Programmation Par Objets Et Langage Java
Partie II. Introduction À Java

Najib Tounsi

Lien permanent : http://www.mescours.ma/Java/PooJavaPart-2-tdm.html

1. Table Des Matières

2. Langage De Programmation Avec Des TADs.

Le langage se prête bien à la programmation avec les types abstraits de données.

- Module (ou TAD) = ***classe*** (ou ***interface***)
- Opération = ***méthode*** sur un objets
- Structure de données = propriétés (***attributs***) de l'objet

3. Exemple De La Classe *Pile*

3.1. La Classe Pile (Source Complet, Documentation Associée)

```
public class Pile {

//
// Déclarations des attributs de la pile
//
    static final int MAX = 8;
    char[] t;
    int top;

//
// Programmation des opérations (méthodes) de la pile
//

    public Pile(){
        // Initialise une pile vide
        t = new char[MAX];
        top = -1;
    }

    public void empiler(char c){
        // Empile le caractère donné
        if (!estPleine())
            t[++top] = c;
        else
            System.out.println("Pile pleine");
    }

    public char sommet(){
        // Retourne le caractère au sommet de la pile
        if (!estVide())
            return t[top];
        else {
            System.out.println("Pile vide Sommet");
            return '\0';
        }
    }

    public void depiler(){
        //    décapite la pile (retire le sommet )
        if (!estVide())
            top--;
        else
            System.out.println("Pile vide Depiler");
    }

    public boolean estVide(){
```

```
        //    Teste si la pile est vide
        return (top < 0);
    }

    public boolean estPleine(){
        // teste si la pile est pleine
        return (top >= (MAX - 1));
    }
}; // Fin class Pile
```

A noter:

1. Aspects présentation
   - Une classe est un bloc dans lequel sont déclarée des données et des fonctions qui les utilisent. Les spécifications aussi, sous forme de commentaires ici.
   - C'est donc une implantation d'un TAD, i.e. choix de représentation (tableau `t` + indice sommet de pile `top`) de la pile et programmation des méthodes.
   - Une seule unité de compilation
   - La syntaxe des instructions est proche de C.
2. Ici on a tout mis dans un même fichier. Avec les interfaces Java, on peut séparer en deux partie: Spécification abstraites & Représentation. (voir plus tard.)
   - Deux unités de compilation.

Remarque: Dans ce dernier cas, il est possible de définir plusieurs implantations (représentations) pour un même TAD (spécification abstraite).

4. Programme Java Qui Utilise La Classe Pile

Un programme principal (classe avec fonction *main*.)

Lire un mot et imprimer son image miroir.

```
public class TestPile {
    static public void main(String args[]){
         // Imprime l'mage miroire d'un mot

         char c;
         Pile p = new Pile();

   // On lit les lettres et on les empile
         while ((c=Support.readChar()) != '#') {
                 p.empiler (c) ;
                 if( p.estPleine()) break;
         }
```

```
    // On depile les lettres et on les imprime
          while (!p.estVide()) {
                c = p.sommet();
                System.out.println (c);
                p.depiler();
          }
      }
}
```

5. A Noter

- On voit qu'on n'a pas besoin d'utiliser un tableau `t[]` et une variable `top`.
  - Plutôt un objet `p` de classe (type) `Pile`.
- L'objet *p* est créé avec l'instruction `new Pile()`.
- Les instructions `p.empiler(c)`, `p.sommet()` etc. sont les appels aux fonctions (*méthodes*) de manipulation de l'objet p.
  - Notation: **objet . méthode-à-appliquer**()
  - On dit qu'on envoie un message à l'objet.
- Quand la méthode `p.empiler(c)` est appelée, le corps de la méthode est exécutée (avec `t[++top] = c;`), où `t` et `top` sont les données représentant l'objet `p` sur lequel s'effectue de l'appel.
- On voit donc que c'est l'objet lui même qui est responsable des instructions qui le manipulent.
  - Les programmes, comme *main* ici, qui utilisent `p` n'ont pas à savoir comment `p` réagit à un message (est-ce qu'il fait `t[++top] = c;` ou autre.) Ce qui compte c'est l'effet final de l'envoi du message, i.e. L'empilement de `c` sur `p`.
- Note: `main` est elle même une fonction enveloppée (déclarée) dans une classe. Mais elles est `static`. (Voir plus tard).
  - Elle ne s'applique pas à un objet particulier.
  - C'est en fait, le point d'entrée dans un programme Java.
- what else ...

6. Spécificités De Java Autres Que La Notion De Classe.

6.1. *Programming In The Small...*

Dans la suite de ce chapitre, on va s'intéresser à l'aspect programmation procédurale du

langage Java

- Instructions de base du langage
- Compilation / exécution
- API Java

7. Hello World

Programme *main* dans une classe *HelloWorld* (fichier HelloWorld.java)

```
// Mon premier programme Java
class HelloWorld {
    static public void main(String args[]){
          System.out.println("Hello, World");
    }
}
```

- Un programme Java est constitué d'au moins la fonction `void main ()`.
- Définie dans une classe `class HelloWorld`.
- Classe écrite dans dans un fichier source `HelloWorld.java` . Même nom que la classe ici.
- Instruction `System.out.println("Hello, World");` qui imprime ses arguments et va à la ligne.
- (L'entête de la méthode *main*() et l'appel à *println*() seront expliqués plus tard)
- `//` introduit un commentaire jusqu'à la fin de la ligne.
- `/*` permet un commentaire sur plusieurs lignes `*/`.

8. Un Programme Plus Complet (Avec E/S De Données)

Programme qui lit un nom et une température en Celcius, convertit cette dernière en Fahrenait et affiche bonjour avec cette nouvelle valeur. (source `Bonjour.java`)

```
class Bonjour {
    static public void main(String args[]) {
        int c;
        String nom;
        double f;

        // Partie lecture de données
```

```
        System.out.print("Quelle est votre nom?: ");
        nom = Support.readString();
        System.out.print("Quelle t° fait-il aujourd'hui?: ");
        c = Support.readInt();

        // Partie calcul
        f = 9./5 * c + 32;

        // Partie sortie des résultats
        System.out.println("Bonjour " + nom);
        System.out.print("Il faut " + c + "° Celsius,");
    System.out.println(" soit "+ f + "° Fahrenait.");
      }
}
```

- Types prédéfinis primitifs `int` et `double`. Types scalaires.
  - Autres types primitifs: `byte`, `short`, `char`, `int`, `long`, `float`, `double`, `boolean`.
  - Commencent par une lettre minuscule.
- Type prédéfini `String`. Classe prédéfinie en réalité.
- Usage de la classe `Support` pour la lecture de données (voir travaux dirigés)
  - `readInt()`, `readDouble()`, `readString()` ... sont des méthodes de cette classe
  - Fichier `Support.class` dans le même répertoire que le programme.
- `println()` *vs* `print()`. Impression avec ou sans saut de ligne.
- Opération `+` (de concaténation) pour imprimer plusieurs données.
- A retenir surtout:

Les instructions Java sont souvent des appels de méthodes sur des objets.

9. Un Programme Plus Complet (Exécution)

On compile et on exécute (en mode ligne commande)

```
$ javac Bonjour.java
$ java Bonjour
Quelle est votre nom?: Ali
Quelle t° fait-il aujourd'hui?: 17
Bonjour Ali
Il faut 17° Celcius, soit 62.6° Fahrenait.
$ _
```

- La commande `javac` sert à *compiler* le fichier source `Bonjour.java` .
- La commande `java` sert à *exécuter* la méthode *main* de la classe `Bonjour` qui vient

d'être compilée.

- L'absence d'une telle fonction dans cette classe est une erreur :
  `Exception in thread "main" java.lang.NoSuchMethodError: main`

- Plus sur ce sujet plus tard.

10. Conversions Entre Entiers Et Réels

- Dans la formule utilisée dans programme précédent `f = 9./5 * c + 32` , on a écrit `9./5` pour que $9 \div 5$ donne un réel double précision. L'expression est alors évaluée en réel `double` .
- Voici quelques conversions élémentaires

```
int i;
double d;
i = 1.23;         // Erreur: Javac trouve un double là où il
                  // int. En effet, 1.23 est un littéral réel
d = 2;            // OK 2.0, conversion implicite
i = 3;            // OK
d = 1.23;         // OK
i = d;            // Idem. Erreur: risque de perte de précisi
                  // d peut contenir un réel très précis.
d = i;            // OK 3.0, conversion implicite aussi.
i = (int) -1.23; // OK valeur tronquée i = -1.
i = (int) d;      // idem, conversion (troncature) forcée (ca
```

- *Cast* = conversion explicite.

11. Typage Fort

- Java est un langage à typage fort.
- Chaque variable a un type.
  - Sa valeur doit être de même type. (*Identity conversion).*
- Dans une expression, le calcul doit se faire sur des valeurs de même type. Sauf cas simples, comme les valeurs numériques.
  - `int i = 2 * "A";` est une erreur.
- Le passage entre entiers et réels (ou caractère et entier) est plus souple.
  - de int à double, possible toujours (implicite): `double d = 123;` .
  - de double à int doit être forcée (explicite): `int i = (int) d;` .
  - de char à int aussi.
  - Dans une expression avec mélange entiers et réels, les entiers sont promus réels.

