Introduction à la programmation en langage Java

Adaptation du cours de Fabien Moutarde

du centre de calcul de l'Ecole Nationale Supérieure des Mines de Paris

Sovanna TAN

UFR Sciences et Technologie

Université Paris 12

email : sovanna@univ-paris12.fr

Décembre 2000

Bibliographie

- *Introduction à la programmation en Java, cours et exercices,* Jean Brondeau, Dunod, 1999
- *Le langage Java, concepts et pratique,* Irène Charon, Hermès, 2000
- *Le développeur Java2*, Antoine Mirecourt & Pierre-Yves Saumont, Osman Eyrolles Multimédia, 1999
- *Java in a nutshell*, David Flanagan, et al., O'Reilly, 1997
- *Java examples in a nutshell*, David Flanagan, O'Reilly, 1997
- *Java Swing*, Robert Eckstein, Marc Loy & Dave Wood, O 'Reilly, 1998

Organisation du cours

- Introduction
- Syntaxe de base
- Classes
- Héritage
- Exceptions
- Threads

Organisation du cours (2)

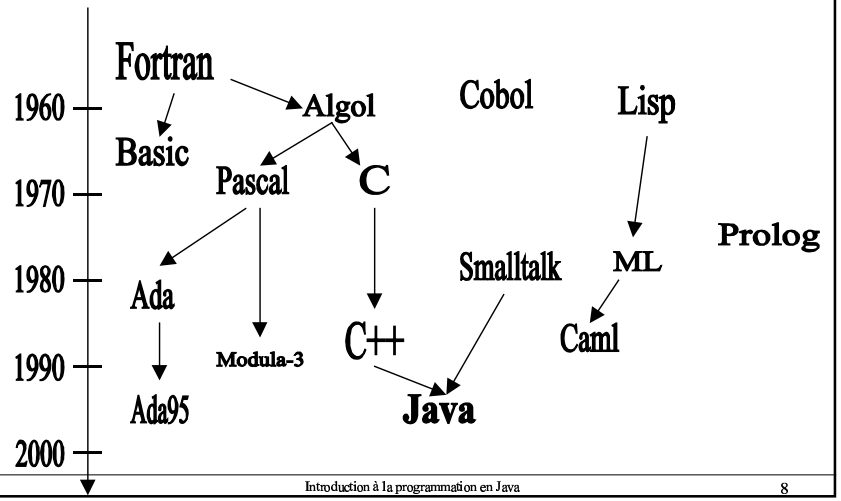
- Paquetages standards
  - `java.lang`
  - Entrée-sorties : paquetage `java.io`
  - Paquetage `java.util`
  - Graphisme : `java.awt, javax.swing`
  - Programmation réseau : `java.net`
  - Accès bases de données : `java.sql`

INTRODUCTION

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, Ada95, C++, Java,...

Les langages de programmation (2)



Historique de Java

- Initialement, projet de SUN 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 par SUN en mai 1995
- Après l'engouement pour les applets reconnu comme un langage à part entière

Historique de Java (2)

- En voie de devenir le principal langage de programmation :
  ≈ 800.000 développeurs Java
  ≈ 1500 applications "industrielles"
  (chiffres fin 1998)
  - centaines d'outils de développement
  - standard de fait
  - processus de normalisation enclenché

Historique de Java (3)

- évolution très rapide
- janvier 1996 : Java 1.0
- décembre 1996 : Java 1.1
  - classes internes,
    nouveau modèle événement pour AWT, JDBC, rmi, ...
- décembre 1998 : Java 1.2
  (renommé Java 2)
  - swing, Java2D, ...

Intérêt de Java

- **logiciels portables**
- **programmes fiables**
  (rigueur du langage => peu de bugs)
- **développement rapide**
- pages Web intéractives
- 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
  - 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 rapp. 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
  - pas de généricité

Java et les autres langages (2)

- ajouts de Java (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

Le JDK

- Java Development Kit
- distribué gratuitement par SUN
- existe pour quasiment tous les types d'ordinateurs et d'OS

Le JDK (2)

- contient :
  - compilateur (`javac`)
  - interpréteur / machine virtuelle (`java`)
  - toute la librairie de classes standards
  - outils divers : génération doc (`javadoc`), visualisation d'applet (`appletviewer`), debugger (`jdb`), ...
- sources disponibles gratuitement (pour les applications non commerciales) mais restent la propriété de SUN (de même que la marque, le logo, ...)

SYNTAXE DE BASE

Variables et types

- notion de variable
  (nom + type + zone mémoire)
- en Java, deux grandes catégories de types :
  - types de base (entiers, flottants, …)
  - références à :
    - tableaux
    - objets
    - interfaces

Types de base

- **`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 spéficier 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 des entiers

- conversion automatique seulement vers types entiers + grands (`int`→`long`, etc...) et vers types flottants

Caractères : char

- 16-bit => 65536 valeurs : presque tous les caractères de toutes les écritures !
- 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 :
  par exemple `'\u03c0'` pour π

Manipulation des caractères

- test du type : `Character.isLetter(c)`, `Character.isDigit(c)`, ...
- convertible automatiquement en `int` ou `long` (et manuellement en `byte` ou `short`)

