Programmation objet en Java

Vincent Vidal

Maître de Conférences
Enseignements : IUT Lyon 1 - pôle AP - Licence ESSIR - bureau 2ème étage
Recherche : Laboratoire LIRIS - bât. Nautibus
E-mail : vincent.vidal@univ-lyon1.fr

Supports de cours et TPs : `http://spiralconnect.univ-lyon1.fr` module "ASPE - Bases de la POO - Java P1 et P2"
48H prévues ≈ 43H de cours+TDs/TPs, 1H - interros, et 2 DS de promos de 2H

Évaluation : 50% Contrôle continu + 50% DS promo + Bonus/Malus TP

Plan

Les tableaux sont des objects en Java

- `int t[];` (ou `int [] t;`) déclare une référence sur un tableau d'entier.
  En aucun cas cette déclaration n'alloue de la mémoire pour stocker les éléments du tableau.
- `new int[5];` alloue l'emplacement mémoire nécessaire pour stocker un tableau de 5 int et retourne une référence sur ce tableau.
  Toutes les cases du tableau alloué (dans le tas) sont initialisées à zéro (ou à null pour des tableaux de références sur des objets).
- `int t[] = new int[5];` ou `int t[]; t = new int[5];`

Les tableaux sont des objects en Java

- `int t[];` (ou `int [] t;`) déclare une référence sur un tableau d'entier.
  En aucun cas cette déclaration n'alloue de la mémoire pour stocker les éléments du tableau.
- `new int[5];` alloue l'emplacement mémoire nécessaire pour stocker un tableau de 5 int et retourne une référence sur ce tableau.
  Toutes les cases du tableau alloué (dans le tas) sont initialisées à zéro (ou à null pour des tableaux de références sur des objets).
- `int t[] = new int[5];` ou `int t[]; t = new int[5];`

Les tableaux sont des objects en Java

- `int t[];` (ou `int [] t;`) déclare une référence sur un tableau d'entier.
  En aucun cas cette déclaration n'alloue de la mémoire pour stocker les éléments du tableau.
- `new int[5];` alloue l'emplacement mémoire nécessaire pour stocker un tableau de 5 int et retourne une référence sur ce tableau.
  Toutes les cases du tableau alloué (dans le tas) sont initialisées à zéro (ou à null pour des tableaux de références sur des objets).
- `int t[] = new int[5];` ou `int t[]; t = new int[5];`

Les tableaux sont des objects en Java

- `int t[];` (ou `int [] t;`) déclare une référence sur un tableau d'entier.
  En aucun cas cette déclaration n'alloue de la mémoire pour stocker les éléments du tableau.
- `new int[5];` alloue l'emplacement mémoire nécessaire pour stocker un tableau de 5 int et retourne une référence sur ce tableau.
  Toutes les cases du tableau alloué (dans le tas) sont initialisées à zéro (ou à null pour des tableaux de références sur des objets).
- `int t[] = new int[5];` ou `int t[]; t = new int[5];`

Les tableaux sont des objects en Java

- `int t[];` (ou `int [] t;`) déclare une référence sur un tableau d'entier.
  En aucun cas cette déclaration n'alloue de la mémoire pour stocker les éléments du tableau.
- `new int[5];` alloue l'emplacement mémoire nécessaire pour stocker un tableau de 5 int et retourne une référence sur ce tableau.
  Toutes les cases du tableau alloué (dans le tas) sont initialisées à zéro (ou à null pour des tableaux de références sur des objets).
- `int t[] = new int[5];` ou `int t[]; t = new int[5];`

Les tableaux sont des objects en Java

- `int t[];` (ou `int [] t;`) déclare une référence sur un tableau d'entier.
  En aucun cas cette déclaration n'alloue de la mémoire pour stocker les éléments du tableau.
- `new int[5];` alloue l'emplacement mémoire nécessaire pour stocker un tableau de 5 int et retourne une référence sur ce tableau.
  Toutes les cases du tableau alloué (dans le tas) sont initialisées à zéro (ou à null pour des tableaux de références sur des objets).
- `int t[] = new int[5];` ou `int t[]; t = new int[5];`

Remarques su l'opérateur new

- `new int[n];` fonctionne uniquement pour $n \geq 0$.
  Un $n < 0$ entraînera une *exception *NegativeArraySizeException*.
- Dans `new int[n];` n peut être une variable saisie au clavier.

(*) La gestion des exceptions sera abordée dans un prochain cours.

Remarques su l'opérateur new

- `new int[n];` fonctionne uniquement pour $n \geq 0$.
  Un $n < 0$ entraînera une *exception *NegativeArraySizeException*.
- Dans `new int[n];` n peut être une variable saisie au clavier.

(*) La gestion des exceptions sera abordée dans un prochain cours.

Remarques su l'opérateur new

- `new int[n];` fonctionne uniquement pour $n \geq 0$.
  Un $n < 0$ entraînera une *exception *NegativeArraySizeException*.
- Dans `new int[n];` n peut être une variable saisie au clavier.

(*) La gestion des exceptions sera abordée dans un prochain cours.

Remarques su l'opérateur new

- `new int[n];` fonctionne uniquement pour $n \geq 0$.
  Un $n < 0$ entraînera une *exception *NegativeArraySizeException*.
- Dans `new int[n];` n peut être une variable saisie au clavier.

(*) La gestion des exceptions sera abordée dans un prochain cours.

Les initialiseurs de tableaux

- `int t[] = {1, 2, 3, 4, 5};` alloue un tableau de 5 cases initialisées avec les valeurs données et stocke la référence de ce tableau dans t.
- `int t[];`
  `t = {1, 2, 3, 4, 5};` génère une erreur de compilation.
- `int n = 3;`
  `int t[] = {n, -n, 2*n};` est autorisé (pas le cas en C).

Les initialiseurs de tableaux

- `int t[] = {1, 2, 3, 4, 5};` alloue un tableau de 5 cases initialisées avec les valeurs données et stocke la référence de ce tableau dans t.
- `int t[];`
  `t = {1, 2, 3, 4, 5};` génère une erreur de compilation.
- `int n = 3;`
  `int t[] = {n, -n, 2*n};` est autorisé (pas le cas en C).

Les initialiseurs de tableaux

- `int t[] = {1, 2, 3, 4, 5};` alloue un tableau de 5 cases initialisées avec les valeurs données et stocke la référence de ce tableau dans t.
- `int t[];`
  `t = {1, 2, 3, 4, 5};` génère une erreur de compilation.
- `int n = 3;`
  `int t[] = {n, -n, 2*n};` est autorisé (pas le cas en C).

Utilisation

`int t[] = new int[5];`

- `t[0] = 7; t[4] = -8;`
- `t[0]++; --t[1];`
- `System.out.println(t[3]);`
- `t[n];` avec $n < 0$ ou $n > 4$ entraînera une exception *ArrayIndexOutOfBoundsException*.

