Mickaël Kerboeuf

Algorithmique et programmation objet

Java en résumé

200 exercices corrigés



ellipses

Références sciences

Algorithmique et programmation objet

Java en résumé

200 exercices corrigés

Mickaël Kerboeuf

ellipses

Collection Références sciences

dirigée par Paul de Laboulaye
paul.delaboulaye@editions-ellipses.fr

Retrouvez tous les livres de la collection et des extraits sur www.editions-ellipses.fr

ISBN 9782340-037984
©Ellipses Édition Marketing S.A., 2020
32, rue Bargue 75740 Paris cedex 15

DANGER
LE PHOTOCOPILLAGE TUE LE LIVRE

Le Code de la propriété intellectuelle n'autorisant, aux termes de l'article L. 122-5.2° et 3°a), d'une part, que les « copies ou reproductions strictement réservées à l'usage privé du copiste et non destinées à une utilisation collective », et d'autre part, que les analyses et les courtes citations dans un but d'exemple et d'illustration, « toute représentation ou reproduction intégrale ou partielle faite sans le consentement de l'auteur ou de ses ayants droit ou ayants cause est illicite » (art. L. 122-4).
Cette représentation ou reproduction, par quelque procédé que ce soit constituerait une contrefaçon sanctionnée par les articles L. 335-2 et suivants du Code de la propriété intellectuelle.

www.editions-ellipses.fr

Avant-propos

Cet ouvrage s'adresse aux personnes désireuses d'acquérir ou de consolider des connaissances en algorithmique et structures de données dans le cadre particulier de la programmation orientée objet avec le langage Java. Pour aborder cet ouvrage, il peut être utile d'avoir quelques notions de programmation procédurale et d'être familiarisé avec les concepts de types de données, d'instructions de contrôle (*e.g.* conditionnelle, itération), de tableaux et de chaînes de caractères [1, 2, 7, 11].

Dans une première partie, les points fondamentaux de la programmation objet avec Java sont rappelés et résumés : constructions impératives, conception objet, encapsulation, héritage, exceptions, généricité, collections, flots de données et *threads*. Chacun de ces points est illustré par une série d'exercices corrigés. À la fin de cette première partie, quatre sujets de synthèse permettent d'aborder tous ces points à une échelle plus conséquente.

Dans une deuxième partie, les structures de données fondamentales qui sous-tendent l'algorithmique traditionnelle sont définies, analysées et outillées en Java. Ces structures sont les listes simplement et doublement chaînées, les arbres quelconques, les arbres binaires de recherche, et les arbres AVL. Ces structures sont ensuite utilisées pour réaliser une authentique API orientée objet, elle-même utilisée pour réaliser une application exigeante en termes de performance (temps d'exécution et occupation mémoire). Dans la continuité de la première partie, cette deuxième partie comporte de nombreux exercices corrigés.

Table des matières

I Programmation Orientée Objet avec Java **7**

1 Fondements impératifs de Java **9**
1.1 Compilateur et machine virtuelle 9
1.2 Syntaxe élémentaire de Java 11
1.3 Organisation du code et API standard 20
1.4 Exercices 24
1.5 Corrections 32

2 Programmation Objet **43**
2.1 Principes de conception objet 43
2.2 Programmation objet avec Java 46
2.3 Exercices 57
2.4 Corrections 79

3 Compléments de programmation **107**
3.1 Introspection 107
3.2 Exceptions 110
3.3 Types énumérés 113
3.4 Généricité 115
3.5 Lambda expressions 118
3.6 Exercices 121
3.7 Corrections 126

4 Compléments d'API **135**
4.1 Collections 135
4.2 Flots de données 145
4.3 Programmation concurrente 152
4.4 Exercices 156
4.5 Corrections 192

II Algorithmique et structures de données avec Java 231

5 Introduction 233
5.1 Objectifs . . . 233
5.2 Fonctionnalités et performances . . . 234

6 Structures de données récursives 237
6.1 Chaînes . . . 237
6.2 Arbres . . . 241
6.3 Package sdr . . . 256
6.4 Corrections . . . 257