(*Widening conversion*)

- `i = 6 / 2.5; // erreur car calcule en double ensuite perte de précision lors de l'affectation.`
- `i = 6 / (int) 2.5;  // OK. i = 3`
- `i = (int) (6 / 2.5); // OK i = 2 !`

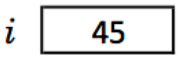
- Il existe des classes `Integer`, `Double` (noter la première lettre en majuscule), `Character` etc... (appelées *Wrappers*) permettant de considérer les types primitifs comme des classes. Avec méthodes de conversions etc.
  - Double d = **Double.valueOf** ("12.34"); // d = 12.34
  - int i; Integer j = new **Integer**(45); i = **j.intValue** (); // i =45
  - int i = **Integer.parseInt** ("456"); // i = 456
- Voir plus loin.

12. Les Types De Données En Java

Il ne convient pas toujours de parler de type de données pour un langage de classes. Néanmoins, il convient de distinguer, pour des raisons de performance, le traitement des données élémentaires des autres objets. Il y a deux genres de variables en Java:

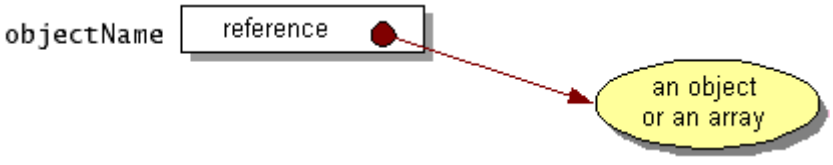
- Les données de **types primitifs**
  - byte, short, int, long, float, double, char, boolean

  - `int i = 45;`



13. Les Types De Données En Java (Suite)

- Les **références**
  - En fait, les types d'objets définis par l'utilisateur, i.e.**les classes**, ou les types d'objets prédéfinis, tableaux (arrays) et chaînes (Strings).



- Tous ces objets instanciés par new.

- `Pile maPile = new Pile();`
- `int p[] = new int [7];`

- A ce propos, à chaque type primitif correspond un type référence. Ce sont justement les *wrappers* `Boolean`, `Byte`, `Integer`, `Double` etc... Ils jouent un intérêt particulier dans la manipulation des types primitifs comme objets.

14. Les Types De Données En Java (Suite)

- Toutes les variables doivent êtres initialisées avant leur utilisation. Par *new* pour les références.
  - Les champs dans les objets peuvent ne pas être initialisés. Ils reçoivent une valeur par défaut. O, `false` ou `null` selon le cas. Voir les constructeurs par défaut plus loin.
- Les **tableaux** sont considérés comme des objets (prédéfinis) et s'utilisent donc avec des références.
  - Font partie, avec le types primitifs, des types de base de Java.
  - Initialisés par new ou littéralement.
  - `char p[] = new char[20]; int t[] = {1,2,5};`
  - Seront vus plus loin.
- Les **chaînes de caractères** sont objets de la classe `String` fournie en Java. Peuvent être considérées aussi comme type de base de Java.

15. Les Types Primitifs

- Les types primitifs sont prédéfinis en Java et désignés par un mot réservé.
- Ce sont les types numériques (entiers et réels) et le type `boolean`.
- Les entiers sont: `byte`, `short`, `int`, `long` et `char`.
- Les réels sont: `float` et `double`.
- Le type `boolean` a deux valeurs: `true` et `false`.

16. Les Types Primitifs (Suite)

- Le type ***byte***

- Le type `byte` est un entier sur 8 bits compris entre -128 et 127 inclus.

- Le type ***short***
  - Le type `short` est un entier sur 16 bits compris entre -32.768 et 32.767 inclus. Avec le type byte, peut servir pour économiser de la mémoire dans les grands tableaux.
- Le type ***int***
  - Le type `int` est un entier sur 32 bits compris entre -2.147.483.648 et 2.147.483.647 inclus. C'est le type d'entier généralement utilisé.
- Le type ***long***
  - Le type `long` est un entier sur 64 bits compris entre -9.223.372.036.854.775.808 et 9.223.372.036.854.775.807 inclus.
- Le type ***float***
  - Le type `float` est un réel simple précision sur 32 bits. Valeur par défaut = 0. A utiliser pour gagner de la place mémoire pour de grands tableaux.
- Le type ***double***
  - Le type `double` est un réel double précision sur 64 bits. Valeur par défaut = 0. C'est le type de réels généralement utilisé.
- Le type ***boolean***
  - Le type `boolean` qui n'a que les deux valeurs `true` et `false`.
- Le type ***char***
  - Le type `char` est un caractère Unicode (UTF-16) sur 16 bits.
  - C'est le type qui représente tous les caractères de toutes le langues. La valeur de chaque caractère est appelée *Codepoint*. Peut être notée `'\uxxxx'` où xxxx est la notation hexadécimale de la valeur.

17. Les Constantes Numériques

- Constantes entières

```
boolean result = true;
char capitalC = 'C';
byte b = 100;
short s = 10_000;
int i = 20;
long l = 5_000_000_000L;
```

- Une constante entière est de type `long` si suivie de `L` (ou `l`).

- Le symbole `_` de séparation des milliers peut être mis partout au milieu des chiffres, pour une meilleure lisibilité.
- Les valeurs hexadécimales (resp. binaires) précédée de `0x` (resp. `0b`)

```
int decVal = 26;
int hexVal = 0x1a;
int binVal = 0b11010;
```

18. Les Constantes Numériques (Suites)

- Les constantes réelles (`float` ou `double`) peuvent s'exprimer en notation scientifique (avec la lettre `E` ou `e` pour l'exposant), ou utiliser les lettres `F` ou `f` (pour flottant), ou la lettre `D` ou `d` (pour double, la cas par défaut).

```
double d1 = 123.4
double d2 = 1.234e2;   // même valeur en notation  scientifiqu
float f1  = 123.4f;
```

- !!

```
float f = 1234;    // OK. Conversion de int vers float.
float f = 123.4;   // erreur. Perte de précision.
                   //123.4 contstante double précision.
```

19. Les Constantes Caractères

- Les constantes littérales caractères, un caractère entre simple quote '.

```
'a', '%', '\t', '\\', '\'', '\u03a9', '\uFFFF', '\177', 'Ω', 'ð', 'é', 'ҷ'
```

  - NB. Utiliser l'option de compilation `-encoding utf8` pour faire accepter les caractères non US_ASCII.

- Pour certains caractères spéciaux, retour à la ligne par exemple, utiliser `'\n'` au lieu de la notation Unicode `'\u000a'` correspondante. Les séquences *escape* Unicode sont traitées et remplacées avant compilation. `'\u000a'` devient saut de ligne dans le source. D'où un risque d'erreur de compilation.
- Une constante caractère est toujours de type `char`.

20. Les Constantes Chaînes

- Les constantes littérales chaînes de caractères. Contiennent 0 ou plusieurs caractères

entre double quote ".

```
""                      // chaîne vide
"\""                    // chaîne avec un seul car. "
"This is a string"      // chaîne avec 16 caractères
"This is a " +          // une expression  constante  String,
    "two-line string"   // formée de deux littéraux
```

- Une constante chaîne est toujours de type `String`. Donc une référence à une instance de classe `String`.
- ...@@@...
- Les constantes chaînes sont toujours «internées» (*interned*). Chaque chaîne est stockée en *une seule* copie.
- Constante particulière: `null`. Peut être affectée à n'importe quelle variable référence.

21. Conversions Et Promotions

- Toute expression a un type qui est déduit du type de ses opérandes.
- Quand un type n'est pas approprié à son contexte, c'est généralement une erreur de compilation. Jave est un langage à typage fort.
- Certaines conversions, dites implicites, sont permises par le langage si le contexte le permet, e.g. conversions entre données numériques compatibles.
- Sinon, la conversion doit être explicite (*cast*).
- Les conversions sur les types primitifs se font sans danger dans le sens d'une promotion (*widening conversion*), sinon (*narrowing conversion*) il y a perte de précision signalée par le compilateur. Forcer la conversion dans ce cas. Exemple:

```
int i = 'A';    // conversion de char vers int (i=65)
float f ; double d = 1234.567890;
f = d       // Erreur perte de précision
f = (float) d;  // OK  mais  f = 1234.567871
```

- D'autres conversions se font vers `String`. L'opérateur de concaténation + appliqué à une chaîne et un autre objet primitif, convertit cet objet vers une chaîne.