Utilisation

int t[] = new int[5] ;

- t[0] = 7 ; t[4] = -8 ;
- t[0]++ ; −−t[1] ;
- System.out.println(t[3]) ;
- t[n] ; avec $n < 0$ ou $n > 4$ entraînera une exception *ArrayIndexOutOfBoundsException*.

Utilisation

int t[] = new int[5] ;

- t[0] = 7 ; t[4] = -8 ;
- t[0]++ ; −−t[1] ;
- System.out.println(t[3]) ;
- t[n] ; avec $n < 0$ ou $n > 4$ entraînera une exception *ArrayIndexOutOfBoundsException*.

Utilisation

int t[] = new int[5] ;

- t[0] = 7 ; t[4] = -8 ;
- t[0]++ ; −−t[1] ;
- System.out.println(t[3]) ;
- t[n] ; avec $n < 0$ ou $n > 4$ entraînera une exception *ArrayIndexOutOfBoundsException*.

Affectation de tableaux

```
int t1[] = new int[5];
int t2[] = new int[2];
```

- `t2 = t1;` va copier la référence contenue dans t1 dans t2. t1 et t2 désignent alors le même tableau d'entiers.
  Si le tableau anciennement référencé par t2 n'est pas référencé par une autre référence, il deviendra candidat au ramasse-miettes.
  C'est le même comportement observé que pour l'affectation o2=o1 où o1 et o2 sont deux références sur un objet de même type.
- `t2 == t1` testera l'égalité des références...

Affectation de tableaux

```
int t1[] = new int[5];
int t2[] = new int[2];
```

- `t2 = t1;` va copier la référence contenue dans t1 dans t2. t1 et t2 désignent alors le même tableau d'entiers.
  Si le tableau anciennement référencé par t2 n'est pas référencé par une autre référence, il deviendra candidat au ramasse-miettes.
  C'est le même comportement observé que pour l'affectation o2=o1 où o1 et o2 sont deux références sur un objet de même type.
- `t2 == t1` testera l'égalité des références...

Affectation de tableaux

```
int t1[] = new int[5];
int t2[] = new int[2];
```

- `t2 = t1;` va copier la référence contenue dans t1 dans t2. t1 et t2 désignent alors le même tableau d'entiers.
  Si le tableau anciennement référencé par t2 n'est pas référencé par une autre référence, il deviendra candidat au ramasse-miettes.
  C'est le même comportement observé que pour l'affectation o2=o1 où o1 et o2 sont deux références sur un objet de même type.
- `t2 == t1` testera l'égalité des références...

Affectation de tableaux

```
int t1[] = new int[5];
int t2[] = new int[2];
```

- `t2 = t1;` va copier la référence contenue dans t1 dans t2. t1 et t2 désignent alors le même tableau d'entiers.
  Si le tableau anciennement référencé par t2 n'est pas référencé par une autre référence, il deviendra candidat au ramasse-miettes.
  C'est le même comportement observé que pour l'affectation o2=o1 où o1 et o2 sont deux références sur un objet de même type.
- `t2 == t1` testera l'égalité des références...

Affectation de tableaux

```
int t1[] = new int[5];
int t2[] = new int[2];
```

- t2 = t1 ; va copier la référence contenue dans t1 dans t2. t1 et t2 désignent alors le même tableau d'entiers.
  Si le tableau anciennement référencé par t2 n'est pas référencé par une autre référence, il deviendra candidat au ramasse-miettes.
  C'est le même comportement observé que pour l'affectation o2=o1 où o1 et o2 sont deux références sur un objet de même type.
- t2 == t1 testera l'égalité des références...

Les tableaux sont des objets en Java

- On a un accès direct à la taille du tableau : t.length

on peut donc parcourir le tableau via :
`int i ; for( i=0 ; i<t.length ; i++) ...`

ou

`for( int i=0 ; i<t.length ; i++) ...`

Quelle est la différence entre ces 2 boucles ?

Les tableaux sont des objets en Java

- On a un accès direct à la taille du tableau : t.length

  on peut donc parcourir le tableau via :
  int i ; for(i=0 ; i<t.length ; i++) ...
  ou
  for(**int** i=0 ; i<t.length ; i++) ...

  Quelle est la différence entre ces 2 boucles ?

Les tableaux sont des objets en Java

- On a un accès direct à la taille du tableau : t.length

  on peut donc parcourir le tableau via :
  int i ; for(i=0 ; i<t.length ; i++) ...
  ou
  for(**int** i=0 ; i<t.length ; i++) ...

  Quelle est la différence entre ces 2 boucles ?

Remarques

- `int t1[] = new int[5];` déclare un tableau de **dimension fixe jusqu'à la libération effective** du tableau.
  Les tableaux de taille dynamique sont implantés avec la classe `ArrayList` *du paquetage* `java.util`.
- `int t1[] = new int[5]; float t2[];` alors `t2 = t1` est interdit !
- Pour passer un tableau alloué dans le tas en argument d'une fonction, il n'est plus nécessaire de passer sa taille (c'était le cas en C).
- Une fonction qui retourne une référence vers un tableau (e.g. `int[]`), nous permet de récupérer sa taille !

Remarques

- `int t1[] = new int[5];` déclare un tableau de **dimension fixe jusqu'à la libération effective** du tableau.
  Les tableaux de taille dynamique sont implantés avec la classe `ArrayList` *du paquetage* `java.util`.
- `int t1[] = new int[5]; float t2[];` alors `t2 = t1` est interdit !
- Pour passer un tableau alloué dans le tas en argument d'une fonction, il n'est plus nécessaire de passer sa taille (c'était le cas en C).
- Une fonction qui retourne une référence vers un tableau (e.g. `int[]`), nous permet de récupérer sa taille !

Remarques

- `int t1[] = new int[5];` déclare un tableau de **dimension fixe jusqu'à la libération effective** du tableau.
  Les tableaux de taille dynamique sont implantés avec la classe `ArrayList` *du paquetage* `java.util`.
- `int t1[] = new int[5]; float t2[];` alors `t2 = t1` est interdit !
- Pour passer un tableau alloué dans le tas en argument d'une fonction, il n'est plus nécessaire de passer sa taille (c'était le cas en C).
- Une fonction qui retourne une référence vers un tableau (e.g. `int[]`), nous permet de récupérer sa taille !

Remarques

- `int t1[] = new int[5];` déclare un tableau de **dimension fixe jusqu'à la libération effective** du tableau.
  Les tableaux de taille dynamique sont implantés avec la classe `ArrayList` *du paquetage* `java.util`.
- `int t1[] = new int[5]; float t2[];` alors `t2 = t1` est interdit !
- Pour passer un tableau alloué dans le tas en argument d'une fonction, il n'est plus nécessaire de passer sa taille (c'était le cas en C).
- Une fonction qui retourne une référence vers un tableau (e.g. `int[]`), nous permet de récupérer sa taille !