7 Collections de données 275
7.1 Collections . . . 275
7.2 Listes . . . 276
7.3 Ensembles . . . 279
7.4 Ensembles ordonnés . . . 281
7.5 Package col . . . 283
7.6 Corrections . . . 284

8 Dictionnaire de mots 293
8.1 Principes et interface . . . 293
8.2 Mise en œuvre avec un ensemble ordonné . . . 294
8.3 Mise en œuvre avec un arbre de mots . . . 295
8.4 Package dic . . . 295
8.5 Corrections . . . 296

Bibliographie 299

Index 301

Première partie

Programmation Orientée Objet avec Java

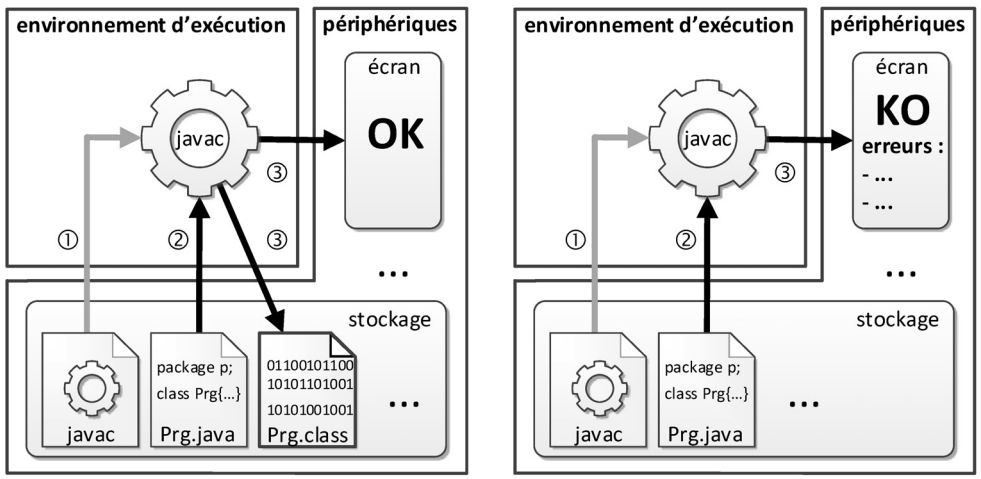
Chapitre 1

Fondements impératifs de Java

1.1 Compilateur et machine virtuelle 9
1.2 Syntaxe élémentaire de Java 11
1.3 Organisation du code et API standard 20
1.4 Exercices 24
1.5 Corrections 32

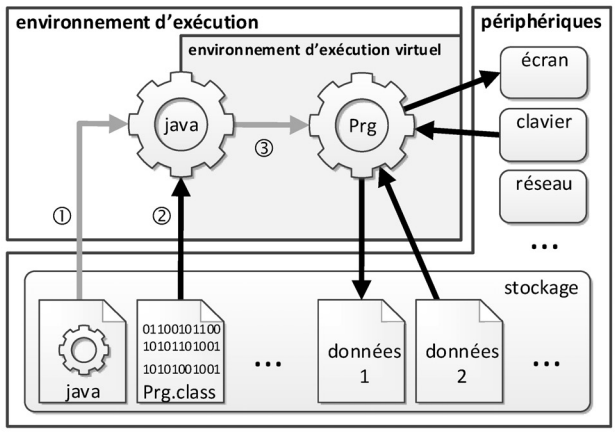
1.1 Compilateur et machine virtuelle

Compilateur Un programme Java [4] est transformé par le *compilateur* [6] en une série d'instructions en langage machine exécutable appelé *bytecode*. Au cours de la transformation, des vérifications sont faites et le programme peut être *rejeté* :