Flottants

- notation "ingénieur" : `2.45e-25`
- littéraux de type double par défaut :
```
float x = 2.5; // Erreur
double y = 2.5; // OK
```
- f ou F pour spécifier un float :
```
float x = 2.5f;
```
- valeurs min/max (de valeur absolue) :
  - `float` =[1.40239846e-45;3.40282347e+38]
  - `double` = [4.9406...e-324 ; 1.7976...e+308]

Conversion des flottants

- conversion automatique : seulement `float`→`double`
- conversion "manuelle" en entier tronque la partie décimale :

```
float x=-2.5f;
int i = (int)x; // => i==-2
```


Constantes

- variable dont la valeur ne peut plus être changée une fois fixée
- mot-clé `final` :

```
final double PI = 3.14159;
PI = 3.14; // ERREUR
```

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

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

Déclarations de variables

- déclaration préalable obligatoire
- identificateurs :
  - caractères Unicode
  - début par lettre, _ ou $
  - ensuite : lettre, chiffre, _ ou $

Déclaration et initialisation des variables

- 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
```

Conventions de nommage

- pas obligatoires, mais recommandées (pour la lisibilité du code)
- identificateurs en minuscules, sauf :
  - majuscule au début des "sous-mots" intérieurs :
    `unExempleSimple`
  - début des noms en majuscule pour classes et interfaces (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

- style C :
  ```
  /* Commentaire de style C qui
   s'étend sur plusieurs lignes */
  ```
- style C++ :
  ```
  //Commentaire -> fin de la ligne
  ```
- 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 tableaux et objets
- plusieurs références différentes peuvent référencer un même objet ou tableau
- les références sont typées

Références (2)

- 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}
```

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

Références et mémoire

- déclaration d'une variable : réservation de l'espace mémoire pour stocker la valeur de la variable pour les types de base

```
int a=1;
// réserve l'espace mémoire pour stocker un
// entier
```

- pour les objets ou les tableaux : réservation de l'espace mémoire pour stocker une référence sur un objet ou un tableau

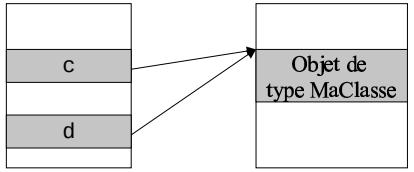
```
MaClasse c;
// c vaut null
```


Références et mémoire

- construction de c

```
c=new MaClasse();
MaClasse d=c;
```

  - allocation d'un emplacement mémoire pour un objet de type `MaClasse`
  - affectation de la variable `c` à la valeur de cet emplacement



Tableaux

- 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`

Déclaration et création de tableaux

- déclaration :
```
int tab[];
int [] tab2; // 2° forme autorisée
```
- création :
  - par `new` après (ou dans) la déclaration :
```
int tab[];
tab = new int[5];
```
  - ou implicite par initialisation
```
int tab2[]={1,2,3};
```

Initialisation de tableaux

- initialisation :

```
// A la déclaration
int tab[] = {1, 2, 3};

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

// Sinon, élément par élément
//   par affectation (ou sinon,
//   valeurs par défaut)
```


Dimension d'un tableau

- accès à la taille :

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

- indices **entre `0` et (`length-1`)**
- si tentative accès hors bornes, lancement de `ArrayIndexOutOfBoundsException`

Tableaux et mémoire

- 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" dès que plus désigné par aucune référence

```
ex. : l'espace occupé par {1,2,3} ci-dessus après
l'instruction t1=t2
```


Copie de tableau

- méthode « directe » : création d'un nouveau tableau puis copie élément par élément
- copie rapide possible par appel de System.arraycopy()

```
// copie des 3 premiers éléments
// de src dans les 3°,4°,5° de dst
System.arraycopy(src,0,dst,2,3);
```

- System.arraycopy() peut marcher pour un déplacement interne à un même tableau :

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

- ATTENTION : pour un tableau d'objets, copie des références => les mêmes objets sont ensuite partagés par les 2 tableaux

Tableaux à plusieurs dimensions

- tableaux de (références vers) tableaux :
  `float matrix[][]=new float[2][3];`

- possibilité de forme non carrée :

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


Références constantes

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

- mot-clé `final` :

```
final double[] tab;
tab = new double[10];
// La référence tab est figée :
tab = new double[20]; // ERREUR
// Le tableau reste modifiable :
tab[0] = 1.732; // OK
```

Chaines de caractères

- objets des classes
  `String` ou **`StringBuffer`**

- chaînes littérales : `"une chaîne"`
- concaténation par +

Chaines de caractères non modifiables

- 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";
```

Chaines de caractères modifiables

- chaînes modifiables : **`StringBuffer`**

```
// Création
StringBuffer buf;
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);
```


Chaines de caractères modifiables (2)

- Remarques :