22. Conversions Et Promotions (Suites)

- Les promotions sont des conversions implicites sur les types primitifs
  - de `byte` à `short`, `int`, `long`, `float`, ou `double`
  - de `short` à `int`, `long`, `float`, ou `double`
  - de `char` à `int`, `long`, `float`, ou `double`

    - `char` promu directement vers `int` et non pas vers `short`
  - de `int` à `long`, `float`, ou `double`
  - de `long` à `float` ou `double`
  - de `float` à `double`

- Exemple

```
long i = 25;    // int vers long OK
```

- !! conversion entiers grands vers `float`.

```
int i = 1234567890;
float f = i;          // int vers float OK.
  // ici f vaut 1.23456794E9 et (int)f vaut 1234567936
```

23. Conversions Et Promotions (Suites)

- C'est parfois intéressant de pouvoir manipuler des nombres sous leur forme chaîne de caractères.
- L'opérateur de concaténation + appliqué à une chaîne et un autre objet, convertit cet objet vers `String`.

```
String s1 = "Douze = " + 12; // s1 = "Douze = 12"
String s2 = "" + 12;     // s2 = "12"
```

- Il ya aussi les méthodes de conversion vers chaîne. Méthode `valueOf()` de `String`.

```
String s2 = String.valueOf(i);
```

- ou la méthode `toString()` de chaque sous-type de `Number`. Exemple:

```
int i;
double d;
String s3 = Integer.toString(i);
String s4 = Double.toString(d);
```

24. Conversions Et Promotions (Suites)

- Un autre type de conversion s'effectue par exemple de `int` vers `Integer` , de `float` vers `Float`, de `char` vers `Character`, etc. c'est à dire d'un type primitif vers le type Wrapper correspondant.
- On appelle *boxing*, ce type de conversion. La conversion inverse est appelée *unboxing*. Exemple:

```
Integer I = new Integer(45);  // I = 45  // Boxing de 45
int i = I.intValue ();        // i est aussi 45 // Unboxing de
```

- Au fait, depuis Java1.5, on peut écrire directement `I = i;` ou `i = I;` sans passer par *new Integer* ou par *intValue*().
- 1. On dit *Autoboxing* / *unboxing*.

25. Les Opérateurs Java

(Extraits de Java Tutorial http://docs.oracle.com/javase/tutorial/java/nutsandbolts/op1.html)

- Simple Assignment Operator

```
=       Simple assignment operator
```

- Arithmetic Operators

```
+       Additive operator (also used for String concatenation)
-       Subtraction operator
*       Multiplication operator
/       Division operator
%       Remainder operator
```

- Unary Operators

```
+       Unary plus operator; indicates positive value
        (numbers are positive without this, however)
-       Unary minus operator; negates an expression
++      Increment operator; increments a value by 1
--      Decrement operator; decrements a value by 1
!       Logical complement operator; inverts the value of a boo
```

- Equality and Relational Operators

```
==      Equal to
!=      Not equal to
>       Greater than
>=      Greater than or equal to
<       Less than
<=      Less than or equal to
```

- Conditional Operators

```
&&      Conditional-AND
```

```
||      Conditional-OR
?:      Ternary (shorthand for if-then-else statement)
```

- Type Comparison Operator

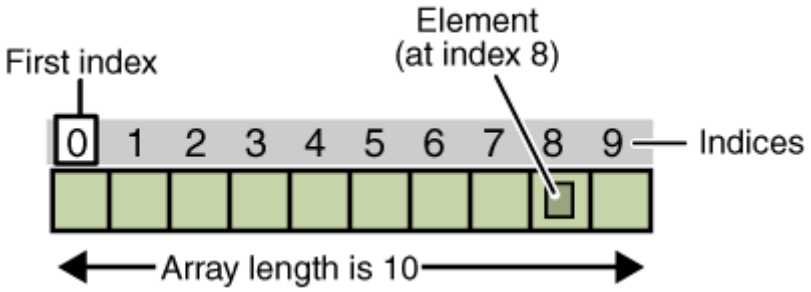
```
instanceof      Compares an object to a specified type
```

- Bitwise and Bit Shift Operators

```
~       Unary bitwise complement
<<      Signed left shift
>>      Signed right shift
>>>     Unsigned right shift
&       Bitwise AND
^       Bitwise exclusive OR
|       Bitwise inclusive
```

26. Les Tableaux En Java (Déclaration)

- Un tableau en Java est un objet qui peut contenir un certain nombre de valeurs, dits éléments, d'un même type. Ce type peut être quelconque.
- La taille d'un tableau est fixée à la création.



(extrait de http://docs.oracle.com/javase/tutorial/java/nutsandbolts/arrays.html)

- Les indices des différents éléments commencent à partir de 0. Le 9e élément est accédé par *t* [8], si *t* est est le nom du tableau. Les indices sont de type `int`.
- Les tableaux en Java sont des objets (variable référence donc) créés dynamiquement.
- Une déclaration se fait par le declarateur [ ].

```
int ai[];         // tableau d'entiers. (Vecteur)
short[] as, ab;   // 2 tableau de short.
```

- Les éléments de tableau peuvent être eux même des tableaux. Tableau à plusieurs dimensions.

```
short[][] matrice;     // tableau deux dimension de short.
int[][][] cube;        // tableau trois dimension de int.
```

- le declarateur peut apparaître avec le type ou avec la variable (ou les deux, non recommandé).

```
int[] ai, mi [];     // Notation mixte. Equivalent à
int ai[], mi [][];
                     // Equivalent aussi à
int [] ai;
int [][] mi;
```

- Les déclarateurs [ ] font plutôt partie du type. Par convention doivent apparaitre avec le type. Notation des deux dernières lignes.

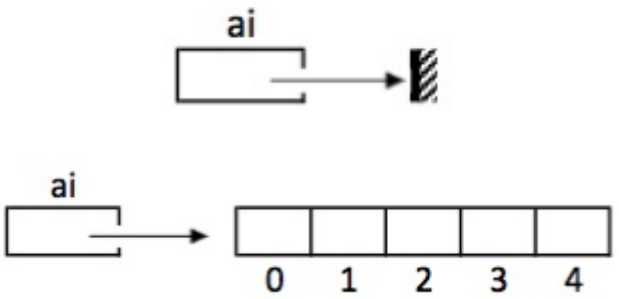
27. Les Tableaux En Java (Création Et Initialisation)

- Un tableau peut être créé par ***new*** (tableau vide) ou par énumération de ses éléments à la déclaration.
- **Création par *new***. Un tableau est un objet après tout.

```
int[] ai = new int[5];   // ai tableau de 5 entiers.
```

- Equivalent à:

```
int[] ai;          //  ai est ici juste une référence
ai = new int[5];   //  ai désigne un tableau de 5 entiers.
```



Référence tableau avant et après new

- Après *new*, la zone du tableau est remplie par des 0, des *false* ou des *null*; selon le cas.
- Exemple d'affectation explicite d'éléments:

```
ai[0] = 2;   // premier élément est 2
ai[1] = 3;   // deuxième élément est 3
ai[2] = -1;  // etc.
```

- Une fois créé un tableau, sa taille est fixée et accessible dans le champ ***length***.

```
System.out.println(ai.length);   // imprime 5
```

- Matrice

```
int m[][] = new int [2][3];  // 2 lignes et 3 colonnes
m[0][0] = 11;
m[1][2] = 23;
 //etc.
System.out.println(m.length);     // 2 (nombre de lignes)
System.out.println(m[1].length);  // 3 (nombre de colonnes)
```

- !! Le débordement de tableau est controlé en Java. Exception *ArrayIndexOutOfBoundsException*.

28. Les Tableaux En Java (Création Et Initialisation)

- **Création par énumération des éléments**. L'autre façon de créer (et remplir) les tabeaux en java, c'est de les initialiser à la déclaration.

```
int[] factorial = { 1, 1, 2, 6, 24, 120, 720, 5040 };

char aoc[]      = { 'c', 'e', ' ', 'n', '\'', 'e', 's', 't',
                    ' ', 'p', 'a', 's', ' ', 'u', 'n', 'e',
                     ' ', 'c', 'h', 'a', 'i', 'n', 'e'};

String[] aos    = { "array", "of", "String", };
```

- Là aussi, la taille du tableau est fixée à la déclaration/initialisation par le nombre de valeurs entre { et }.
- Pour les tableaux à plusieurs dimensions:

```
int[][] m = {
             {1, 2, 3)},
             {4, 5, 6}
};                        // matrice à deux lignes et 3 colonnes

String[][] noms = {
    {"M. ", "Mme ", "Mlle"},
    {"BenAhmed", "BenAli"}
 };                 // ici il y a 2 lignes, mais de longueur d
```

- A la différence avec les tableaux de C, il est possible d'avoir des lignes de longueur différentes. Si *nom* [0][2] ("`Mme`") est défini, *nom* [1][2] est un débordement de tableau.