Remarques

- `int t1[] = new int[5];` déclare un tableau de **dimension fixe jusqu'à la libération effective** du tableau.
  Les tableaux de taille dynamique sont implantés avec la classe `ArrayList` *du paquetage* `java.util`.
- `int t1[] = new int[5]; float t2[];` alors `t2 = t1` est interdit !
- Pour passer un tableau alloué dans le tas en argument d'une fonction, il n'est plus nécessaire de passer sa taille (c'était le cas en C).
- Une fonction qui retourne une référence vers un tableau (e.g. `int[]`), nous permet de récupérer sa taille !

Les tableaux de caractères

```
char [] tc1 = {'h', 'e', 'l', 'l', 'o'};
char [] tc2 = new char[26];
for(int i=0; i<tc2.length; i++) tc2[i] = (char)('A' + i);
```

- System.out.println(tc*i*); affichera bien tous les caractères du tableau car la méthode println est redéfinie pour les tableaux de caractères (mais pas pour les autres types de tableau).
- System.out.println("tc*i*=" + tc*i*); n'affichera pas le résultat attendu...

Les tableaux de caractères

```
char [] tc1 = {'h', 'e', 'l', 'l', 'o'};
char [] tc2 = new char[26];
for(int i=0; i<tc2.length; i++) tc2[i] = (char)('A' + i);
```

- System.out.println(tc*i*); affichera bien tous les caractères du tableau car la méthode println est redéfinie pour les tableaux de caractères (mais pas pour les autres types de tableau).
- System.out.println("tc*i*=" + tc*i*); n'affichera pas le résultat attendu...

Les tableaux d'objets

On réserve la place pour stocker *12* références d'objet de type Type :

```
Type [] tc = new Type[12];
```

Ensuite on doit allouer un objet pour chaque case :

```
for(int i=0; i<tc.length; i++) tc[i] = new Type(...);
```

A partir de là, on manipule une case du tableau comme un objet Java usuel :

```
tc[1].affiche();
```

Les tableaux d'objets

On réserve la place pour stocker *12* références d'objet de type Type :
Type [] tc = new Type[*12*] ;

Ensuite on doit allouer un objet pour chaque case :
for(**int** i=0 ; i<tc.length ; i++) tc[i] = new Type(...) ;

A partir de là, on manipule une case du tableau comme un objet Java usuel :
tc[1].affiche() ;

Les tableaux d'objets

On réserve la place pour stocker *12* références d'objet de type Type :
`Type [] tc = new Type[12];`

Ensuite on doit allouer un objet pour chaque case :
`for(int i=0; i<tc.length; i++) tc[i] = new Type(...);`

A partir de là, on manipule une case du tableau comme un objet Java usuel :
`tc[1].affiche();`

Java ne dispose pas de "vrais" tableaux nD...

- On peut "simuler" les tableaux 2D avec des tableaux 1D de références sur des tableaux. Par contre la taille de chaque "sous-tableau" ne sera pas obligatoirement la même...
  Et les lignes d'un tableau nD ne seront pas contiguës en mémoire.

Java ne dispose pas de "vrais" tableaux nD...

- On peut "simuler" les tableaux 2D avec des tableaux 1D de références sur des tableaux. Par contre la taille de chaque "sous-tableau" ne sera pas obligatoirement la même...
  Et les lignes d'un tableau nD ne seront pas contiguës en mémoire.

Java ne dispose pas de "vrais" tableaux nD...

- On peut "simuler" les tableaux 2D avec des tableaux 1D de références sur des tableaux. Par contre la taille de chaque "sous-tableau" ne sera pas obligatoirement la même...
  Et les lignes d'un tableau nD ne seront pas contiguës en mémoire.

Déclaration d'un tableau 2D

Les 3 déclarations suivantes sont équivalentes :

```
int t [] [] ;
int [] t [] ;
int [] [] t ;
```

Initialiseurs d'un tableau 2D

```
int t1 [] [] = { new int [4], new int[3] } ;
int t2 [] [] = { {1,3,5}, {0,2,4,6,8,10}, {1,2,3,5,7} } ;
```

- t1.length vaut 2.
  t1[0].length vaut 4.
  t1[1].length vaut 3.
- t2.length vaut 3.
  t2[0].length vaut 3.
  t2[1].length vaut 6...

Initialiseurs d'un tableau 2D

```
int t1 [] [] = { new int [4], new int[3] };
int t2 [] [] = { {1,3,5}, {0,2,4,6,8,10}, {1,2,3,5,7} };
```

- t1.length vaut 2.
  t1[0].length vaut 4.
  t1[1].length vaut 3.
- t2.length vaut 3.
  t2[0].length vaut 3.
  t2[1].length vaut 6...

Initialiseurs d'un tableau 2D

```
int t1 [] [] = { new int [4], new int[3] } ;
int t2 [] [] = { {1,3,5}, {0,2,4,6,8,10}, {1,2,3,5,7} } ;
```

- t1.length vaut 2.
  t1[0].length vaut 4.
  t1[1].length vaut 3.
- t2.length vaut 3.
  t2[0].length vaut 3.
  t2[1].length vaut 6...

Initialiseurs d'un tableau 2D

```
int t1 [] [] = { new int [4], new int[3] };
int t2 [] [] = { {1,3,5}, {0,2,4,6,8,10}, {1,2,3,5,7} };
```

- t1.length vaut 2.
  t1[0].length vaut 4.
  t1[1].length vaut 3.
- t2.length vaut 3.
  t2[0].length vaut 3.
  t2[1].length vaut 6...

Initialiseurs d'un tableau 2D

```
int t1 [] [] = { new int [4], new int[3] };
int t2 [] [] = { {1,3,5}, {0,2,4,6,8,10}, {1,2,3,5,7} };
```

- t1.length vaut 2.
  t1[0].length vaut 4.
  t1[1].length vaut 3.
- t2.length vaut 3.
  t2[0].length vaut 3.
  t2[1].length vaut 6...

Initialiseurs d'un tableau 2D

```
int t1 [] [] = { new int [4], new int[3] } ;
int t2 [] [] = { {1,3,5}, {0,2,4,6,8,10}, {1,2,3,5,7} } ;
```

- t1.length vaut 2.
  t1[0].length vaut 4.
  t1[1].length vaut 3.
- t2.length vaut 3.
  t2[0].length vaut 3.
  t2[1].length vaut 6...