```
// ATTENTION :
//  - pas de compareTo()
//  - pas de equals() ne teste pas l'égalité
//               des chaînes contenues
```

Principaux opérateurs

- affectation : `=`
- arithmétiques : `+ - * / %`
- comparaisons : `< <= > >= == !=`
- booléens : `&& || ! ^ & |`
- opérations bit-à-bit (sur les entiers) :
  `& | ^ ~ << >> >>>`

Principaux opérateurs

- opération et affectation simultanées :
  `+= -= *= /= %= &= |=`
  `^= <<= >>= >>>=`
- pré/post-incrémentation : `++`
  pré/post-décrémentation : `--`

Opérateurs arithmétiques

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

- 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	`expr1&&expr2`
||	ou	`expr1||expr2`
^	ou exclusif (xor)	`expr1^expr2`
!	négation	`!expr`
&	et (non optimisé)	`expr1&expr2`
|	ou (non optimisé)	`expr1|expr2`

Opérateurs booléens

- les opérandes doivent être des expressions à valeurs booléennes
- remarques :
  - pour &&, deuxième opérande non évalué si premier est faux (pas pour &)
  - pour ||, deuxième opérande non évalué si premier est vrai (pas pour |)

Opérateurs bit-à-bit

Opérateur	Fonction	Usage
&	et	op1&op2
|	ou	op1|op2
^	ou exclusif (xor)	op1^op2
~	négation	~op
<<	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 ;  ⇔  i = i+3;
```

Incrémentation et décrémentation

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

- servent essentiellement à compacter les écritures :

$$\texttt{n=i++;} \Leftrightarrow \begin{cases} \texttt{n=i;} \\ \texttt{i=i+1;} \end{cases}$$

$$\texttt{n=++i;} \Leftrightarrow \begin{cases} \texttt{i=i+1;} \\ \texttt{n=i;} \end{cases}$$

Autres opérateurs

- ternaire conditionnel :
  `bool ? expr1 : expr2`
- conversion : `(type)`
  ```
  float x = 1.5f;
  int i = (int)x;
  ```
- allocation mémoire (création de tableaux ou d'objets) : `new`
- test de type : `instanceof`

  `(ref instanceof MaClasse)` vaut :
  - `true` si `ref` référence un objet qui peut être considéré comme de type `MaClasse`
  - `false` sinon (y compris si `ref==null`)

Priorité des opérateurs

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

Instructions

- chaque instruction se termine par un ';'
  Exemples :

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

Blocs

- on peut grouper plusieurs instructions en un bloc délimité par `{` et `}`

  Exemple :

```
{
int x;
x = a;
a = b;
b = x;
}
```


Instructions de contrôle

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

Exécution conditionnelle

- **if** (boolExpr)
  instruction;
- **if** (boolExpr) {
  // Exécuté si VRAI
  }
  else {
  // Exécuté si FAUX
  }

- **if** (boolExpr) {
  // Exécuté si VRAI
  }
  else if (boolExpr2)
  instruction;
  else {
  // ...
  }

Cas multiples

- **switch** (expr) {
  case cst1:
  // instructions si expr==cst1
  break;
  case cst2:
  // instructions si expr==cst2
  case cst3:
  // instructions si
  // expr==cst3 || expr==cst2
  break;
  default:
  //instructions si aucune
  // des valeurs prévues
  break;
  }

Cas multiples (2)

- `expr` : de type entier ou `char`
- `cst1`, ... : littéral ou constante (`final`)

Boucles de répétitions

-
```
while (boolExpr) {
  // Corps de la boucle
}
```

-
```
do {
 // Corps de la boucle
} while (boolExpr);
```

Boucles d'itération

- `for (initialisations ; boolExpr ; incréments) { // Corps de la boucle }`
- initialisations : déclaration et/ou affectations, séparées par des virgules
- incréments : expressions séparées par des virgules

Boucles d'itération (2)

- La boucle `for` est équivalente à :

```
initialisations
while (boolExpr) {
  // Corps de la boucle
  incréments
}
```

Interruptions des boucles

- `break` : sortie de la boucle
- `continue` : passage à l'itération suivante
- `return` : sortie immédiate de la fonction en cours, donc à fortiori sortie de la boucle en cours par la même occasion

Les fonctions

- éléments essentiels de structuration en programmation impérative « classique »
- 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)
- en Java, chaque fonction est de plus définie dans une classe (ex. : la classe `Math` regroupe les fonctions mathématiques)

Exemple de fonction

- Les fonctions de la programmation impérative classique se définissent avec le mot-clé `static`

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


Fonction : retour

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

Fonction : retour (2)

- Exemple

```
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);
```

depuis la 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);
    System.out.println(k);
    }
  }
```

Appel de fonction (2)

- appel de fonction : les arguments doivent être du type prévu ou de type convertible automatiquement vers type prévu

Programme principal

- En Java, exécution de programme = recherche et appel d'une fonction publique particulière :
  - nommée `main`,
  - prenant en paramètre un `String[]`,
  - ne retournant rien,.
  - déclarée avec le mot-clé `public`.

Exemple :

```
class MaClasse {
 public static void main(String [] args) {
    // Programme principal
 }
}
```

Passage des paramètres

- par valeur pour les types de base :

```
public static void echange(int i, int j){
  // i et j = copies locales des valeurs
  int tmp=i;
  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
}
```


Passage des paramètres (2)

- par valeur des références :

```
public static void ech(String s1,
                       String s2) {
  // s1 et s2 = copies locales des références
  String tmp=s1;
  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 (3)

- la référence est passée par valeur, mais l'objet référencé peut être modifié :

```
public static void increm(int t[]) {
  for (int i=0; i<t.length; i++)
    t[i]++;
}
public static void main(String[] args){
  int tab[]={1,2,3};
  increm(tab);
  // tab vaut {2,3,4}
}
```


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

Surcharge

- En Java, contrairement au C, possibilité de définir dans la même classe plusieurs fonctions de même nom, différenciées par le type de leurs arguments
- une différence fondée uniquement sur le type de retour est interdite

Exemple de surcharge

```
class NomClasse {
   public static int longueur(String s){
     return s.length();
   }
   public static int longueur(int i){
     String chiffres;
     chiffres=String.valueOf(i);
     return chiffres.length();
   }
   public static void main(String [] args) {
      int k=12;
      String ch="oui";
      int l1=longueur(k);
      int l2=longueur(ch);
   }
  }
```

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

- Pas de readInt(), readFloat(), readLine(),... en standard !!