29. Copie De Tableaux

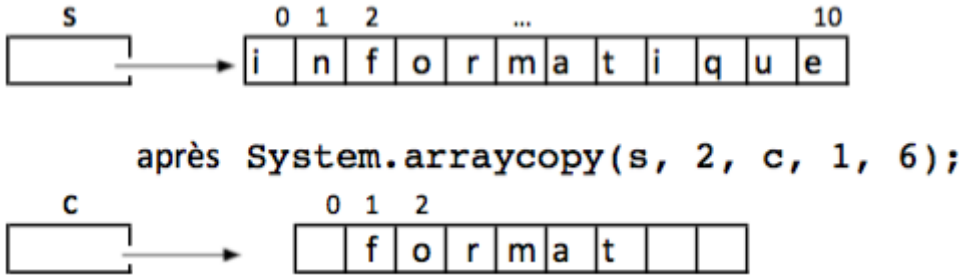
- A la différence de C, les tableaux peuvent s'affecter. C'est une copie de référence.

```
String [] noms = {"Ali", "Amina"};
String [] mesNoms = new String [3];
mesNoms = noms;
```

  - Il y a un seul tableau en mémoire, mais deux variables *noms et mesNoms* pour les désigner. Si on fait `noms [1] = "Fatima";` , la valeur *mesNoms* [1] a changé aussi pour `"Fatima"` .
  - NB. Quoique le tableau *mesNoms* a 3 éléments, après affectation il n'en a que 2. (exercice: Pourquoi?)
- Pour une vrai copie de tableau, utiliser la classe *System* de Java et la méthode *arraycopy*() qui permet de copier, dans le sens dupliquer, un tableau (ou une zone de tableau). Elle a pour profile

```
public static void arraycopy( Object src, int srcPos,
                              Object dest, int destPos, int le
```

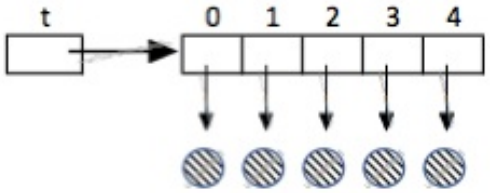
- où on spécifie le tableau source `src` et le tableau cible `dest` , l'indice du début de la zone à copier, l'indice de l'endroit dans le tableau final où doit se faire la copie et la taille de la zone à copier.



- ici, on a copié depuis *s*, indice 2, vers *c*, indice 1, une zone de 6 éléments.

30. Tableaux D'objets Non Primitifs

- Les éléments d'un tableau d'objets non primitifs doivent, *en plus* du tableau lui-même, être initialisés individuellement par *new*.



- C'est le cas d'ailleurs d'un tableau à deux dimensions (une matrice) ou d'un tableau de `String`.

```
int[][] m = new int[2][];  // Creation deux références lignes
                           //  m[0] et m[1].
m[0] = new int[3];       // Création trois éléments entiers
m[1] = new int[2];       // Création deux éléments entiers
                         // Tous initialisés à nulles
```

- ou d'un tableau de `String`.

```
String[] s = new String[3];  // trois éléments tous egaux à nu
s[1] = new String("toto");   // le deuxième éléments est une
                             // référence à "toto"
```

- En général (soit une classe `Point`)

```
Point [] p = new Point [10];    // vecteur p de 10 Point
for (int i=0; i<10; i++)
     p[i] = new Point();        // instanciation de chaque Poi
```

31. Création De Tableau Par New

- On peut instancier un tableau Java avec l'écriture

```
new Type [] {liste d'objets};
```

- qui crée un tableau d'objets du type donné, initialisés avec les objets données.
- Usage:

```
String s [] = new String [] {"abc", "def"};
```

- qui crée un tableau *s* de deux `String` initialisé aux valeurs fournies. Ce qui équivaut à:

```
String s[] = new String[2];
s[0] = "abc";
s[1] = "def";
```

- La déclaration

```
int t [] = new int[] { 11, 23, 45 };
```

- crée un tableau *t* de trois entiers 11, 23 et 45.

```
Point p [] = {new Point(1,2), new Point(2,3)}
```

- idem pour deux points de coordonnées (1,2) et (2,3).

32. Les Objets String ...

- Les chaînes de caractères, mots, phrases, textes, sont des séquences de caractères. En java, ils sont considérés des objets. Leur classe est prédéfinie: `String`.
- Les caractères sont des valeurs entières (*code points*) Unicode.
- Création d'une chaîne `String`. Le plus simple c'est par une contante entre double quotes.

```
String salut = "Bonjour";
```

- Qui équivaut à

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

- En effet,

  *une instance d'un objet se crée par **new**, et cela peut se faire à partir de plusieurs sources en paramètre. Ici à partir d'un littéral.*

- Un objet `String` n'est pas la même chose qu'un tableau de `char`.

```
char [] bonjour = {'b', 'o', 'n', 'j', 'o', 'u', 'r'};
String salut = bonjour;            // erreur compilation: incor
```

- Faire plutôt

```
char [] bonjour = {'b', 'o', 'n', 'j', 'o', 'u', 'r'};
String salut = new String (bonjour);       // :-)
```

33. Les Objets String

- Les objets `String` sont des objets immuables (ang. *immutable*), c'est à dire ils ne changent pas. Quand on manipule une chaîne, par conacténation par exemple (opérateur `+`), c'est une nouvelle chaîne qui est créée comme résultat de l'opération.

```
String s = "Bonjour";
s = s + " Ali";          // nouvelle chaîne "Bonjour Ali" désig
```

- Les constantes `Strings` sont "internés" (*interned*).
- En java il n'est stocké qu'un seul exemplaire de chaque valeur de chaîne distincte (qui doit être immuable donc). Efficacité.
- Les valeurs distinctes sont stockés dans un *pool* de chaînes.
- cf. méthode `String.intern()`.

34. Opérations Sur Les Chaînes: Length(), CharAt()

- ***length***(), méthode d'accès, qui retourne la taille d'une chaîne: `int length()`

```
String palindrome = "Dot saw I was Tod";
int len = palindrome.length();             // len vaut  17.
```

- Ne pas confondre avec le champ *length* pour un tableau, qui est une propriété et non une méthode (pas de parenthèses donc).
- La taille d'une chaîne est calculée en nombre de codes caractères.
- ***charAt*** (). Méthode d'accès qui retourne le caractère de rang *i* donné dans une chaîne. *i* à partir de 0. `int charAt(int index)`.

```
char c;
String salut ="Bonjour";
c = salut.charAt(2);   // c devient 'n'
```

- Exemple: programme palindrome.
  - Résultat: `"doT saw I was toD"`
  - Exercice: Vérifier le programme sur "Niagara. O roar again!".

35. Opérations Sur Les Chaînes: + Et Concat()

- ***concat*** (). Opération de concaténation. Méthode qui retourne une nouvelle chaîne par concaténation de deux autres. `String concat(String str)`.

```
String nom = " Ali";
String s = salut.concat(nom). // s est "Bonjour Ali"
"bon".concat("jour")          // retourne la chaîne "bonjour"
```

- Mais la notation la plus courante est l'opérateur `+`.

```
salut + " Ali"
salut + " " + "Ali"
"La racine de 2 est " + Math.sqrt(2)
  // donne "La racine de 2 est 1.4142135623730951"
```

- Cet opérateur `+` convertit vers `String` son autre opérande numérique.
- Attention! `+` est associatif à gauche. $a + b + c$ **est** $(a + b) + c$.

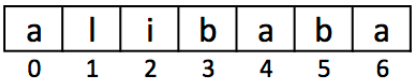
```
3 + 4 + " chats"              // est la chaîne "7 chats"
"chats " + 4 + 3              // est la chaine "chats 43"
```

36. Recherche Dans Une Chaîne: `indexOf()`

Méthodes ***indexOf***() et ***lastIndexOf***(). Recherche de caractère ou sous-chaîne dans une chaîne.

- `int indexOf(int car)` et `int indexOf(int car, int debut)`.
  - Retournent le rang (à partir de 0, ou à partir du rang spécifié) de la première occurrence du caractère donné. Sinon -1.
- `int lastIndexOf(int car)` et `int lastIndexOf(int car, int debut)`.
  - Retournent la dernière occurrence, à partir de 0 ou à partir de rang spécifié mais en arrière, du caractère donné.
- `int indexOf(String mot)` et `int lastIndexOf(String mot, int debut)"`.
  - Recherche de rang de première (ou dernière) occurrence de mot (ou sous chaîne). Mêmes spécifications.

```
"alibaba".indexOf('b')         // retourne 3
"alibaba".indexOf('b', 4)      // retourne 5
"alibaba".lastIndexOf('a')     // retourne 6
"alibaba".lastIndexOf('a', 5)  // retourne 4
```



37. Extraction De Sous Chaîne: `substring()`

Méthode ***substring***().

- "**`String substring(int indexDebut)`**" et
- "**`String substring(int indexDebut, int indexFin)`**".

```
String s ="/users/me/hello.java";
int dot = s.lastIndexOf('.');
int sep = s.lastIndexOf('/');
System.out.println (s.substring(dot+1));       // donne java
System.out.println (s.substring(sep+1,dot));   // donne hello
```

- Pour les autres méthodes sur les chaînes voir http://docs.oracle.com/javase/7/docs/api/java/lang/String.html

38. Les Classes StringBuffer Et StringBuilder

- Ce sont des chaînes de caractères modifiables. Un objet `StringBuffer` contient une chaîne dont le contenu et la longueur peuvent varier.
- Les principales opérations sur les `StringBuffer` sont ***append*** () et ***insert*** (), qui permettent de rallonger une chaîne ou y insèrer quelque chose. Méthodes surchargées pour accepter plusieurs types.

```
StringBuffer sb = new StringBuffer("Chat");
System.out.println(sb.append("on"));       // Chaton
System.out.println(sb.insert(3,"rles"));   // Charleston
System.out.println(sb.append(3.4567));     // Charleston3.456
```

- Les chaînes `StringBuffer` ont une capacité ***capacity***(). Cela permet de ne pas allouer de mémoire suplémentaire tant que la longueur ***length***() de la chaîne ne dépasse pas cette capacité. Sinon, il y a allocation d'espace suplémentaire.

```
StringBuffer sb = new StringBuffer("Salut!");
System.out.println(sb.length() + "  " + sb.capacity());     //
sb.append(" Comment ça va ?");
System.out.println(sb.length() + "  " + sb.capacity());     //
```

- Depuis JDK 5, La classe `StringBuilder`, plus performante, est recommandée à la place de `StringBuffer`.