Sans les initialiseurs...

```
int t [] [] ;
t = new int [2][] ; // tableau de 2 tableaux de int
t[0] = new int [3] ;
t[1] = new int [4] ;

int t2[][] = new int [][2] ; // engendre une erreur de compilation

int t3[][] = new int [5][2] ;// Facilité de Java (matrice carrée)
```

Exemple de parcours d'un tableau 2D

```
static void affiche(int t[][])
{
   int col ;
   for(int lig=0; lig<t.length; lig++)
   {
      for(col=0; col<t[lig].length; col++)
         System.out.print(t[lig][col]+ " ") ;

      System.out.println() ;
   }
}
```

Exemple de parcours d'un tableau 2D en JDK 5.0

```
static void afficheJDK5(int t[][])
{
   for(int [] ligne : t)
   {
      for(int valCase : ligne)
         System.out.print(valCase+ " ") ;

      System.out.println() ;
   }
}
```

Ce parcours JDK 5.0 n'est valide que pour un accès aux valeurs des cases, pas pour les modifier.

Depuis JDK 5.0 le dernier argument d'une méthode peut être variable en nombre

```
static int somme(int ... valeurs)
{
   int s = 0 ;
   for(int argi : valeurs) // le dernier argument est
   {                       // interprété comme un
      s += argi ;          // tableau 1D
   }
   return s ;
}
// exemples d'appel valide:
somme() ;
somme(5, 7, 9) ;
somme(tab1D) ; // tab1D étant de type int []
```

Ellipse et surdéfinition de méthodes

Règle : on cherche en premier une méthode sans ellipse qui correspond, en utilisant les conversions non-dégradantes si nécessaire. Si aucune méthode n'est trouvée, alors on effectue la recherche en faisant intervenir les méthodes à ellipse.

- void f(int ... valeurs) et void f(int [] t) ne peuvent pas coexister.
- void f(int i1, int i2) et void f(int ... valeurs) peuvent coexister.
- void f(double d1, double d2) et void f(int ... valeurs) peuvent coexister.

Ellipse et surdéfinition de méthodes

Règle : on cherche en premier une méthode sans ellipse qui correspond, en utilisant les conversions non-dégradantes si nécessaire. Si aucune méthode n'est trouvée, alors on effectue la recherche en faisant intervenir les méthodes à ellipse.

- `void f(int ... valeurs)` et `void f(int [] t)` ne peuvent pas coexister.
- `void f(int i1, int i2)` et `void f(int ... valeurs)` peuvent coexister.
- `void f(double d1, double d2)` et `void f(int ... valeurs)` peuvent coexister.

Ellipse et surdéfinition de méthodes

Règle : on cherche en premier une méthode sans ellipse qui correspond, en utilisant les conversions non-dégradantes si nécessaire. Si aucune méthode n'est trouvée, alors on effectue la recherche en faisant intervenir les méthodes à ellipse.

- void f(int ... valeurs) et void f(int [] t) ne peuvent pas coexister.
- void f(int i1, int i2) et void f(int ... valeurs) peuvent coexister.
- void f(double d1, double d2) et void f(int ... valeurs) peuvent coexister.

Ellipse et surdéfinition de méthodes

Règle : on cherche en premier une méthode sans ellipse qui correspond, en utilisant les conversions non-dégradantes si nécessaire. Si aucune méthode n'est trouvée, alors on effectue la recherche en faisant intervenir les méthodes à ellipse.

- `void f(int ... valeurs)` et `void f(int [] t)` ne peuvent pas coexister.
- `void f(int i1, int i2)` et `void f(int ... valeurs)` peuvent coexister.
- `void f(double d1, double d2)` et `void f(int ... valeurs)` peuvent coexister.

Java dispose de la classe String pour manipuler les chaînes de caractères

Les chaînes de caractères constantes telles que "bonjour" sont en fait des objets de type String construits automatiquement par le compilateur.

Les objets de type String **ne sont pas modifiables** (ils sont immuables).

Si votre programme nécessite la modification de chaînes de caractères (pour des raisons de performance), alors vous pouvez utiliser la classe **StringBuffer**.

Java dispose de la classe String pour manipuler les chaînes de caractères

Les chaînes de caractères constantes telles que "bonjour" sont en fait des objets de type String construits automatiquement par le compilateur.

Les objets de type String **ne sont pas modifiables** (ils sont immuables).

Si votre programme nécessite la modification de chaînes de caractères (pour des raisons de performance), alors vous pouvez utiliser la classe **StringBuffer**.

Java dispose de la classe String pour manipuler les chaînes de caractères

Les chaînes de caractères constantes telles que "bonjour" sont en fait des objets de type String construits automatiquement par le compilateur.

Les objets de type String **ne sont pas modifiables** (ils sont immuables).

Si votre programme nécessite la modification de chaînes de caractères (pour des raisons de performance), alors vous pouvez utiliser la classe **StringBuffer**.

Les constructeurs classiques

```
String ch ; // je déclare une référence sur
            // un objet de type String
String ch1 = new String() ; // constructeur sans arg
String ch2 = new String("salut") ;
String ch3 = new String(ch2) ; // constructeur de recopie
...
System.out.println(ch2) ;
```

Les chaînes constantes

```
String ch = "Bonjour" ;
String ch2 = "Bonjour\ncomment allez-vous?" ;
String ch3 = "\t message décalé" ;
...
System.out.println(ch2) ;
```

Entre les guillemets, nous pouvons utiliser des caractères usuels, des caractères spéciaux (\n etc.) et des codes unicodes (en hexa ou octal).

Les méthodes de la classe String

```
String ch = new String("salut") ;
ch.length() ;  // la longueur (bien une méthode!)
ch.charAt(0) ; // accès au caractère d'indice 0 ('s')
String chConcat = ch + ch ; // concaténation de 2 chaînes
String chConcat2 = ch + 2 ; // concaténation d'1 chaîne et
                            // d'1 opérande de type
                            // primitif
// chaque concaténation crée une nouvelle chaîne!
```

Remarque : l'opérateur + peut fonctionner entre un objet de type String et un object d'un autre type car elle utilise la méthode toString() existant pour tout objet Java.

Les méthodes de la classe String

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

ch.indexOf('l') ; // recherche de la première occurrence
ch.indexOf("lu") ; // d'un caractère ou d'une sous-chaîne

ch.lastIndexOf('u') ; // idem indexOf mais dernière
ch.lastIndexOf("lu") ; // occurrence
```

Egalité de 2 chaînes : la méthode equals