Exemple de programme

```
public class MaClasse {
    // Un programme "ordinaire" contient
    //  toujours un programme principal
    //  défini comme ci-dessous
    public static void main(String[] args){
      int i;
      for (i=0; i<10; i++)
        System.out.println(i);
    } // fin du programme principal
  }  // 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)
- compilation du fichier `MaClasse.java` :
  ```
  javac MaClasse.java
  ```
  cela produit un fichier nommé
  ```
  MaClasse.class
  ```
- exécution du `main` de MaClasse :
  ```
  java MaClasse
  ```


CLASSES

Classes

- **classe** = modèle d'objet
  = attributs + méthodes
  - implémentation d'un type de données défini par le programmeur
- **attributs** : les **données propres à l'objet**
  - encore appelés champs ou variables d'état
- **méthodes** : des **opérations applicables à l'objet**
  - consultation, modification des attributs

Exemple de classe

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

Objet

- instance d'une classe
- un exemplaire des variables d'état (attributs) est crée à chaque création d'objet
- l'état d'un objet ne peut être manipulé que par le code de la classe qui lui a servi de modèle
- création uniquement avec l'opérateur `new` :

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


Destruction d'objet

- destruction automatique par le "garbage collector" quand il n'y a plus aucune référence vers l'objet :

```
Cercle c2=new Cercle();
// ...
c2=c1;
// => objet Cercle créé par new
//  pour le c2 initial
//  pourra être détruit
```

Attribut

- utilisation dans le cas général : usage de l'opérateur **.** ("point")
```
Cercle c1 = new Cercle();
float r=c1.rayon;
```
- utilisation depuis une méthode :
  accès direct pour l'instance courante
```
class Cercle {
 // ...
 float comparerA(Cercle autre){
   return rayon - autre.rayon;
 }
}
```


Méthode

- appel "direct" depuis une autre méthode de la même classe :
```
class Cercle {
 //...
 void afficherAire(){
   float a=calculerSurface();
   System.out.println(a);
 }
}
```
- appel des méthodes dans les autres cas :
  usage de opérateur **.** :
```
float s = c1.calculerSurface();
```

Surcharge des méthodes

- **signature** d'une fonction ou d'une méthode : type de retour, nom et types des paramètres
  - En Java utilisée dans la définition des "interfaces" qui permettent de spécifier les classes
  - En C et C++, utilisée pour déclarer les fonctions et les méthodes
- **surcharge** : plusieurs méthodes de même nom, mais de signatures différentes (i.e. avec types de paramètres différents, pas uniquement des types de retour différents)

La référence "this"

- possibilité dans les méthodes de désigner explicitement l'instance courante :
  - pour accéder aux attributs "masqués" par des paramètres :

```
class Cercle {
 // ...
 float comparerA(float rayon){
   return this.rayon-rayon;
 }
}
```

La référence "`this`" (2)

– 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'implémentation, et ne laisser voir que l'interface fonctionnelle
  - facilite le développement en équipe et la maintenance
- pour chaque attribut et méthode, visibilité possible :
  - `public`
  - `private`
  - `protected`
  - par défaut, accès "package"

Encapsulation (2)

- encapsulation "usuelle" : attributs privés, et méthodes publiques (sauf celles à usage interne) :

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


Constructeurs

- méthodes pour initialiser les attributs lors de la création :

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

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

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

Constructeurs (2)

- il peut y en avoir plusieurs (surcharges avec différentes signatures)
- on peut invoquer un constructeur depuis un autre :

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


Constructeurs (3)

- conseil de programmation :
  - faire d'abord un constructeur "complet" avec tous les paramètres possibles,
  - faire en sorte que tous les autres (en général un constructeur sans argument et un pour construire par recopie d'un objet de même type) appellent le constructeur "complet »
- REMARQUE :
  - si une classe ne possède pas de constructeur, Java fournit un constructeur par défaut sans paramètre
  - ce constructeur disparaît dès qu'un constructeur est défini dans la classe, il est important de le définir explicitement

Destructeur

- appelé par le garbage collector 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 mort
- `void finalize() {...}` *`// méthode de la classe // Object que l'on peut surcharger`*
- appelé une seule fois par instance (donc pas ré-appelé 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 les objets :

```
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();
```

Attribut statique

- variable partagée par toutes les instances de la classe
  - variable globale à une classe
  - nommé aussi attribut de classe
- mot-clé `static`
- exemple :
  - `Math.PI`

Autre exemple d'attribut statique

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

Constantes de classe et type énuméré

- constante de classe : attribut `static` et `final`
- absence de type énuméré en Java => utiliser des constantes de classe, en faisant éventuellement une classe dédiée à cela