Les *Wrappers* Java

On va donner des exemple pour la classe des entiers `Integer`. Les autres cas sont analogues.

- Des classes Java appelées *Wrapper* (≃ emballage) sont employées afin de présenter des valeurs primitives comme des objets. Pour les utiliser comme des références, les passer en paramètre là où il faut un objet ou leur appliquer des méthodes par exemple.

- Pour **`int`** on a la classe **`Integer`**, pour **`char`** on a la classe **`Character`**, pour **`float`** on a la classe **`Float`**, etc. (*truc*: première lettre majuscule).
- Un objet de ces classe est construit à partir d'un objet primitif correspondant ou à partir d'une chaîne.

```
Integer I = new Integer (2);
Integer J = new Integer ("23");
```

- Pour les types numériques, ces classes contiennent, entre autres, les valeurs maximums/minimums possibles.

```
 System.out.println(Integer.MAX_VALUE) ;   // 2147483647
 System.out.println(Integer.MIN_VALUE) ;   // -2147483648
 System.out.println(Integer.SIZE) ;        // 32
 System.out.println(Integer.TYPE) ;        // int
```

39. Conversions Par *Wrapper*

- La conversion de et vers le type correspondant de fait par des méthodes associées.

```
Integer I = new Integer (34);      // Integer à partir de 34
I = Integer.valueOf(34);                  // idem mais sans n
int i = I.intValue();              // retrouve le 34 int
```

- De `String` vers `Integer` et `int`

```
Integer I;
I = new Integer ("123");             // Par constructeur
I = Integer.valueOf("123");          // ou comme ceci

int i;
i = Integer.parseInt("456");                 // de String vers
```

- De `Integer et int` vers `String`

```
Integer I = 345;
String s = I.toString();   // s devient "345"

int i = 34;
s = String.valueOf(i);             // s devient "34"
```

- Depuis JDK 5 on peut écrire directement:

```
 I = 3;            // Autoboxing
 i = I;            // Unboxing
```

- On dit *autoboxing* / *unboxing*.

40. Utilitaires Des *Wrappers*

- La classe `Character` est intéressante dans le sens où elle offre les méthodes sur les caractères: ***isSpace***(), ***isUpperCase***(), ***isLetter***(), ***isDigit***() , ***hashCode***() etc.
  - `http://docs.oracle.com/javase/6/docs/api/java/lang/Character.html`

```
Character.isLetter('!')          // false
Character.isLetter('ج')          // true
Character.isDigit('5')           // true
Character c = 'e';
c.hashCode()                     // vaut 101
Character.toUpperCase('e')       // E
```

- et aussi un certain nombre de propriétés Unicode sur les caractères (e.g. `Character.DIRECTIONALITY_LEFT_TO_RIGHT`) .
- Se rappeler surtout

41. Les Instructions De Java.

- Les expressions Java forment la principale composante des instructions Java.
- Les instructions peuvent être groupées en blocs.

42. Expressions Java

- **Expression**: combinaison bien formée d'opérateurs et d'opérandes qui a pour résultat une valeur. Un opérande est une variabe ou un appel de méthode. Une expression a un résultat et un type.

```
x + y / Math.sqrt(2)

Character.isUpperCase(c)

int nombre = 5;
```

- L'expression `nombre = 5` est de type entier, car 5 (partie gauche de `=`) est évalué à entier.

43. Instructions Java

- **Instruction expression**: Unité complète d'exécution. Expression suivie du caractère point-virgule (;)
  - Expression d'affectation
  - utilisation de `++` ou `--`
  - Appel de méthode
  - Création d'objet
- Exemples
  - `delta = b*b - 4*a*c;`
  - `i++;`
  - `System.out.println (i);`
  - `Integer intObj = new Integer(4);`
- Il y a **trois types d'instructions** en Java:
- Instruction de déclaration
  - `double x = 11.234;`
- Instruction expression (cf. ci-dessus)
- Structure de contrôle. Régulation de l'ordre d'exécutions des instructions: séquence, choix, répétition, rupture de contrôle.

44. Bloc D'instructions

- **Bloc d'instructions**: suite de 0 ou plusieurs instructions entre accolades `{}` .
- Utile pour délimiter la portée d'une structure de contrôle.

```
if (a > b) {
    b++;
} else {
    a--;
}
```

- Ou d'une déclaration

```
{
     int i;
        // ... i visible dans ce bloc
}
```

45. Structures De Contrôle

- Syntaxe semblable à celle du langage C

Type d'instruction	**Mots clés**
Boucle	`while` , `do-while` , `for`
Choix d'exécution	`if-else` , `switch-case`
Gestion d'Exception	`try-catch-finally` , `throw`
Branchement	`break` , `continue` , `label:` , `return`

- Java recommande de toujours utiliser les {} pour délimiter le champ d'une structure de contrôle.

46. Instruction `If-else`

- Instruction *if-(then)*

```
if (moyenne >= 16){
    System.out.println ("Admis(e) avec félicitations");
}
System.out.println ("Admis(e)");
```

- Les accolades ne sont pas nécessaires s'il y a une seule instruction. Mais ne coûtent rien à les mettre.
- Instruction *if-(then)-else*

```
moyenne = 16.5;
if (moyenne >= 17){
  mention = "TB";
} else if (moyenne >= 15) {
     mention = "B";
} else if (moyenne >= 13) {
    mention = "AB";
} else {
    mention = "P";
}
System.out.println ("Admis(e) avec mention: " + mention);
```

- Résultat "Admis(e) avec mention: B".
- Même si 16.5 est aussi supérieure à 13, c'est le premier test satisfait qui s'exécute.
- Forme réduite de *if-(then)-else*

```
max = (a > b ? a : b);      // si (a > b)  alors a  sinon  b
```

47. Instruction Switch

```
switch (note) {
       case 20 : case 19 : case 18 : case 17 :
            System.out.println ("Mention Très Bien");
            break;
       case 16 :  case 15 :
            System.out.println ("Mention Bien");
            break;
      case 14 : case 13 :
            System.out.println ("Mention Assez Bien");
            break;
       case 12 : default :
            System.out.println (" Mention Passable\n");
            break;
}
```

- Comme en C, les cas peuvent-être factorisés.
- `switch` fonctionne avec un paramètre de type primitif: `byte`, `short`, `char` et `int`.
- ou un type énuméré

```
enum Taille { S, M, L, XL, XXL };
    ...
    public static void f(Taille t) {
        switch (t) {
            case S:
                System.out.println("Small = 1 ou 2");
                break;
            case M:
                System.out.println("Medium = 3");
                break;
            case L:
                System.out.println("Large = 4");
                break;
            case XL: case XXL:
                System.out.println("eXtra Large = 5 ou 6");
                break;
        }
    }
```

- Source complet

48. Instructions While Et Do-while

- Forme ***while***
  - **while** (expression) {
    instruction(s)
    }
- *Expression* pouvant être nulle au début. Rien à exécuter
- Exemple: Copie de chaîne

```
public class WhileDemo {
    public static void main(String[] args) {
        String copyFromMe = "Copy this string until you " +
                            "encounter the letter 'g', not inc
        StringBuffer copyToMe = new StringBuffer();
        int i = 0;
        char c = copyFromMe.charAt(i);
        while (c != 'g') {
            copyToMe.append(c);
            c = copyFromMe.charAt(++i);
        }
        System.out.println(copyToMe);
    }
}
```

- Résultat: `"Copy this strin"`

49. Instructions While Et Do-while

- Forme ***do-while***
  - **do** {
    instruction(s)
    } **while** (expression)
- Au moins *un* passage dans la boucle.
- Même exemple

```
public class DoWhileDemo {
    public static void main(String[] args) {
        String copyFromMe = "Copy this string until you " +
                            "encounter the letter 'g', include
        StringBuffer copyToMe = new StringBuffer();
        int i = 0;
        char c;
        do {
            c = copyFromMe.charAt(i++);
```

```
            copyToMe.append(c);
        } while (c != 'g');
        System.out.println(copyToMe);
    }
}
```

- Résultat: `"Copy this string"`

50. Instruction For

- Forme
  - **for** (initialisation; test-arrêt; incément) {
    instruction(s)
    }
- Exemple: Parcours d'un tableau

```
int[] t = { 32, 87, 3, 589, 12, 1076, 8, 622, 127 };
for (int i = 0; i < t.length; i++) {
     System.out.print(t[i] + " ");
}
System.out.println();
```

- Résultat: `"32 87 3 589 12 1076 8 622 127"`
- Forme améliorée, recommandée

```
for (int element : t){
   System.out.print(element + " ");
}
System.out.println();
```

- On a ici un *itérateur abstrait*...