```
String ch1 = "hello" ;
String ch2 = "salut" ;
...
ch1.equals(ch1) ; // vrai
ch1.equals(ch2) ; // faux

ch1.equals("hello") ; // vrai
ch1.equals("salut") ; // faux

ch2.equals("salut") ; // vrai
ch2.equals("hello") ; // faux
```

equals compare les caractères tandis que == et != comparent uniquement les références.

Egalité de 2 chaînes : la méthode equals

equalsIgnoreCase compare 2 chaînes de caractères sans tenir compte de la casse des caractères.

Comparaison de 2 chaînes : la méthode compareTo

Comparaison basée sur l'ordre lexicographique induit par la valeur des codes unicode des caractères.

ch1.compareTo(ch2) fournit :

- un entier négatif si ch1 arrive avant ch2 ;
- un entier nul si ch1 et ch2 sont égales ;
- un entier positif si ch1 arrive après ch2.

Comparaison de 2 chaînes : la méthode compareTo

Comparaison basée sur l'ordre lexicographique induit par la valeur des codes unicode des caractères.

ch1.compareTo(ch2) fournit :

- un entier négatif si ch1 arrive avant ch2 ;
- un entier nul si ch1 et ch2 sont égales ;
- un entier positif si ch1 arrive après ch2.

Comparaison de 2 chaînes : la méthode compareTo

Comparaison basée sur l'ordre lexicographique induit par la valeur des codes unicode des caractères.

ch1.compareTo(ch2) fournit :

- un entier négatif si ch1 arrive avant ch2 ;
- un entier nul si ch1 et ch2 sont égales ;
- un entier positif si ch1 arrive après ch2.

Depuis le JDK 7.0 : l'instruction switch sur des String

```
String mois = lireString() ;
switch(mois)
{
   case "janvier" : ...
   case "fevrier" : ...
   ...
}
```

La comparaison des étiquettes est réalisée en interne avec la méthode equals.

Méthodes de modification de String

```
String ch = "SAlut";
String ch2 = ch.replace('t', 'e') ; // "SAlue"
String ch3 = ch.substring(2) ;      // "lut"
String ch4 = ch.substring(1, 3) ;   // "Al" [Deb, Fin[
String ch5 = ch.toLowerCase() ;     // "salut"
String ch6 = ch.toUpperCase() ;     // "SALUT"
```

Ces méthodes créent une nouvelle instance (car les objets de type String ne sont pas modifiables).

Conversions entre chaînes et tableaux de caractères

```
// d'un tableau de char vers une chaîne
char [] tabc = {'s', 'a', 'l', 'u', 't' } ;
String chmot = new String(tabc) ;
String chmot2 = new String(tabc, 2, 3) ; // index 1er car
                                         // et longueur

// d'une chaîne vers un tableau de char
String mot = "salut" ;
char [] tabc2 = mot.toCharArray() ;
```

Conversions entre chaînes et chaînes modifiables

```
// d'une chaîne vers une chaîne modifiable
String ch = "salut" ;
StringBuffer chBuff = new StringBuffer(ch) ;

// d'une chaîne modifiable vers une chaîne
StringBuffer chBuff2 = new StringBuffer() ;
chBuff2.append ("sallut") ;
chBuff2.deleteCharAt (2) ;
String ch2 = chBuff2.toString() ;
```

Depuis le JDK 5.0, il existe une classe StringBuilder semblable à StringBuffer mais plus adaptée à la programmation NON concurrente (ses méthodes ne sont pas synchronisées).

D'un type primitif vers une chaîne

```
int n = 47 ;
String chi = String.valueOf(n) ;

double x = 54.3 ;
String chd = String.valueOf(x) ;
```

La méthode statique valueOf **de la classe String est surdéfinie pour chaque type primitif.**

Remarque : 2 autres solutions : "" + n ou Integer.toString(n)

D'un type primitif vers une chaîne

```
int n = 47 ;
String chi = String.valueOf(n) ;

double x = 54.3 ;
String chd = String.valueOf(x) ;
```

La méthode statique valueOf **de la classe String est surdéfinie pour chaque type primitif.**

Remarque : 2 autres solutions : "" + n ou Integer.toString(n)

D'une chaîne vers un type primitif

Il faudra utiliser une méthode statique parseXXX de la classe enveloppe associée au type primitif (e.g. Integer pour int).

```
String ch = "1234";
int n = Integer.parseInt(ch) ;

String ch = "42.7";
double x = Double.parseDouble(ch) ;
```

Ce type de conversion impose des contraintes de formatage de la chaîne : par exemple une chaîne contenant un entier ne doit pas commencer par '+'. En cas d'échec de conversion une exception *NumberFormatException* est lancée.

Définition

Un *type énuméré* est un type représenté par un ensemble fini de constantes dont on choisit explicitement les identificateurs des constantes.

Depuis JDK 5.0, Java possède des types énumérés.

```
enum Jour { lundi, mardi, mercredi, jeudi, vendredi,
    samedi, dimanche } // en dehors d'une méthode
                       // ou d'une classe
...
Jour courant = Jour.mercredi ; // dans une méthode
```

Jour est une classe. lundi, mardi etc. sont des instances finales (non-modifiables) de cette classe. Toutes les classes d'énumération dérivent de la classe Enum.

Définition

Un *type énuméré* est un type représenté par un ensemble fini de constantes dont on choisit explicitement les identificateurs des constantes.

Depuis JDK 5.0, Java possède des types énumérés.

```
enum Jour { lundi, mardi, mercredi, jeudi, vendredi,
    samedi, dimanche } // en dehors d'une méthode
                       // ou d'une classe
...
Jour courant = Jour.mercredi ; // dans une méthode
```

Jour est une classe. lundi, mardi etc. sont des instances finales (non-modifiables) de cette classe. Toutes les classes d'énumération dérivent de la classe Enum.

Définition

Un *type énuméré* est un type représenté par un ensemble fini de constantes dont on choisit explicitement les identificateurs des constantes.

Depuis JDK 5.0, Java possède des types énumérés.

```
enum Jour { lundi, mardi, mercredi, jeudi, vendredi,
    samedi, dimanche } // en dehors d'une méthode
                       // ou d'une classe
...
Jour courant = Jour.mercredi ; // dans une méthode
```

Jour est une classe. lundi, mardi etc. sont des instances finales (non-modifiables) de cette classe. Toutes les classes d'énumération dérivent de la classe Enum.

Définition

Un *type énuméré* est un type représenté par un ensemble fini de constantes dont on choisit explicitement les identificateurs des constantes.

Depuis JDK 5.0, Java possède des types énumérés.