```
class Direction {
  public static final int NORD=1;
  public static final int EST=2;
  public static final int SUD=3;
  public static final int OUEST=4;
}

tournerVers(Direction.EST);
```


Méthode statique

- type de méthode ne s'appliquant pas à une instance particulière de la classe
  - équivalent des fonctions "ordinaires" des langages non-objet
  - appelée aussi méthode de classe
- mot-clef `static`
- exemples :
  - `Math.log(double)`
  - fonction principale : `main`

Autre exemple de méthode statique

```
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 :

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

Paquetage

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

Organisation en fichiers

- au maximum une classe ou interface publique par fichier source
- s'il y en a une, le nom du fichier doit être celui de la classe
- il peut y avoir d'autres classes et interfaces non publiques dans le même fichier

Compilation et exécution

- compilation du fichier PubClasse.java :
  `javac PubClasse.java`
  produit pour chaque classe un fichier nommé
  `NomClasse.class`
- exécution du `main` de la classe NomClasse : `java NomClasse`

  (`java nomPackage.NomClasse` depuis le répertoire-père de nomPackage)

Paquetage : importation

- nom (complet) d'une classe (publique) à l'extérieur de son paquetage :
  `nomPackage.NomClasse`
  sauf si classe importée
- importation : permet d'utiliser une classe d'un autre paquetage avec nom court
- importation de classe (publique) par :
  `import nomPackage.NomClasse;`
- importation "à la demande" de toute classe publique du package :
  `import nomPackage.*;`

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 sont liés

Documentation automatique

- outil `javadoc` du JDK
- écrire commentaires spécifiques
  `/** bla-bla ... */` juste avant déclaration :
  - de chaque classe (ou interface)
  - de chaque méthode
  - de chaque attribut
- `javadoc NomClasse.java` produit automatiquement des fichiers HTML décrivant la classe et intégrant ces commentaires
- 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 formattés 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`)

Exemple

```
package info;
/**
* Calcul des factorielles jusqu'à 20!,
*/
public class fact{
   static long[] table=new long[21];
   static {
     table[0]=1;
   }
   static int last=0;
/**
* calcule n! pour n entier positif inférieur ou égal à 20
* @param n entier positif inférieur ou égal à 20
* @return  n!
*/
```

Exemple (2)

```
public static long factorielle(int n){
    if(n>table.length){
       System.out.println("Débordement");
        return -1;
    }
    else if (n<0){
       System.out.println("n doit être positif");
       return -1;
    }
    while(last<n){
        int l=last++;
       table[last]=table[l]*last;
    }
    return table[n];
}
```


Exemple (3)

```
public static void main(String[] args){
    long n=factorielle(20);
    for(int i=1;i<21;i++)
      System.out.println(table[i]);
  }
}
```

Exemple (4)

```
1
2
6
24
120
720
5040
40320
362880
3628800
39916800
479001600
6227020800
87178291200
1307674368000
20922789888000
355687428096000
6402373705728000
121645100408832000
2432902008176640000
```


HERITAGE, classe racine, interfaces

Héritage

- définir une classe (classe dérivée) à partir d'une classe existante comme extension de cette dernière (super classe ou classe de base)
  - la classe dérivée possède tous les attributs et les méthodes de la classe de base et des attributs ou des méthodes spécifiques
  - elle ne peut pas accéder aux membres privés de sa classe de base
- permet de réutiliser une classe existante en l'adaptant, et/ou de factoriser des choses communes à plusieurs classes
- on peut faire évoluer le code sans modifier ce qui fonctionne

Héritage (2)

- défini avec le mot-clef `extends`

```
class Figure{
    private String nom;
    protected Position pos;
    public Figure(Position p){pos=p;}
    public void deplacer(int dx,int dy){
      pos.ajouter(dx,dy); }
    public void afficher() {
      pos.afficher(); }
    public boolean contient(Figure f){…}
}
class Cercle extends Figure{
    protected float rayon;//…
    public void afficher(){…}
}
```

Héritage (3)

- la sous-classe hérite de tous les membres (sauf membres privés et constructeurs) :

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

- on peut définir autant de niveau d'héritage que l'on veut
- héritage simple uniquement (pas plus d'une classe mère)
  - toutes les classes dérivent de la classe racine `java.lang.Object`

Héritage et polymorphisme

- **polymorphisme d'héritage** : 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
```

Polymorphisme dynamique

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

```
Figure f = new Cercle();
f.afficher();
// appel de afficher() de Cercle f.afficher();
```

- permet par exemple d'itérer des traitements sur des objets de type différents ayant une classe parente commune

Polymorphisme dynamique (2)