51. Instructions De Traitement D'exception Try-catch-finally

- Pour une meilleure prise en compte des erreurs survenues lors de l'exécution d'une méthode.
- C'est le demandeur d'une méthode (qui envoie le message) qui doit gérer si cela s'est bien passée ou pas, et traiter la ou les exceptions (erreurs) survenues lors de l'exécution de la méthode.
- Forme générale
  - **try** {

```
    instruction(s)
} catch (nom de type d'exception) {
    instruction(s)
} finally {
   instruction(s)
}
```

- Trois instructions utilisées
  - **try**, indique un bloc d'instructions dans lequel une exception peut être levée (thrown), c'est à dire une erreur peut se produire dans une méthode appelée.
  - Un bloc **catch** doit être associé pour traiter l'exception produite. Un bloc catch par type particulier d'exception. Les instructions de ce bloc sont exéctées dans le cas où ce type d'exception s'est produit dans le **try** corespondant...
  - Un bloc **finally** (optionnel) est aussi associé pour exécuter des instructions finales aussi bien après **try** qu'après **catch**
- Voir plus loin un exemple.

52. Rupture De Contrôle (Instructions De Branchement)

- Trois type de branchement
  - **beak**
  - **continue**
  - **return**
- Pas de goto :-)

53. Instruction Break

- Arrête le déroulement normale d'une structure de contrôle.
- Exemple: Recherche d'un élément dans un tableau. ***break*** termine la boucle (extrait de *Java Tutorial*).

```
class BreakDemo {
    public static void main(String[] args) {

        int[] arrayOfInts =
            { 32, 87, 3, 589, 12, 1076, 2000, 8, 622, 127 };
        int searchfor = 12;

        int i;
```

```
        boolean foundIt = false;

        for (i = 0; i < arrayOfInts.length; i++) {
            if (arrayOfInts[i] == searchfor) {
                foundIt = true;
                break;
            }
        }

        if (foundIt) {
            System.out.println("Found " + searchfor + " at ind
        } else {
            System.out.println(searchfor + " not in the array"
        }
    }
}
```

- Résultat: `Found 12 at index 4.`
- Remplace bien `Goto`.

54. Instruction Break (suite)

- ***break*** avec étiquette (*label*) pour désigner la boucle, plus externe, à quitter. Exemple (extrait de *Java Tutorial*): Recherche dans une matrice. On arrête la recherche dans la boucle des colonnes ET aussi celle des lignes.

```
class BreakWithLabelDemo {
    public static void main(String[] args) {

        int[][] arrayOfInts = {
            { 32, 87, 3, 589 },
            { 12, 1076, 2000, 8 },
            { 622, 127, 77, 955 }
        };
        int searchfor = 12;

        int i;
        int j = 0;
        boolean foundIt = false;

    search:
        for (i = 0; i < arrayOfInts.length; i++) {
            for (j = 0; j < arrayOfInts[i].length;
                 j++) {
                if (arrayOfInts[i][j] == searchfor) {
                    foundIt = true;
```

```
                    break search;
                }
            }
        }

        if (foundIt) {
            System.out.println("Found " + searchfor +
                               " at " + i + ", " + j);
        } else {
            System.out.println(searchfor +
                               " not in the array");
        }
    }
}
```

- Autrement le *break* quitte la boucle interne et continue la boucle externe.
- Résultat: `Found 12 at 1, 0`.

55. L'INSTRUCTION CONTINUE

- Utile pour sauter l'itération courante et passer au tour suivant dans une boucle, *for*, *while*, *do while*.
- Exemple: Calcul du nombre de lettres 'p' dans une chaîne. La boucle traite les 'p's trouvé sinon passe à la lettre suivante

```
class ContinueDemo {
    public static void main(String[] args) {

        String searchMe
            = "peter piper picked a " +
              "peck of pickled peppers";
        int max = searchMe.length();
        int numPs = 0;

        for (int i = 0; i < max; i++) {
            // interested only in p's
            if (searchMe.charAt(i) != 'p')
                continue;

            // process p's
            numPs++;
        }
        System.out.println("Found " +
            numPs + " p's in the string.");
```

```
    }
}
```

- Résultat: `Found 9 p's in the string.`
- L'instruction `continue` marche aussi avec une étiquette (*label*) pour reprendre l'exécution d'une boucle plus externe. (Exemple).

56. L'instruction Return

- Permet de terminer l'exécution et quitter la méthode en cours.
- Le programme reprend après l'appel de la méthode quittée.
  - `retrun;`
- Envoi un résultat éventuellement.
  - `return ++count;`

57. Compilation Et Exécution D'un Programme Java

- Soit le fichier **`HelloWorld.java`** contenant la classe `HelloWorld` munie de la méthode `main` qui imprime `"hello, World"`.

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

- On compile de programme par la commande **`javac`** appliquée au fichier source. (Exemple donné sous langage de commande UNIX)

```
$ ls
HelloWorld.java
$ javac HelloWorld.java
$ ls
HelloWorld.class
HelloWorld.java
```

- Le résultat de la compilation est la création du fichier **`HelloWorld.class`** correspondant à la classe `HelloWorld`.
- On exécute le programme avec la commande **`java`** appliquée au fichier `.class` générée.

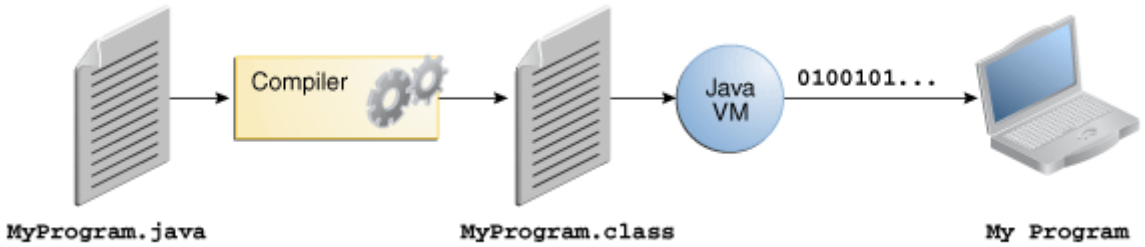
```
$ java HelloWorld
```

```
Hello, World
```

- A noter:
  - Le nom du fichier `HelloWorld.class` provient de nom de la classe dans le fichier source.
  - La commande `java` admet ce nom de classe comme argument.
  - La commande `java` exécute la méthode main de cette classe
- Au fait, le fichier `HelloWorld.class` est le module "*objet*" correspondant à la classe `HelloWorld`.
- En Java, à chaque classe source, correspond un fichier `.class` compilé par `javac`.

58. La Machine Virtuelle Java

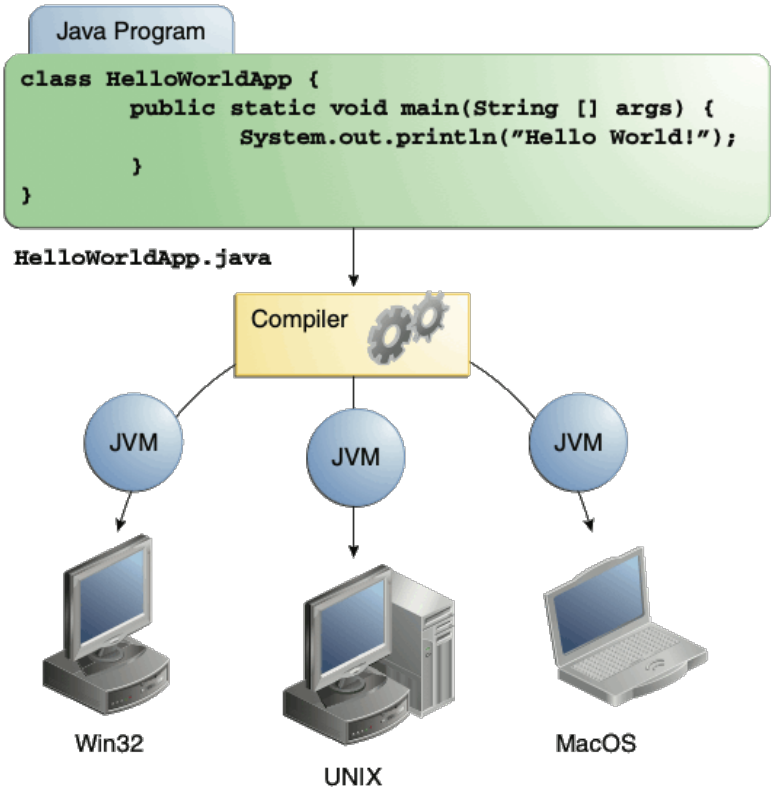
- Une classe source Java est compilée vers un langage qui n'est pas directement reconnaissable (exécutée) par un processeur donnée comme le serait un langage machine.
- C'est un langage intermédiaire, appelé *ByteCode*, qui se veut indépendant de toute plate-forme.
- Ce langage *ByteCode* est destiné à une *machine virtuelle* (VM) qui l'interprète pour un processeur donnée.