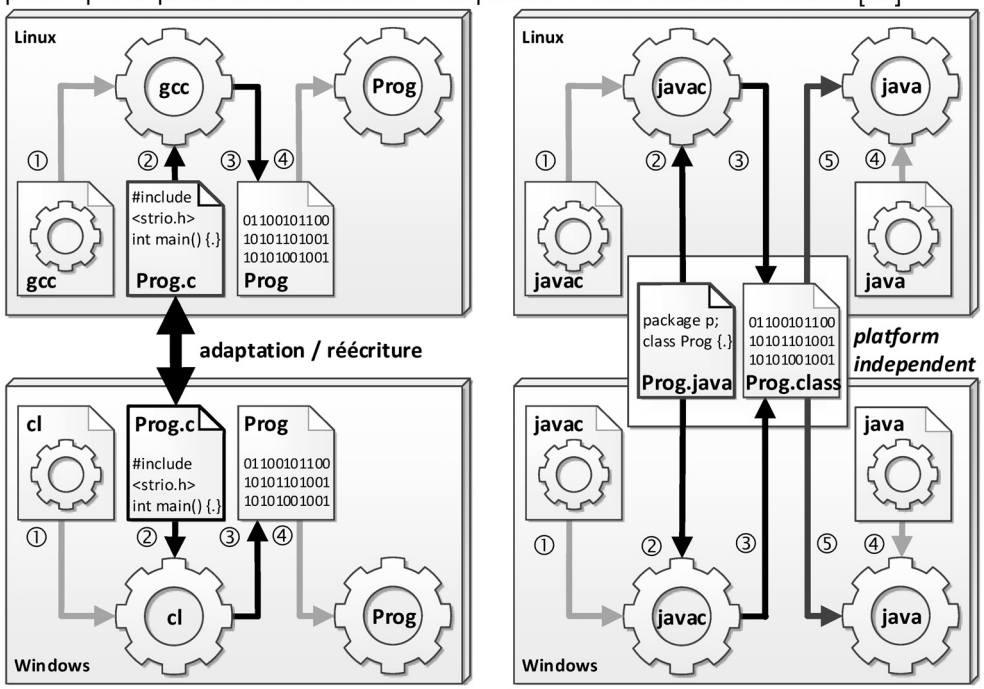
1. exécution du compilateur (`javac`) avec en paramètre le nom du fichier `Prg.java`
2. chargement et compilation du fichier de code source `Prg.java`
3. production du bytecode `Prg.class` (à gauche), ou messages d'erreurs (à droite)

Machine virtuelle Le *bytecode* est chargé en mémoire et exécuté par une *machine virtuelle* (JVM) qui simule le fonctionnement d'une machine réelle :



1. exécution de la JVM avec en paramètre le nom du programme à charger (`Prg`)
2. chargement du bytecode `Prg.class`
3. exécution dans l'environnement virtuel du programme `Prg`

Portabilité Le *bytecode* produit par le compilateur peut être exécuté sur n'importe quelle plateforme d'exécution disposant d'une machine virtuelle [10] :



1.2 Syntaxe élémentaire de Java

1.2.1 Commentaires et annotations

Les commentaires et les annotations apparaissent dans le code source mais ne font pas partie du programme. Illustration :

```
// commentaire sur 1 ligne
/* commentaire
   sur plusieurs lignes...  */
int x = 0; // commentaire sur la variable x...
int y = /* commentaire au milieu d'une instruction ! */ 0;
```

```
/** Ce commentaire long est destiné à javadoc.
    Il documente la classe A suivante... */
class A{ /* ... */ }
```

Une annotation est une instruction typiquement destinée au compilateur accolée à un élément du programme (classe, attribut, instruction, etc.). Elle correspond à un mot-clé précédé du symbole `@`. Exemple :

```
@SuppressWarnings("unchecked")
public class A { /* ... */ }
```

1.2.2 Identifiants

Un identifiant permet de nommer une variable, un attribut, un paramètre, une classe, une interface, un type énuméré, une méthode ou un package. Sa première lettre est soit un caractère alphabétique non accentué, soit l'un des deux caractères `$` ou `_`. Il n'y a pas de contrainte pour les autres lettres.

Conventions Un nom de classe, d'interface ou de type énuméré commence par une majuscule. Une constante est entièrement en majuscules. Les autres identifiants commencent par une minuscule. Les caractères accentués sont déconseillés.

Mots réservés Les mots réservés du langage ne peuvent pas être utilisés comme identifiant (*e.g.* `class`, `while`, `true`, `return`, etc.).