```
class Triangle extends Figure {…}
class Rectangle extends Figure {…}
  //...
Cercle c1=new Cercle();
Triangle t1=new Triangle();
Rectangle r=new Rectangle();
Cercle c2=new Cercle();
Triangle t2=new Triangle();
// creation d'une liste [c1,t1,r,c2,t2]
for(Figure f = …; …;){ // f parcourt la liste
   f.afficher();
}
```

Héritage et visibilité

- la classe fille ne peut accéder qu'aux membres (attributs ou méthodes)
  - publics,
  - protégés hérités,
  - et "package", à condition de faire partie du même paquetage que sa classe mère :

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


Héritage et constructeurs

- les constructeurs ne sont pas hérités, mais on peut appeler ceux de la classe mère avec le mot-clé `super` :

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

Héritage et constructeurs

- l'appel de constructeur de classe mère doit être la première instruction
- si la 1ère instruction n'appelle ni constructeur de mère, ni autre constructeur de fille, alors appel automatique de `super()` sans argument
- ordre des opérations :
  1/ appel constructeur 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 :

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

Spécialisation de méthode (2)

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

Méthode abstraite

- méthode non implémentée (on ne fait que spécifier son nom, les types de ses paramètres et son type de retour)
- peut être utilisée dans une méthode non abstraite
- mot-clef `abstract`

```
public abstract void colorier();
```

- destinée à être définie dans les classes filles
- ne peut exister que dans une classe elle-même déclarée abstraite
- impossible pour les méthodes "statiques"

Classe abstraite

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

```
abstract class Figure {
  //…

    public abstract void colorier();
}
```

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

Exemple de classe abstraite

```
abstract class Forme{
  abstract float perimetre();
  abstract float surface();
  void decritEtalement(){
    float lePerimetre=perimetre();
    if(surface >= lePerimetre*lePerimetre/16 )
       System.out.println(this+" s'etale plus
  qu'un carre");
    else
       System.out.println(this+" s'etale moins
  qu'un carre");
  }
}
```

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`

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

- comme classes non dérivables, mais plus flexible

Classes ou interfaces internes (ou "nichées")

- classes ou interfaces définies à l'intérieur d'une autre (au même niveau que attributs/méthodes)
- intérêt : classes (ou interfaces) utilitaires très liées à la classe englobante
- pour les classes internes, 2 catégories :
  - statiques : classes "normales" mais fortement liées aux englobantes
  - membres : associées à chaque instance => peuvent accéder directement aux attributs privés de la classe englobante

Exemple de classes internes

```
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 implémentation)

Exemple de classe locale

```
class MaClasse{
  public void init(final int x) {
     class Local{
        Local(){
           System.out.println(x);
        }
        long carre(){
          return x*x;
        }
     }
     System.out.println(new Local().carre());
  }
}
```

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

tabM[0]=new Fille(); // OK
```


Classe racine `Object`

- ancêtre de toutes les classes, définie dans `java.lang`
- définit donc des méthodes héritées par toutes les classes :
  - `public boolean equals(Object obj)`
    par défaut, retourne this==obj, mais prévue pour être redéfinie en comparaison de contenu (ex : classe String)
  - `public String toString()`
    par défaut "NomClasse@"+hashCode(), mais à redéfinir en une représentation String de l'objet
  - `public int hashCode()`
  - `protected Object clone()`
  - `public Class getClass()`
  - ...

Programmation générique

- la classe `Object` permet aussi de faire de la programmation générique, i.e. des classes ou fonctions qui peuvent fonctionner avec des instances de n'importe quelle classe
- 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 classe générique : la classe `LinkedList` de `java.util` qui est une liste dont les éléments peuvent être des objets de n'importe quelle classe

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=(int[])tab.clone();
// noter la conversion en int[]
```

- attention si tableau est multidimensionnel ou s'il contient des objets d'objets, car copie des références

Disposer de la méthode `clone()`

- pour les objets que l'on veut clonables :
  1/ déclarer que la classe implémente l'interface `Cloneable`,
  2/ redéfinir la méthode `clone()` comme `public`, et soit retournant `super.clone()`, soit adaptée à la classe

```
public class Voiture implements Cloneable{
      String marque;
      Voiture(String s){
           marque=s;
      }
     public Voiture clone(){
          return new Voiture(new String(marque));
     }
}
```


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();
```

Classes `Class, Method, Field` (2)

- permet de tout savoir sur la classe :
  - `Method[] getDeclaredMethods()`
  - `Field[] getDeclaredFields()`
  - `Constructors[] getDeclaredConstructors()`
  - `Class[] getInterfaces()`
  - `Class getSuperClass()`
  - `boolean isInterface()`
  - `boolean isArray()`
  - `boolean isPrimitive()`

Interfaces

- définition abstraite d'un service, indépendamment de la façon dont il est implémenté, spécification d'un type
- 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);
}
```


Utilisation des interfaces

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

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

Utilisation des interfaces (2)

- une classe qui implémente une interface doit définir toutes les méthodes de l'interface (ou bien être abstraite) :

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


Utilisation des interfaces (3)

- si `MaClasse` implémente `Interf`, alors on peut appliquer à toute instance de `MaClasse` toutes les méthodes de `Interf` :

```
Cercle cercle = new Cercle();
cercle.grossir(2);
cercle.colorier(Coloriable.ROUGE);
```

- chaque interface définit un type de référence

Utilisation des interfaces (4)

- une référence à une interface peut désigner toute instance de toute classe qui implémente l'interface en question :

```
Redimensionnable r;
Coloriable c;
Cercle cercle = new Cercle();
r = cercle;
c = cercle;
// cercle, c et r forment
// 3 vues différentes du même objet
```


Utilisation des interfaces (5)

