel` : texte affiché par le programme
- `JList` : liste « scrollable » de choix
- `JTextField`, `JTextArea` : pour entrer du texte
- `JPanel` : conteneur de base pour grouper des composants
- `JDialog` : fenêtre secondaire (« esclave »)
- `JMenu`, `JMenuBar`, `JMenuItem`, `JPopupMenu` : pour les menus
- `JScrollBar` : barre de défilement
- `JScrollPane` : pour donner une vue scrollable d'un seul composant
- `JWindow` : fenêtre sans barre de titre ni bordure

Ces équivalents Swing ont généralement plus de fonctionnalités que leurs homologues AWT

Nouveaux composants élémentaires

- `JRadioButton` : sorte de `JCheckBox`, mais avec un autre look et permettant choix mutuellement exclusifs via `ButtonGroup`
- `JComboBox` : liste déroulante (≈ `Choice` de AWT mais avec plus de fonctionnalités)
- `JPasswordField` : sorte de `JTextField` masquant les caractères tapés (par exemple pour saisie de mot de passe)
- `JTextPane` : zone de texte éditable avec police de caractères et style
- `JSlider` : curseur pour choisir graphiquement une valeur numérique
- `JToolTip` : bulle d'aide
- `JProgressBar` : barre d'avancement de tâche

Nouveaux composants de haut niveau

- `JTable` : tableau (éditable) de données
- `JTree` : représentation de données arborescentes (façon Windows Explorer)
- `JToolbar` : barre d'outils
- `JColorChooser` : utilitaire pour choix de couleur

Principaux nouveaux conteneurs

- `JOptionPane` : boites de dialogue usuelles, créables et affichables par un simple appel de fonction :
  `JOptionPane.showMessageDialog(…);`
  --> message + 1 bouton OK
  `int r=JOptionPane.showConfirmDialog(…);`
  --> message + boutons style Oui / Non / Annuler
  `int r=JOptionPane.showOptionDialog(…);`
  --> message + choix de boutons
  `String s=JOptionPane.showInputDialog(…);`
  --> message + zone saisie texte + boutons OK / Annuler
- `JSplitPane` : pour afficher deux composants côte à côte (ou l'un au-dessus de l'autre), et avec ligne de séparation déplaçable par l'utilisateur
- `JTabbedPane` : regroupement de plusieurs composants accessibles via des onglets
- `JFileChooser` : fenêtre de sélection de fichier (≈ `FileDialog` de AWT mais en mieux)
- `JInternalFrame` : pour faire des sous-fenêtres dans une fenêtre (« bureau virtuel »)

PROGRAMMATION RESEAU : paquetage `java.net`

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

Accès aux protocoles Internet

- classe URL (protocoles http, ftp, ...)
  - constructeurs : `URL(String nom)`, `URL(URL base, String nom)`, ou `URL(protocole,host,port,file)`
  - ouverture de flux en lecture : `InputStream openStream()`
  - manipulation plus fine, infos sur le contenu : `URLConnection openConnection()`, puis méthodes de `URLConnection` :
    - `connect()`,
    - `getContentType(), getContentLength(), ...`
    - `getInputStream(), getOutputStream()`
- conversion chaîne au format url (pour envoi de donnée à programme CGI) : `String URLEncoder.encode(String)`
- manipulation adresse internet : classe `InetAddress` (instances obtenues par `InetAddress.getLocalHost()` ou `InetAddress.getByName(String host)`)

Exemple d'utilisation de la classe URL

```
import java.net.*;
import java.io.*;

class VisuHTML {
  public static void main(String[] arg) {
    try{
      URL url = new URL(arg[0]);
      URLConnection c = url.openConnection();
      c.connect();
      String type = c.getContentType();
      if (type.startsWith("text")) {
         Reader in = new InputStreamReader( c.getInputStream() );
         BufferedReader bin = new BufferedReader(in);
         String ligne;
         while ( (ligne=bin.readLine()) != null)
           System.out.println(ligne);
         bin.close();
      }
      else System.out.println("Fichier de type " + c.getContentType());
    }
    catch(Exception e) { System.out.println(e); }
  }
}
```


Connexion réseau bas niveau

- communication entre deux programmes : via une connexion passant par un « port » (logique) d'une machine « hôte »
- une « socket » (« prise ») = extrémité d'une connexion
- écrire un client : classe `Socket`
  - constructeur `Socket(String host,int port)`
  - ouverture flux I/O : `getInputStream(), getOutputStream()`
  - fermeture connexion : `close()`
- écrire un serveur : classe `ServerSocket`
  - constructeur : `ServerSocket(int port)`
  - méthode pour attendre connexion de clients : `Socket accept()`
  - fermeture serveur : `close()`
- manipulation directe de paquets : classes `DatagramPacket` et `DatagramSocket`