1.2.3 Types primitifs

Les types primitifs représentent des données simples, manipulées par *valeur*, et qui peuvent généralement être désignées directement par des *littéraux* :

- boolean : littéraux `true` et `false`
- char : exemples de littéraux : `'a'`, `'5'`, `'\\'`, `'\''`, `'\u0BA3'`

- types des nombres entiers :

type	décimal	binaire	hexa.	encodage
`byte`	∅	∅	∅	1 octet
`short`	∅	∅	∅	2 octets
`int`	`27`	`0b11011`	`0x1B`	4 octets
`long`	`27l`	`0b11011l`	`0x1Bl`	8 octets

- types des nombres flottants :

type	décimal	binaire	hexa.	encodage
`float`	`6.4375E2f`	∅	`0xA1.8P2f`	4 octets
`double`	`6.4375E2`	∅	`0xA1.8P2`	8 octets

Transtypage automatique Une valeur d'un certain type primitif peut être admise là où une valeur d'un type primitif *éventuellement différent* est attendue :

Valeur attendue	Valeur admise
`boolean`	`boolean`
`char`	`char`, `byte`, `short`, `int` (encodage unicode de l'entier fourni)
`byte`	`byte`
`short`	`short`, `byte`
`int`	`int`, `short`, `byte`, `char` (valeur unicode du caractère fourni)
`long`	`long`, `int`, `short`, `byte`, `char`
`float`	`float`, `long`, `int`, `short`, `byte`, `char`
`double`	`double`, `float`, `long`, `int`, `short`, `byte`, `char`

1.2.4 Types structurés

Un *type structuré* correspond à une *classe*. Ses instances sont les valeurs du type. Elles ne peuvent pas être *physiquement stockées* dans des variables. Elles sont donc manipulées par l'intermédiaire d'*adresses mémoire* appelées *références*. Le littéral `null` représente une référence qui ne désigne aucune instance.

1.2.5 Déclaration et affectation

Déclaration La déclaration d'une variable associe un *nom* à un *type* :

```
int i1;       // déclaration de i1, de type int (primitif)
Integer i2;   // déclaration de i2, de type Integer (classe)
Cloneable c; // déclaration de c, de type Cloneable (interface)
int x, y, z; // déclaration simultanée de 3 variables de même type
```

Une variable a une *portée* limitée au bloc de code dans lequel elle est définie. Le *masquage* de variables est interdit. Une variable déclarée au niveau d'une classe

est appelée *attribut*. Dans tous les autres cas, une variable est dite *locale*. Les paramètres d'une fonction sont assimilables à des variables locales à la fonction. La notion de variable globale n'existe pas en Java.

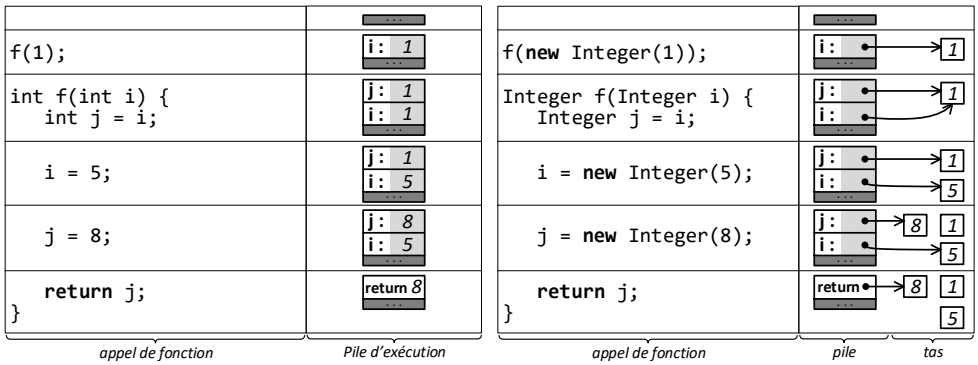
Affectation et valeurs initiales La valeur affectée à une variable est soit un littéral (*e.g.* `-18`, `true`, `'c'`, ou `null`), soit le contenu d'une autre variable, soit la valeur d'une expression, soit le résultat d'un appel de fonction. Les variables peuvent être initialisées à leur déclaration. Exemples :

```
int x, y       // déclarations combinées de deux variables
x = 15;        // affectation d'un littéral
y = x;         // affectation du contenu d'une autre variable
int z = 8      // déclaration et initialisation
int a = 2*5;   // initialisation par une expression
int b = f();   // initialisation par le résultat de l'appel de f
```

Les variables locales doivent être initialisées avant d'être lues, contrairement aux *attributs* qui disposent de valeurs initiales par défaut.