- si une méthode attend en argument une référence à `Interf`, on peut lui passer une référence à toute instance de toute classe qui implémente `Interf`

```
interface Comparable {
  int comparerA(Object o);
}
class Tri {
  public static void trier(Comparable[]
                                tab) {
    //...
    if (tab[i].comparerA(tab[j]) > 0)
      //...
  }
}
```

Utilisation des interfaces (6)

```
class W implements Comparable {//...
  public int comparerA(Object o){...}
}

W[] tabW=new W[10];
// ...
Tri.trier(tabW);
```


Les interfaces comme paramètre de type fonction

- une interface peut jouer le rôle de "pointeur sur fonction" en Java :

```
interface FonctionUneVariable{
  double valeur(double d);
}
class Exp implements FonctionUneVariable{
  double valeur(double d){
    return Math.exp(d);

  }
}
```

Les interfaces comme paramètre de type fonction

```
class Integration{
  public static double integrer(
   FonctionUneVariable 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/n;
  }
}
double intExp = 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 mère implémente une interface, alors toutes ses classes filles héritent de cette propriété :

```
class Mere implements Z{...}
class Fille extends Mere {...}
//...
Fille f = new Fille();
// => f peut être vue comme réf. à Z
//        (et aussi comme réf. à 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'implémentent
- exemple : interface `Cloneable` pour identifier les classes aux instances desquelles on peut appliquer la méthode de duplication `clone()`

EXCEPTIONS

Exceptions

- mécanisme pour traiter les anomalies se produisant à l'exécution
- principe :
  - signaler tout problème dès sa détection
  - mais regrouper le traitement des problèmes ailleurs, en fonction de leur type
- divers types d'exceptions (classes) sont prédéfinies, et générées automatiquement à l'exécution

Capture des exceptions

- si rien n'est prévu, toute exception générée provoque l'arrêt complet du programme
- pour éviter cela, prévoir capture et traitement des exceptions (mots-clefs `try`, `catch` et `finally`)

```
int[] tab;
boolean searchTab(int val) {
  try {
    //...
    if (tab[i]==val)
      out.print(i);
    //...
  }
```

Capture des exceptions (2)

```
 catch(NullPointerException e) {
    System.err.println("tab==null");
    return false; }
  catch(ArrayIndexOutOfBoundsException e){
    //...
   }
  catch(IOException e) {
    //...
   }
 finally { out.close(); }
}
```


Traitement des exceptions

- si exception générée dans le bloc `try` :
  1/ on passe immédiatement dans le premier gestionnaire `catch` compatible avec l'exception (même classe, ou classe ancêtre),
  2/ on exécute le `catch`,
  3/ puis on continue APRES 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 pile 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 une exception non traitée)

Lancement d'exception

- mot-clef `throw`

```
if (test_anomalie)
  throw new Exception("blabla");
```

- interrompt immédiatement le cours normal du programme pour rechercher un gestionnaire adéquat englobant
- lancer de préférence une exception d'un type spécifique à l'anomalie

Catégories d'exceptions

- classe mère : `Throwable`
- erreurs "système" : classe `Error` (dérivée de `Throwable`)
- autres anomalies : classe `Exception`
  cas particulier : classe `RuntimeException` et ses sous-classes
- type d'exception créé par le programmeur

Catégories d'exceptions (2)

- méthodes communes :
  - `NomException(String)` : constructeur avec message explicatif sur la cause de l'exception
  - `String getMessage()` : renvoie le message explicatif en question
  - `void printStackTrace()` : affiche la pile d'appel jusqu'au point de lancement de l'exception

Exceptions usuelles

- `NullPointerException`
- `NegativeArraySizeException`
- `IndexOutOfBoundsException`
- `ArrayIndexOutOfBoundsException`
- `StringIndexOutOfBoundsException`
- `IllegalArgumentException`
- `ArithmeticException`
- `NumberFormatException`
- `IOException`
- `InterruptedException`

Création de types d'exception

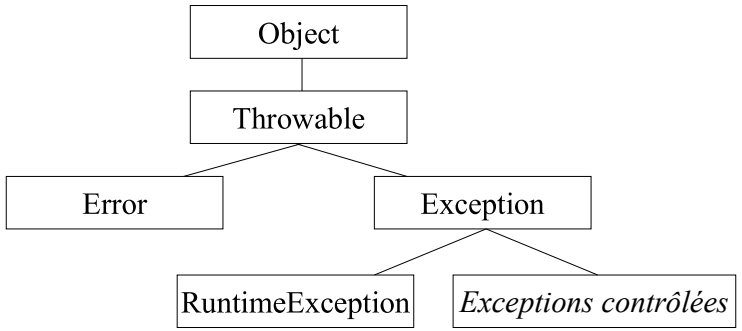
- nouveaux types d'exception : il suffit de créer une sous-classe de `Exception` ou de `RuntimeException` (ou d'une de leurs sous-classes prédéfinies) :

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

  //...

  if (test_anomalie)
   throw new MonException("commentaire");
```


Déclaration des exceptions

- Le mot-clé `throws` permet de déclarer les exceptions dans la signature des méthodes
- les exceptions dérivant de la classe `Exception` 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 les méthodes appelées, et non traitées)

Exceptions contrôlées



Exemple d'utilisation de `throws`