```
enum Jour { lundi, mardi, mercredi, jeudi, vendredi,
    samedi, dimanche } // en dehors d'une méthode
                       // ou d'une classe
...
Jour courant = Jour.mercredi ; // dans une méthode
```

Jour est une classe. lundi, mardi etc. sont des instances finales (non-modifiables) de cette classe. Toutes les classes d'énumération dérivent de la classe Enum.

Relation d'ordre

Un *type énuméré* Java est muni d'une relation d'ordre via une valeur entière (le rang de la valeur au moment de la déclaration).

- Jour.lundi.ordinal() vaut 0, Jour.mardi.ordinal() vaut 1, ..., Jour.dimanche.ordinal() vaut 6 ; l'ordre induit est donc Jour.lundi < Jour.mardi < ... < Jour.dimanche
- courant.compareTo(Jour.lundi) sera positif, courant.compareTo(Jour.mercredi) sera nul et courant.compareTo(Jour.vendredi) sera négatif
- courant.equals(Jour.mercredi) sera vrai et une comparaison avec tout autre jour sera fausse.
  Ici courant == Jour.mercredi ou courant != Jour.mercredi ont du sens car il n'existe qu'une seule instance en mémoire de Jour.XXX.

Relation d'ordre

Un *type énuméré* Java est muni d'une relation d'ordre via une valeur entière (le rang de la valeur au moment de la déclaration).

- Jour.lundi.ordinal() vaut 0, Jour.mardi.ordinal() vaut 1, ..., Jour.dimanche.ordinal() vaut 6 ; l'ordre induit est donc Jour.lundi < Jour.mardi < ... < Jour.dimanche
- courant.compareTo(Jour.lundi) sera positif,
  courant.compareTo(Jour.mercredi) sera nul
  et courant.compareTo(Jour.vendredi) sera négatif
- courant.equals(Jour.mercredi) sera vrai et une comparaison avec tout autre jour sera fausse.
  Ici courant == Jour.mercredi ou courant != Jour.mercredi ont du sens car il n'existe qu'une seule instance en mémoire de Jour.XXX.

Relation d'ordre

Un *type énuméré* Java est muni d'une relation d'ordre via une valeur entière (le rang de la valeur au moment de la déclaration).

- Jour.lundi.ordinal() vaut 0, Jour.mardi.ordinal() vaut 1, ..., Jour.dimanche.ordinal() vaut 6 ; l'ordre induit est donc Jour.lundi < Jour.mardi < ... < Jour.dimanche
- courant.compareTo(Jour.lundi) sera positif, courant.compareTo(Jour.mercredi) sera nul et courant.compareTo(Jour.vendredi) sera négatif
- courant.equals(Jour.mercredi) sera vrai et une comparaison avec tout autre jour sera fausse.
  Ici courant == Jour.mercredi ou courant != Jour.mercredi ont du sens car il n'existe qu'une seule instance en mémoire de Jour.XXX.

Relation d'ordre

Un *type énuméré* Java est muni d'une relation d'ordre via une valeur entière (le rang de la valeur au moment de la déclaration).

- Jour.lundi.ordinal() vaut 0, Jour.mardi.ordinal() vaut 1, ..., Jour.dimanche.ordinal() vaut 6 ; l'ordre induit est donc Jour.lundi < Jour.mardi < ... < Jour.dimanche
- courant.compareTo(Jour.lundi) sera positif, courant.compareTo(Jour.mercredi) sera nul et courant.compareTo(Jour.vendredi) sera négatif
- courant.equals(Jour.mercredi) sera vrai et une comparaison avec tout autre jour sera fausse.

Ici courant == Jour.mercredi ou courant != Jour.mercredi ont du sens car il n'existe qu'une seule instance en mémoire de Jour.XXX.

Relation d'ordre

Un *type énuméré* Java est muni d'une relation d'ordre via une valeur entière (le rang de la valeur au moment de la déclaration).

- Jour.lundi.ordinal() vaut 0, Jour.mardi.ordinal() vaut 1, ..., Jour.dimanche.ordinal() vaut 6 ; l'ordre induit est donc Jour.lundi < Jour.mardi < ... < Jour.dimanche
- courant.compareTo(Jour.lundi) sera positif, courant.compareTo(Jour.mercredi) sera nul et courant.compareTo(Jour.vendredi) sera négatif
- courant.equals(Jour.mercredi) sera vrai et une comparaison avec tout autre jour sera fausse.
  Ici courant == Jour.mercredi ou courant != Jour.mercredi ont du sens car il n'existe qu'une seule instance en mémoire de Jour.XXX.

L'instruction switch sur des types énumérés

```
...
Jour courant = Jour.mercredi ; // dans une méthode...
switch(courant)
{
   // Il est nécessaire d'utiliser les noms des constantes
   // sans les préfixer du nom de leur classe
   case samedi :
   case dimanche :
            System.out.println("C'est le week-end!") ;
            break ;
   default : System.out.println("C'est un jour de
    travail!") ;
}
```

Conversions entre chaînes et types énumérés

```
// d'un type énuméré vers une chaîne
String ch = Jour.mercredi.toString() ;

// d'une chaîne vers un type énuméré
Jour courant = Jour.valueOf("lundi") ;
```

Itération sur les valeurs d'un type énuméré