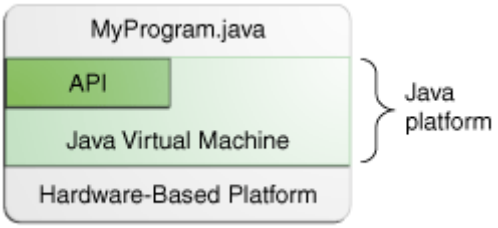
- La commande `java` justement est une réalisation de la VM sur une plateforme donnée.
- Un navigateur Web contient aussi une machine virtuelle pour exécuter des *applets*.

59. Write Once Run Everywhere

- La machine virtuelle Java est implantée sur plusieurs OS, le même fichier `.class` peut s'éxecuter sur *MacOS*, *Linux* ou *Windows* etc.



- *JVM* est la machine virtuelle fournie avec la plateforme Java.



(*Figures extraites de http://docs.oracle.com/javase/tutorial/getStarted/intro/definition.html*)

60. Environnement De Développement Intégrés *Vs.* Mode Commande

- On peut créer des programmes Java, les compiler et les exécuter directement sous le système d'exploitation sous-jacent, en mode ligne commande.
  - Commandes ***javac***, ***javadoc***, ***java***, ***jar*** etc..
- On peut aussi le faire en utilisant un logiciel approprié, un environnement de développement intégré, **IDE**, qui permet de construire une application complète sans se soucier du OS sous-jacent ou des détails d'organisation des fichiers sources.
  - On crée un source dans une fenêtre d'édition, on clique pour compiler et pour exécuter...
- Les IDEs rendent transparente la gestion des fichiers sources, des fichiers `.class` et des packages (répertoires) qui les contiennent.

    - *Eclipse*, *Netbeans*, *JCreator*, ...
- Dans le cadre de ce cours, nous utilisons le mode commande d'un OS. Cela permet de se concentrer sur le langage plutôt que sur les outils associés. En plus cela permet de mieux se rendre compte de ce qui se passe en coulisse des IDEs.

61. Surcharge De Fonction

- En principe, une méthode a un nom unique dans une classe.
- Cependant Java permet à une méthode d'avoir le même nom que d'autres grâce au mécanisme de surcharge (ang. *overload*).
- Java utilise leur signature, nom et paramètres de la méthode, pour distinguer entre les différentes méthodes ayant le même nom dans une classe.
- Ce sont le nombre et le type des paramètres qui permet de distinguer. Exemple (extrait de *Java Tutorial*):

```
class DataArtist {
   static void draw(String s) {
       System.out.println("Ceci est une chaîne: "+s);
   }
   static void draw(int i) {
       System.out.println("Ceci est un entier: "+i);
   }
   static void draw(double f) {
       System.out.println("Maintenant un double: "+f);
   }
   static void draw(int i, double f) {
       System.out.format("Une entier %d et un double %f %n",i
   }
}
```

- où la même méthode `draw` s'applique à plusieurs paramètres, chacun à sa façon.
- Les différents appels suivant correspondent aux bonnes fonctions:

```
DataArtist.draw ("Picasso");      // 1ère méthode, draw(Strin
DataArtist.draw (1);              // 2e méthode, draw(Int)
DataArtist.draw (3.1459);         // 3e méthode, draw(double)
DataArtist.draw (2, 1.68);        // 4e méthode, draw (int, d
```

- Le paramètre retour d'une fonction ne permet pas de distinguer entre deux fonctions. `static int draw(int)` est la même signature que `static void draw(int)`.
- La surcharge est surtout utile pour définir plusieurs constructeurs pour un objet.
- NB. Avec les interfaces et l'héritage, on considérera encore la surcharge.

62. Paramètres De Fonctions

- Les objets en Java sont passés en paramètre par *référence*.
- Mais les valeurs scalaires (objets primitifs) sont passées par *valeur*.
- Soit les deux méthodes qui modifient leur paramètres

```
public static void modifObj(int p[]) {
      p[0] = p[0] + 200;    // Objet référencé par p est modif
}

public static void modifVal(int x) {
      x = x + 200;          //  paramètre x modifié
 }
```

- Les appels suivants sont instructifs

```
int x = 2;
modifVal (x);
        // ici, x vaut toujours = 2

int [] t = {2, 3};
modifObj (t);      // t[0] a changé en 202
```

- Attention! Si on change LE paramètre

```
public static void modifObj(int p[]) {
    int t[]={200,300};
    p = t;                 // p[0] = 200
}
```

- Le même appel n'a pas le même effet maintenant

```
int [] t = {2, 3};
modifObj (t);       // t[0] vaut toujours = 2
```

63. Gestion Des Exceptions

- Exemple simple: Calcule de factorielle d'un entier positif ou nul et récupération du cas d'erreur "entier négatif".

```
try {
     int n = fact(5);
}
catch (ExceptionFactNegatif e) {
   System.out.println("Valeur negative pour factorielle");
}
```

- Avec la fonction *fact* qui soulève (***throw***) l'exception *ExceptionFactNegatif*

```
static public int fact(int x) throws ExceptionFactNegatif {
    if( x < 0 )
        throw new ExceptionFactNegatif();
    else
        // calcul factorielle f ...
        return f;
}
```

- et une classe `ExceptionFactNegatif` pour créer des objets porteurs de l'exception

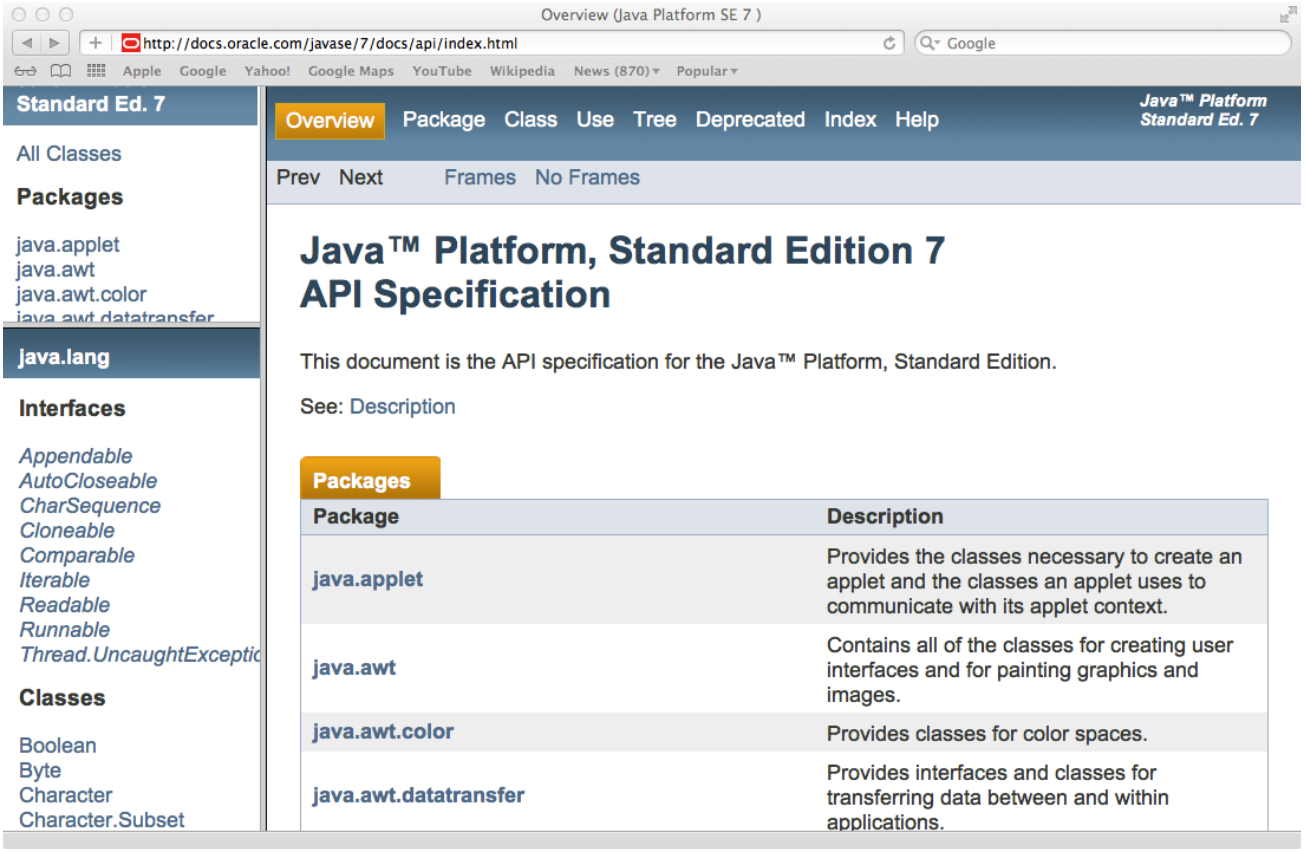
```
class ExceptionFactNegatif extends Throwable {};
```

- Sous-classe de ***Throwable***, pour pouvoir être soulevée comme exception.
- NB. Une fonction qui soulève une exception ne *doit pas* être appelée en dehors d'un bloc `try`.
- Exemple de la pile.