```
void lire() throws MonException, IOException {
    // lancement explicite
    if (test_anomalie)
      throw new MonException();
    // appel de méthode susceptible de
    //  générer une exception contrôlée
    int car=System.in.read();

}

catch(MonException e){

   e.printStackTrace();

   System.exit(1);

}
```

THREADS

Programmation multi-threads

- un thread = 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 interfaces), 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 millisecons)`
- 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 implémentant 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) {
     // ...  }
  }
}
```

Suspension/arrêt de thread (2)

```
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) {}
    }
 }
```

Exemple de thread (2)

```
public int valeur(){
  return compteur;}
public void suspendre(String s){
  System.out.println("Suspension de
  "+s+":"+valeur());
  pause = true;}
public void redemarrer() {
    pause=false;}
public void tuer(String s) {
  System.out.println("Arret de "+s+"
  :"+valeur());
  stop = true;}
```


Exemple de thread (3)

```
public static void main(String args[]){
  Compteur c1=new Compteur();
  Compteur c2=new Compteur();
  Compteur c3=new Compteur();
  c1.start();
  c2.start();
  c3.start();
  c3.suspendre("c3");
  try{
   sleep(60);
  }
  catch (InterruptedException e) {}
```

Exemple de thread (4)

```
    c3.redemarrer();
    c2.suspendre("c2");
    try{
     sleep(100);
    }
    catch (InterruptedException e) {}
    c2.redemarrer();
    c3.tuer("c3");
    c2.tuer("c2");
    c1.tuer("c1");
   }
  }
```


Exemple de thread (5)

Suspension de c3:0
Suspension de c2:2
Arret de c3 :1
Arret de c2 :2
Arret de c1 :5

Suspension de c3:0
Suspension de c2:2
Arret de c3 :2
Arret de c2 :2
Arret de c1 :4

Suspension de c3:0
Suspension de c2:2
Arret de c3 :1
Arret de c2 :2
Arret de c1 :4

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(); // déverrouille l'objet courant
    } catch(InterruptedException e) {
       return null;
    }
    return tete;
  }
```

Synchronisation des exécutions (2)

- 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 (3)

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

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

}
```

Exemple de synchronisation

```
class Compte {
  int capital = 0;
// cette méthode incrémente capital de deCombien
  void credite(Banquier leBanquier, int
 deCombien) {
    int montant;
    System.out.println(leBanquier.nom + "
 commence son travail");
    montant = capital;
    leBanquier.yield();
    System.out.println(leBanquier.nom + "
 continue son travail");
    capital = montant + deCombien; } }
```


Exemple de synchronisation (2)

```
class Banquier extends Thread {
  Compte unCompte;
  String nom;
  Banquier(Compte unCompte, String nom) {
    this.unCompte = unCompte;
    this.nom = nom;
  }
  public void run() {
    System.out.println(nom + " est embauche");
    unCompte.credite(this,1);
  }
}
```

Exemple de synchronisation (3)

```
class BanquierTer extends Banquier {
  BanquierTer(Compte unCompte, String nom) {
    super(unCompte, nom);
  }
  public void run() {
    System.out.println(nom+" est embauche");
    synchronized(unCompte) {
      unCompte.credite(this, 1);
    }
  }
}
```


Exemple de synchronisation (4)

```
class BanqueTer {
  public static void main(String[]arg) {
    Compte unCompte = new Compte();
    BanquierTer Jean, Jacques;
    (Jean = new BanquierTer(unCompte,
"Jean")).start();
    (Jacques = new BanquierTer(unCompte,
"Jacques")).start();
    try {
      Jean.join(); Jacques.join();
    } catch(InterruptedException e){}
    System.out.println("Votre capital est de "+
unCompte.capital); } }
```

Exemple de synchronisation (5)

- donne à l'exécution :

```
Jean est embauche
Jean commence son travail
Jean continue son travail
Jacques est embauche
Jacques commence son travail
Jacques continue son travail
Votre capital est de 2
```


Blocages

- la programmation avec threads nécessite de faire très attention à ce qu'aucun deadlock ne puisse se produire
- cas typiques de blocage :
  - t_1 attend la fin de t_2 et réciproquement
  - t_1 attend la fin de t_2 alors qu'il a un lock sur un objet sur lequel t_2 attends de pouvoir mettre un lock
  - t_1 suspend t_2 pendant qu'il a un lock sur o, puis essaie de prendre un lock sur le même o

Priorités des threads

- + sa priorité grande, + 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

Exemple de classe `Runnable`

```
class RepetiteurR implements Runnable {
  String chaine;
  RepetiteurR(String chaine) {
    this.chaine = chaine;
  }
  public void run() {
    for (int i = 0; i < 2; i++) {
      System.out.println(chaine);
      Thread.currentThread().setPriority(
  Thread.MIN_PRIORITY);
    }
  }
}
```

Exemple de classe `Runnable` (2)

```
class EcrivainR {
  public static void main(String[] arg) {
    new Thread(new RepetiteurR("soleil")).start();
    new Thread(new RepetiteurR("neige")).start();
    new Thread(new RepetiteurR("ski")).start();
    System.out.println("A la montagne");
  }
}
```

```
soleil
soleil
neige
neige
```

```
A la montagne
ski
ski
```


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, ...)