1.2.6 Allocation de mémoire

L'allocation statique n'existe pas en Java. Il n'est possible d'allouer la mémoire que dynamiquement à l'appel d'une fonction (allocation sur la pile) ou à l'exécution de new (allocation sur le tas). L'opérateur new est réservé à l'*instanciation* de classes. Le *garbage collector* se charge de libérer l'espace alloué par new. Illustration :



1.2.7 Expressions

Une *expression* est une combinaison de littéraux, de variables, d'opérateurs et d'appels de fonctions. L'*évaluation* d'une expression produit une *valeur primitive* ou une *référence*.

Opérateurs sur les nombres S'appliquent à `byte`, `short`, `int`, `long`, `float`, `double` :

Opérateurs arithmétiques usuels	`+`, `-`, `*`, `/`, `%`
Opérateurs bit à bit usuels	`~`, `&`, `|`, `^`
Opérateurs rotation de bits	`<<`, `>>`, `>>>`
Opérateurs de comparaison	`<`, `<=`, `>`, `>=`
Opérateurs d'incrémentation et de décrémentation	`++`, `--`

Opérateurs sur les booléens Ils s'appliquent au type `boolean` :

Opérateurs logiques usuels	`!`, `&&`, `||`
Opérateur conditionnel	`(?:)`

Opérateurs sur les objets Il s'appliquent à une *référence* d'objet :

Test explicite de type d'objet	`instanceof`
Accès aux champs d'une structure	`.` (opérateur *point*)

Un objet peut être *polymorphe*, *i.e.* conforme à plusieurs types. L'opérateur `instanceof` permet de tester si l'objet référencé est *au moins* du type correspondant à la deuxième opérande (le nom d'une classe, d'une interface ou d'un type énuméré). Exemple :

```
Object x = /* valeur initiale quelconque... */ ;
boolean test1 = x instanceof Integer; // x est-il un Integer ?
boolean test2 = x instanceof Object;  // x est-il un Object ? (oui)
```

En Java, les structures sont des classes et leurs champs sont des *attributs* ou des *méthodes*. L'opérateur *point* permet d'accéder au champ indiqué :

```
class A {
  int x;                    // attribut x (variable)
  void f() { /* ... */ }  // méthode f (fonction)
}
A refObjetA = new A();
refObjetA.x = 3; // accès au champ x (écriture de variable)
refObjetA.f();   // accès au champ f (appel de fonction)
```

Opérateurs sur des objets ou des valeurs primitives Ils s'appliquent indifféremment à des valeurs primitives ou à des références d'objets :

Test d'égalité et de différence	`==`, `!=`
Coercition de type (*cast*)	`( · ) ·`

Une valeur primitive numérique peut être *castée* en un type primitif numérique différent auquel cas la donnée initiale peut être modifiée :

```
int   b = 130;
byte  o = (byte)b; // o : -126 ((130+128)%256 - 128)
float f = 3.56;
int   i = (int)f;  // i :    3 (partie entière de 3.56)
```

Un objet peut être *casté* en un type correspondant à une classe seulement si cet objet a *effectivement* ce type parmi tous ses types (objet polymorphe) :

```
Object o1 = new Integer(3); // Ok : Integer hérite de Object
Object o2 = new Object();
Integer j = (Integer)o1;    // Ok.
Integer k = (Integer)o2;    // Ok... Mais erreur à l'exécution !
```

Une valeur primitive peut être *castée* en un type correspondant à une classe seulement si cette classe correspond à une *classe enveloppe*. Les valeurs primitives peuvent être *automatiquement encapsulées* dans un objet de la classe associée :

```
Integer i = new Integer(3); // création explicite d'objet
Integer i = 3;              // Ok : raccourci de la ligne précédente
Integer i = (Integer)3;     // Idem ligne précédente ; cast superflu
Object  o = 3;              // Ok : équivaut à new Integer(3)
Object  o = (Object)3;      // Idem ligne précédente ; cast superflu
```