64. Les APIs Java

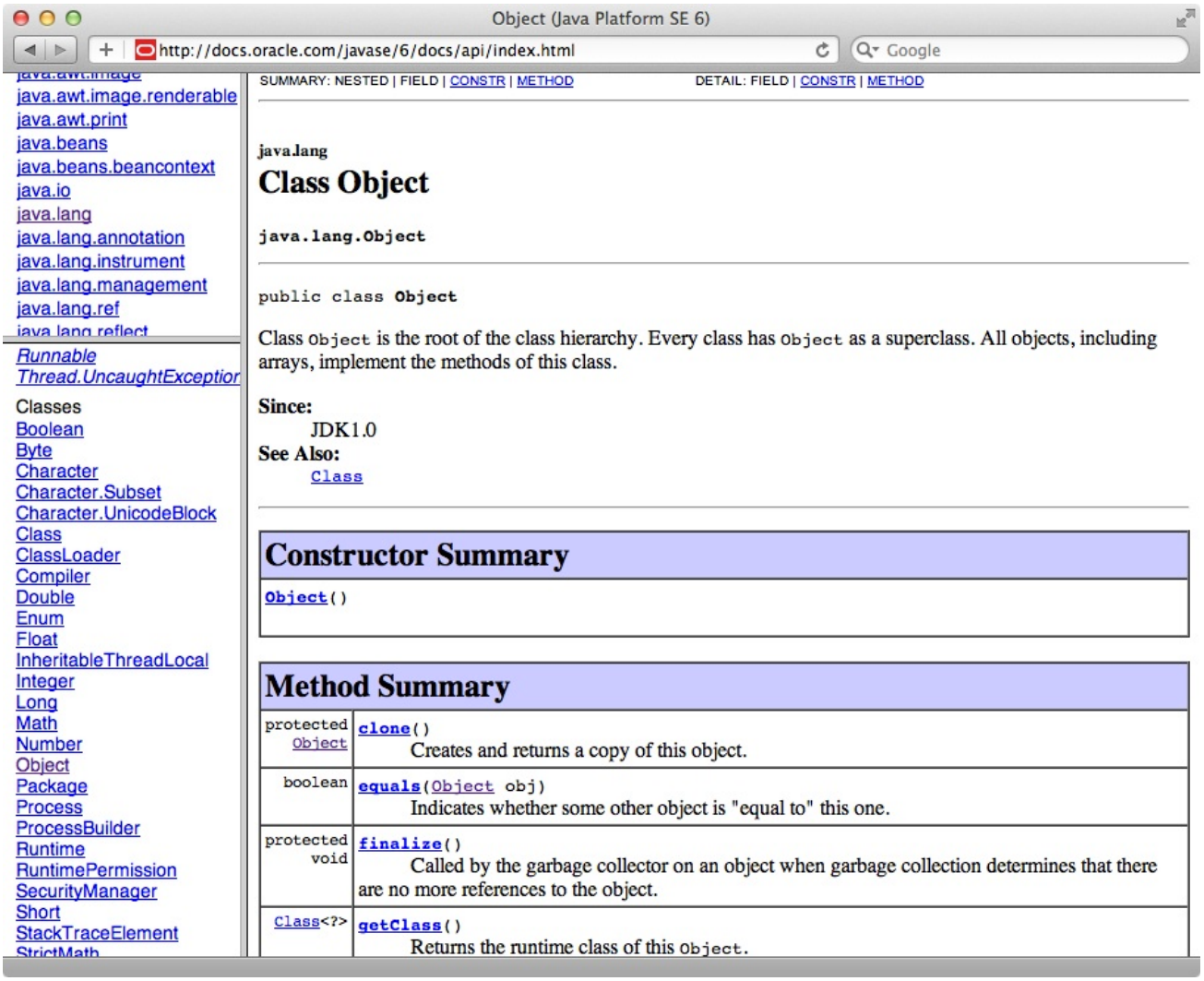
- Java fournit un ensemble très riche de ***classe*** et d'***interfaces*** organisés en ***packages*** et couvrant la plus part des domaines d'applications.
    - http://docs.oracle.com/javase/7/docs/api/index.html
- Ces classes et interfaces constituent les ***API*** Java, dont les champs et méthodes sont décrits en détails dans la documentation.
- La programmation Java consiste principalement à utiliser cette bibliothèque de classes et d'interfaces.
    - Etendre les classes par héritage
    - Implémenter les interfaces

65. Les APIs Java



66. Les APIs Java

Ancien style. Jusqu'à Java 6.



67. Exemples De Packages (`java.lang`)

- Package **`java.lang`**
- Fournit les classes qui sont fondamentales à la conception du langage de programmation Java.
- Les classes les plus importantes sont `Object`, qui est la racine de la hiérarchie des classes, et la classe `Class`, dont les instances représentent les classes à l'exécution.
- On trouve aussi dans ce package les classes *Wrapper*, `Boolean`, `Character`, `Integer`, `Long`, `Float` et `Double`.
- On trouve ausi la classe `Math`, qui fournit les fonctions mathématiques usuelles.
- On y trouve aussi les classes de la famille `String`, la classe `System`, `Throwable`, `Thread` etc.
- On y trouve aussi les interfaces: `Cloneable`, `Runnable` entre autres.
- Importer une classe se fait par `import java.lang.Math`
- Importer toutes les classes `import java.lang.*`
- On peut utiliser les classes et les interfaces de ce package sans faire `import`.

68. Package **java.util**

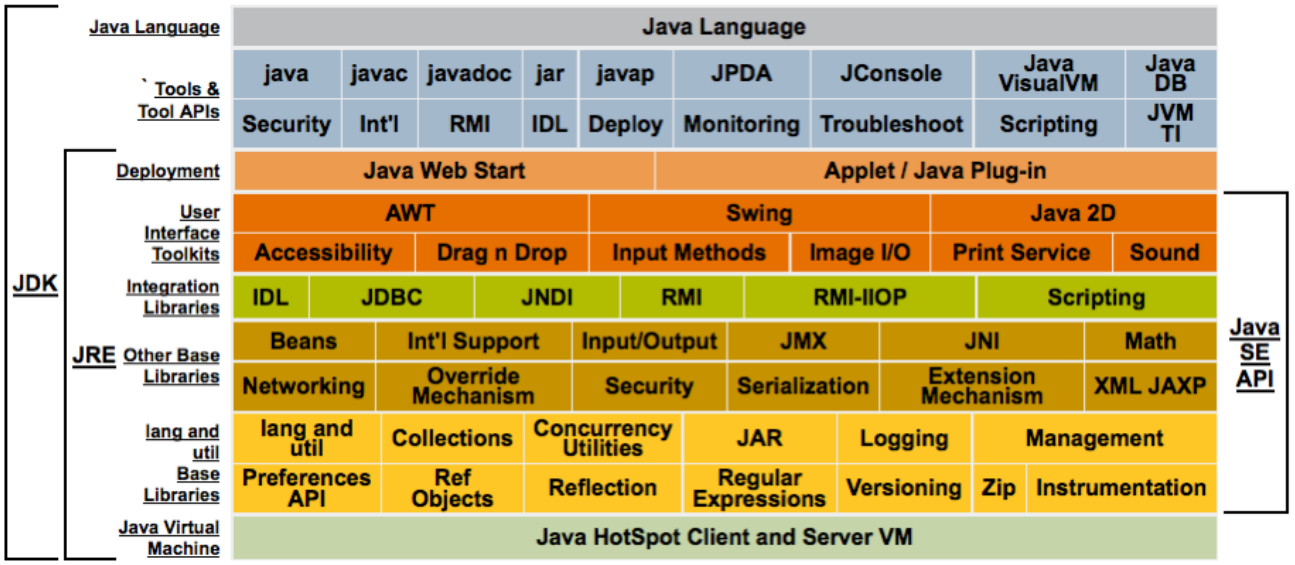
- Package **java.util**
- Contient le cadre des classes génériques *Collections*, (*Set*, *List*, *Map*, *Iterator*...)
- donnant les classes structure de données: `List`, `Vector`, `Stack`, `HashMap`, `HashTable`, etc.
- et d'autres classes utilitaires:
  - générateur de nombres aléatoires,
  - classes `Calendar`, `Date`, `Scanner` ...

69. Package **java.io**

- Package **java.io**
- Tout ce qui concerne les entrées/sorties par flux de données et le système de fichiers.
- `Console`, `Input(Output)Stream`, `FileInput(Output)Stream`, `PrintStream`, `Reader`, `Writer` ...
- `EOFException`, `IOException` ...

70. Plateforme Java

- Actuellement (2012) **Java Platform Standard Edition 7**
- Java SE 7 se compose de deux produits: le Kit de développement (JDK) 7 et l'environnement d'exécution (JRE) 7.
- JRE 7 fournit les bibliothèques, la machine virtuelle Java (JVM), et d'autres composants pour exécuter des applets et des applications Java.
- JDK est un sur-ensemble de JRE, et contient en plus de JRE les autres outils comme le compilateur le débogueur ...
- Description of Java Conceptual Diagram



- http://docs.oracle.com/javase/7/docs/index.html

71. That's All Folks...

Voir quand même:

http://www.java-tips.org/java-se-tips/java.lang/

qui contient quelques bon exemples de programmes et astuces Java

ou en general http://www.java-tips.org/java-se-tips/ et http://www.java-tips.org/