Exemple de Client

```
import java.net.*;  import java.io.*;
class ClientMiroir {
  public static void main(String[] arg){
    try {
      Scanner sc = new Scanner(System.in);
      // ouverture socket (port=9999, host=arg[0]) :
      Socket client = new Socket(arg[0], 9999);
      // ouverture de deux flux sur socket :
      OutputStream out = client.getOutputStream();  // flux en écriture
      InputStream in = client.getInputStream();     // flux en lecture
      while (true) {
         String s = sc.nextLine();  // lecture ligne au clavier
         out.write(s.getBytes());   // recopie sur socket
         out.write('\n');
         out.flush();
         // lecture sur socket :
         byte [] buf = new byte[1000];
         in.read(buf);
         s = new String(buf);
         System.out.println(s);
      }
    }
    catch (IOException e) { System.out.println(e); }
  }
}
```


Exemple de serveur

```
import java.util.*;  import java.net.*;  import java.io.*;
class ServeurMiroir {
  public static void main(String[] arg){
    try {
      // mise en route du serveur sur port 9999 :
      ServerSocket serveur = new ServerSocket(9999);
      Socket client = serveur.accept();  // attente connexion client
      InputStream is = client.getInputStream();  // ouvre flux lecture
      OutputStream out = client.getOutputStream();  // idem en écriture
      // fonctionnement du serveur :
      StringBuffer buf = new StringBuffer();
      do {
        int c = is.read();  // lecture caractère par caractère
        if (c == -1) { break; }  // fin de flux
        if (c != '\n') { buf.append( (char) c ); } // cararactère std.
        else {  // traitement des fins de ligne sur flux lecture :
          buf.reverse();
          out.write(buf.toString().getBytes());
          out.flush();
          buf = new StringBuffer();
        }
      } while (true);
    } catch (IOException e) { System.err.println(e); }
  }
}
```

PROGRAMMATION DISTRIBUEE : `java.rmi`

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

Principe

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

Utilisation

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

Exemple

Interface

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

Objets distants et serveur RMI

```
import java.rmi.*;
import java.rmi.server.*; // pour UnicastRemoteObject
public class CompteDist extends UnicastRemoteObject implements Compte {
  private int solde = 0;
  public CompteDistant() throws RemoteException{}
  public int lireSolde() throws RemoteException { return solde; }
  public void deposer(int x) throws RemoteException { solde += x; }
  public void retirer(int x) throws RemoteException{ solde -= x; }
  /* programme serveur créant les comptes */
  public static void main(String[] arg) {
    CompteDistant cd1, cd2;
    try {
      // création des comptes :
      cd1 = new CompteDist(); cd2 = new CompteDist(); cd2.deposer(1000);
       // pour gestion sécurité :
       System.setSecurityManager(new RMISecurityManager());
      // enregistrement des objets
      Naming.rebind("Dupond", cd1); Naming.rebind("Martin",cd2);
    } catch(Exception e) { System.out.println(e); }
}}
```

Exemple (suite)

Client RMI

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

ACCES BASES DE DONNEES : `java.sql`

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

Principe de programmation

- 1/ on charge le driver de BD par `Class.forName(nomClasseDriver)`
- 2/ on se connecte :
  `Connection c = DriverManager.getConnection(String url)`
  où `url` est de la forme `jdbc:nom_driver:nom_base`
- 3/ on crée un objet `Statement` à partir de la connexion :
  `Statement = c.createStatement();`
- 4/ on peut ensuite passer directement les instructions SQL à l'objet `Statement` :
  - modifications : `s.executeUpdate(String commandeSQL)`
  - requête : `ResultSet rs=s.executeQuery(String commandeSQL)`
  - exploitation du `ResultSet` :
    - ligne suivante : `rs.next()`
    - lecture champ : `rs.getString(nom)`, `rs.getFloat(n)`

LES JAVABEANS : `java.beans`

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

JavaBeans (suite)

- l'outil de développement doit pouvoir reconnaître automatiquement les propriétés paramétrables du bean

- possible grâce à l'introspection Java (cf. classe **`Class`**, ...) et au respect de règles de conception (« design patterns ») :
  - pour chaque propriété, méthode `<TypeProp> get<NomProp>()` et `void set<NomProp>(<TypeProp> val)`
  - …

INDEX (1)

INDEX (2)

INDEX (3)

INDEX (4)

INDEX (5)

INDEX (6)

INDEX (7)

INDEX (8)

REFERENCES

Pour approfondir un point, ou revenir sur quelque chose que vous n'avez pas bien compris, vous pouvez vous reporter :

1°/ aux ressources suggérées dans la page : `www.ensmp.fr/CC/Docs/Java` et en particulier :

- le « **Java Tutorial** » d'Oracle ;
- les autres sites rassemblant ***exemples, tutoriaux, …*** comme l'excellent `www.java2s.com` (en anglais) ou `www.developpez.com` (en français)

2°/ à un livre sur Java, par exemple :

- « *Programmer en Java* », C. Delannoy, éd. Eyrolles
- « *The Java programming language* », K. Arnold, J. Gosling et D. Holmes, éd. Addison Wesley