```
for( Jour j : Jour.values() ) // Obligatoirement style
{                             // JDK 5.0
   ...
}
```

Plan

Décrire votre code dans votre code lui-même plutôt que dans un document séparé vous aide à garder votre documentation à jour.

Page de référence pour Javadoc :
http://www.oracle.com/technetwork/java/javase/documentation/index-jsp-135444.html#javadocdocuments

Les commentaires /** ... */

Objectif principal : Décrire les spécifications d'une API.

Plus de détails sur comment écrire les commentaires Javadoc :
http://www.oracle.com/technetwork/java/javase/documentation/index-137868.html

Décrire votre code dans votre code lui-même plutôt que dans un document séparé vous aide à garder votre documentation à jour.

Page de référence pour Javadoc :
`http://www.oracle.com/technetwork/java/javase/documentation/index-jsp-135444.html#javadocdocuments`

Les commentaires /** ... */

Objectif principal : Décrire les spécifications d'une API.

Plus de détails sur comment écrire les commentaires Javadoc :
`http://www.oracle.com/technetwork/java/javase/documentation/index-137868.html`

Décrire votre code dans votre code lui-même plutôt que dans un document séparé vous aide à garder votre documentation à jour.

Page de référence pour Javadoc :
`http://www.oracle.com/technetwork/java/javase/documentation/index-jsp-135444.html#javadocdocuments`

Les commentaires /** ... */

Objectif principal : Décrire les spécifications d'une API.

Plus de détails sur comment écrire les commentaires Javadoc :
`http://www.oracle.com/technetwork/java/javase/documentation/index-137868.html`

Décrire votre code dans votre code lui-même plutôt que dans un document séparé vous aide à garder votre documentation à jour.

Page de référence pour Javadoc :
`http://www.oracle.com/technetwork/java/javase/documentation/index-jsp-135444.html#javadocdocuments`

Les commentaires /** ... */

Objectif principal : **Décrire les spécifications d'une API.**

Plus de détails sur comment écrire les commentaires Javadoc :
`http://www.oracle.com/technetwork/java/javase/documentation/index-137868.html`

Décrire votre code dans votre code lui-même plutôt que dans un document séparé vous aide à garder votre documentation à jour.

Page de référence pour Javadoc :
`http://www.oracle.com/technetwork/java/javase/documentation/index-jsp-135444.html#javadocdocuments`

Les commentaires /** ... */

Objectif principal : **Décrire les spécifications d'une API.**

Plus de détails sur comment écrire les commentaires Javadoc :
`http://www.oracle.com/technetwork/java/javase/documentation/index-137868.html`

Quelques règles pour Javadoc

- Un commentaire Javadoc n'est pas un tutoriel ou un guide d'utilisation, mais **une spécification**. Si cela est vraiment utile à la compréhension, alors on placera un lien (doc, site Web) vers une explication plus approfondie.
- On ne décrit pas les détails d'implémentations, mais le **comportement d'une méthode**.
- On utilise la 3ème personne dans une description et on évite l'impératif.

Quelques règles pour Javadoc

- Un commentaire Javadoc n'est pas un tutoriel ou un guide d'utilisation, mais **une spécification**. Si cela est vraiment utile à la compréhension, alors on placera un lien (doc, site Web) vers une explication plus approfondie.
- On ne décrit pas les détails d'implémentations, mais le **comportement d'une méthode**.
- On utilise la 3ème personne dans une description et on évite l'impératif.

Quelques règles pour Javadoc

- Un commentaire Javadoc n'est pas un tutoriel ou un guide d'utilisation, mais **une spécification**. Si cela est vraiment utile à la compréhension, alors on placera un lien (doc, site Web) vers une explication plus approfondie.
- On ne décrit pas les détails d'implémentations, mais le **comportement d'une méthode**.
- On utilise la 3ème personne dans une description et on évite l'impératif.

Quelques règles pour Javadoc

- Conditions aux bornes, intervalles des paramètres, comportement dans les cas critiques (e.g. objet `null`, division par zéro).
- Les assertions (pré-conditions et post-conditions) décrites doivent être **indépendantes de l'implémentation**.
- En l'absence de commentaires à ce sujet, une méthode est supposée *thread-safe*.
- On peut éviter de réécrire entièrement les commentaires des méthodes si : elle *redéfinit* une méthode d'une classe parent (@Override), ou si elle *implémente une méthode d'une interface* (cf. cours sur l'héritage). Dans ces cas un lien vers la méthode redéfinie sera rajouté par Javadoc.

Quelques règles pour Javadoc

- Conditions aux bornes, intervalles des paramètres, comportement dans les cas critiques (e.g. objet `null`, division par zéro).
- Les assertions (pré-conditions et post-conditions) décrites doivent être **indépendantes de l'implémentation**.
- En l'absence de commentaires à ce sujet, une méthode est supposée *thread-safe*.
- On peut éviter de réécrire entièrement les commentaires des méthodes si : elle *redéfinit* une méthode d'une classe parent (@Override), ou si elle *implémente une méthode d'une interface* (cf. cours sur l'héritage). Dans ces cas un lien vers la méthode redéfinie sera rajouté par Javadoc.

Quelques règles pour Javadoc

- Conditions aux bornes, intervalles des paramètres, comportement dans les cas critiques (e.g. objet `null`, division par zéro).
- Les assertions (pré-conditions et post-conditions) décrites doivent être **indépendantes de l'implémentation**.
- En l'absence de commentaires à ce sujet, une méthode est supposée *thread-safe*.
- On peut éviter de réécrire entièrement les commentaires des méthodes si : elle *redéfinit* une méthode d'une classe parent (@Override), ou si elle *implémente une méthode d'une interface* (cf. cours sur l'héritage). Dans ces cas un lien vers la méthode redéfinie sera rajouté par Javadoc.

Quelques règles pour Javadoc

- Conditions aux bornes, intervalles des paramètres, comportement dans les cas critiques (e.g. objet `null`, division par zéro).
- Les assertions (pré-conditions et post-conditions) décrites doivent être **indépendantes de l'implémentation**.
- En l'absence de commentaires à ce sujet, une méthode est supposée *thread-safe*.
- On peut éviter de réécrire entièrement les commentaires des méthodes si : elle *redéfinit* une méthode d'une classe parent (@Override), ou si elle *implémente une méthode d'une interface* (cf. cours sur l'héritage). Dans ces cas un lien vers la méthode redéfinie sera rajouté par Javadoc.

Quelques règles pour Javadoc

- Conditions aux bornes, intervalles des paramètres, comportement dans les cas critiques (e.g. objet `null`, division par zéro).
- Les assertions (pré-conditions et post-conditions) décrites doivent être **indépendantes de l'implémentation**.
- En l'absence de commentaires à ce sujet, une méthode est supposée *thread-safe*.
- On peut éviter de réécrire entièrement les commentaires des méthodes si : elle *redéfinit* une méthode d'une classe parent (@Override), ou si elle *implémente une méthode d'une interface* (cf. cours sur l'héritage). Dans ces cas un lien vers la méthode redéfinie sera rajouté par Javadoc.

Quelques règles pour Javadoc

- Conditions aux bornes, intervalles des paramètres, comportement dans les cas critiques (e.g. objet `null`, division par zéro).
- Les assertions (pré-conditions et post-conditions) décrites doivent être **indépendantes de l'implémentation**.
- En l'absence de commentaires à ce sujet, une méthode est supposée *thread-safe*.
- On peut éviter de réécrire entièrement les commentaires des méthodes si : elle *redéfinit* une méthode d'une classe parent (@Override), ou si elle *implémente une méthode d'une interface* (cf. cours sur l'héritage). Dans ces cas un lien vers la méthode redéfinie sera rajouté par Javadoc.

L'ordre (à respecter) des tags Javadoc

1. @author *nom* (classes et interfaces) ;
2. @version %I%, %G% (classes et interfaces) ;
3. @param *nom description* (méthodes et constructeurs) ;
4. @return (méthodes qui retournent qqc différent de void) ;
5. @exception ou @throws (si des exceptions sont jetées) ;
6. @see *type* ;
7. @since *1.0* ;
8. @serial ;
9. @deprecated (depuis quand et quoi utiliser à la place).

Un même tag peut apparaître sur plusieurs lignes successives : @author, @param, @exception, et @see.

L'ordre (à respecter) des tags Javadoc

1. @author *nom* (classes et interfaces) ;
2. @version %I%, %G% (classes et interfaces) ;
3. @param *nom description* (méthodes et constructeurs) ;
4. @return (méthodes qui retournent qqc différent de void) ;
5. @exception ou @throws (si des exceptions sont jetées) ;
6. @see *type* ;
7. @since *1.0* ;
8. @serial ;
9. @deprecated (depuis quand et quoi utiliser à la place).

Un même tag peut apparaître sur plusieurs lignes successives : @author, @param, @exception, et @see.

L'ordre (à respecter) des tags Javadoc

1. @author *nom* (classes et interfaces) ;
2. @version %I%, %G% (classes et interfaces) ;
3. @param *nom description* (méthodes et constructeurs) ;
4. @return (méthodes qui retournent qqc différent de void) ;
5. @exception ou @throws (si des exceptions sont jetées) ;
6. @see *type* ;
7. @since *1.0* ;
8. @serial ;
9. @deprecated (depuis quand et quoi utiliser à la place).

Un même tag peut apparaître sur plusieurs lignes successives : @author, @param, @exception, et @see.

L'ordre (à respecter) des tags Javadoc

1. @author *nom* (classes et interfaces) ;
2. @version %I%, %G% (classes et interfaces) ;
3. @param *nom description* (méthodes et constructeurs) ;
4. @return (méthodes qui retournent qqc différent de void) ;
5. @exception ou @throws (si des exceptions sont jetées) ;
6. @see *type* ;
7. @since *1.0* ;
8. @serial ;
9. @deprecated (depuis quand et quoi utiliser à la place).

Un même tag peut apparaître sur plusieurs lignes successives : @author, @param, @exception, et @see.

L'ordre (à respecter) des tags Javadoc

1. @author *nom* (classes et interfaces) ;
2. @version %I%, %G% (classes et interfaces) ;
3. @param *nom description* (méthodes et constructeurs) ;
4. @return (méthodes qui retournent qqc différent de void) ;
5. @exception ou @throws (si des exceptions sont jetées) ;
6. @see *type* ;
7. @since *1.0* ;
8. @serial ;
9. @deprecated (depuis quand et quoi utiliser à la place).

Un même tag peut apparaître sur plusieurs lignes successives : @author, @param, @exception, et @see.

L'ordre (à respecter) des tags Javadoc

1. @author *nom* (classes et interfaces) ;
2. @version %I%, %G% (classes et interfaces) ;
3. @param *nom description* (méthodes et constructeurs) ;
4. @return (méthodes qui retournent qqc différent de void) ;
5. @exception ou @throws (si des exceptions sont jetées) ;
6. @see *type* ;
7. @since *1.0* ;
8. @serial ;
9. @deprecated (depuis quand et quoi utiliser à la place).

Un même tag peut apparaître sur plusieurs lignes successives : @author, @param, @exception, et @see.

L'ordre (à respecter) des tags Javadoc

1. @author *nom* (classes et interfaces) ;
2. @version %I%, %G% (classes et interfaces) ;
3. @param *nom description* (méthodes et constructeurs) ;
4. @return (méthodes qui retournent qqc différent de void) ;
5. @exception ou @throws (si des exceptions sont jetées) ;
6. @see *type* ;
7. @since *1.0* ;
8. @serial ;
9. @deprecated (depuis quand et quoi utiliser à la place).

Un même tag peut apparaître sur plusieurs lignes successives : @author, @param, @exception, et @see.

L'ordre (à respecter) des tags Javadoc

1. @author *nom* (classes et interfaces) ;
2. @version %I%, %G% (classes et interfaces) ;
3. @param *nom description* (méthodes et constructeurs) ;
4. @return (méthodes qui retournent qqc différent de void) ;
5. @exception ou @throws (si des exceptions sont jetées) ;
6. @see *type* ;
7. @since *1.0* ;
8. @serial ;
9. @deprecated (depuis quand et quoi utiliser à la place).

Un même tag peut apparaître sur plusieurs lignes successives : @author, @param, @exception, et @see.

L'ordre (à respecter) des tags Javadoc

1. @author *nom* (classes et interfaces) ;
2. @version %I%, %G% (classes et interfaces) ;
3. @param *nom description* (méthodes et constructeurs) ;
4. @return (méthodes qui retournent qqc différent de void) ;
5. @exception ou @throws (si des exceptions sont jetées) ;
6. @see *type* ;
7. @since *1.0* ;
8. @serial ;
9. @deprecated (depuis quand et quoi utiliser à la place).

Un même tag peut apparaître sur plusieurs lignes successives : @author, @param, @exception, et @see.

Exemple de documentation Javadoc

```java
/**
* Addition de deux entiers.
* @param a Le premier entier.
* @param b Le second entier.
* @return La somme de a et de b.
*/
public int addition (int a, int b) {
   return a + b ;
}
```

Exemple de documentation Javadoc

```
/**
* Division entière.
* @param a Le premier entier.
* @param b Le second entier.
* @return Le résultat de la division entière de a par b.
* @throws ArithmeticException lorsque b vaut 0.
*/
public int division (int a, int b) throws
    ArithmeticException {
   return a / b ;
}
```

Exemple de documentation Javadoc

```
/**
* Classe représentant un enseignant.
* @author Jean Dupont
* @version 4.2
* @see Etudiant
*/
public class Enseignant extends Personne {
    ...
}
```

Exemple de documentation Javadoc

http://www.oracle.com/technetwork/java/javase/documentation/index-137868.html#examples

Comment générer la documentation Javadoc ?

Grâce au programme javadoc.exe fourni dans le Java SDK, et customisé via des Javadoc doclets :

- **NetBeans** : *cliquez-droit sur le projet* et choisissez Generate Javadoc. Par défaut, le doclet génère les fichiers de documentation dans le répertoire javadoc de votre répertoire utilisateur. Shift+F1 pour rechercher dans la documentation générée.
- **eclipse** : *cliquer sur le menu Projet*. Puis sélectionnez l'option "Générer la Javadoc".
- **IntelliJ** : *cliquer sur le menu Tools*. Puis sélectionnez l'option "Generate Javadoc...".

Comment générer la documentation Javadoc ?

Grâce au programme javadoc.exe fourni dans le Java SDK, et customisé via des Javadoc doclets :

- **NetBeans** : *cliquez-droit sur le projet* et choisissez Generate Javadoc. Par défaut, le doclet génère les fichiers de documentation dans le répertoire javadoc de votre répertoire utilisateur. Shift+F1 pour rechercher dans la documentation générée.
- **eclipse** : *cliquer sur le menu Projet*. Puis sélectionnez l'option "Générer la Javadoc".
- **IntelliJ** : *cliquer sur le menu Tools*. Puis sélectionnez l'option "Generate Javadoc...".

Comment générer la documentation Javadoc ?

Grâce au programme javadoc.exe fourni dans le Java SDK, et customisé via des Javadoc doclets :

- **NetBeans** : *cliquez-droit sur le projet* et choisissez Generate Javadoc. Par défaut, le doclet génère les fichiers de documentation dans le répertoire javadoc de votre répertoire utilisateur. Shift+F1 pour rechercher dans la documentation générée.
- **eclipse** : *cliquer sur le menu Projet*. Puis sélectionnez l'option "Générer la Javadoc".
- **IntelliJ** : *cliquer sur le menu Tools*. Puis sélectionnez l'option "Generate Javadoc...".

Comment générer la documentation Javadoc ?

Grâce au programme javadoc.exe fourni dans le Java SDK, et customisé via des Javadoc doclets :

- **NetBeans** : *cliquez-droit sur le projet* et choisissez Generate Javadoc. Par défaut, le doclet génère les fichiers de documentation dans le répertoire javadoc de votre répertoire utilisateur. Shift+F1 pour rechercher dans la documentation générée.
- **eclipse** : *cliquer sur le menu Projet*. Puis sélectionnez l'option "Générer la Javadoc".
- **IntelliJ** : *cliquer sur le menu Tools*. Puis sélectionnez l'option "Generate Javadoc...".