Inversement, seules les instances des classes enveloppes peuvent être utilisées là où des valeurs primitives sont requises. Dans ce cas, le cast est valide et superflu. Dans tous les autres cas, il est invalide et rejeté par le compilateur :

```
Integer n = new Integer(3);
int     i = n;      // Ok : raccourci de int i = Integer.valueOf(n);
int     i = (int)n; // Idem ligne précédente ; cast superflu
```

Opérateurs spéciaux Opérateurs hors catégories :

Concaténation de chaînes de caractères	+
Affectations	=, +=, -=, *=, /=
Instanciation	new
Accès tableau	· [·]

Les chaînes de caractères sont des *objets* de type `String` et peuvent être désignés par des *littéraux* comme "abc". L'opérateur + correspond à la concaténation de chaînes *seulement si* au moins une opérande est de type `String`. Si l'autre opérande est une valeur primitive, sa représentation littérale est utilisée. Si l'autre opérande est un objet, le résultat de `toString` invoqué sur cet objet est utilisé :

```
"2"+2                    // valeur : "22"
Object o = null;  "2"+o // valeur : "2null"
o = new Object(); "2"+o // implicitement "2"+o.toString()
```

1.2.8 Structures de contrôle

En Java, les instructions s'enchaînent par défaut et implicitement en *séquence*. Les *structures de contrôle* sont les instructions qui modifient cet enchaînement.

Itérations L'itération fondamentale est la boucle *while*. Exemple :

```
while (i<10)  { /* ... */ i++; } // condition de sortie : i>=10
```

Variante *do ... while* : le corps est exécuté au moins une fois :

```
do { i = /* lecture d'un entier supérieur à 10 */ ; }
while (i<10);
```

Variante *for* : intègre des clauses d'initialisation et de mise à jour :

```
for (int i=0; i<10; i++) { /* 10 itérations... */ }
```

Variante de type *for each* : dédiée au parcours séquentiel en lecture seule des éléments d'une *collection* ou d'un tableau :

```
char[] tab = new char[10]; // création d'un tableau de 10 caractères
for(char x : tab) {/* x prend toutes les valeurs de tab */}
```

Conditionnelles La conditionnelle simple utilise un booléen pour exécuter une certaine portion de code (obligatoire) ou une autre (optionnelle) :

```
if (a<0)  { a=b; } else { a=c; }
```

Le sélecteur (ou *switch*) utilise une valeur énumérée pour sélectionner une portion de code à exécuter. Cette valeur peut être un entier, un caractère, une chaîne de caractères (classe `String`), ou un objet de type `Enum`. Exemple :

```
int i = /* ... */ ;
switch (i) { // sélecteur
   case 0  : /* ... */  break;
   case 1  : /* ... */  break;
   default : /* ... */
}
```

Sauts et arrêts inconditionnels Les instructions `continue` et `break` interrompent une boucle. Après `continue`, l'exécution reprend au début de la boucle. Après `break`, l'exécution reprend à la suite de la boucle :

```
String suiteNombres ="";
for (int j = 0; j<10; j++) { // valeurs de j envisagées : 0, 1 ... 9
   if (j == 4) continue;     // valeur 4 exclue
   if (j == 7) break;        // valeurs 7 et suivantes exclues
   suiteNombres += j+"␣";    // concaténation de la valeur retenue
} // suiteNombres : "0 1 2 3 5 6 "
```

Appel et retour de fonction L'appel de fonction consiste à invoquer une portion de code identifiée par une *signature*. L'exécution de la fonction prend fin grâce à l'instruction `return`. Les fonctions en Java sont appelées *méthodes* et sont définies au niveau d'une classe :

```
class A { // déclaration de classe
   int fact (int n) { // déclaration de méthode
      int res = 1;
      for (int i=n; i>1; i--) res *= i;
      return res;    // interruption et renvoi du résultat
   }
   /* ... */
   int i = fact(5);  // appel de méthode
}
```

Exceptions Lorsqu'une *exception* survient, elle interrompt d'exécution du programme. Elle peut éventuellement être *rattrapée* auquel cas le programme reprend son exécution. En Java, les exceptions sont des *objets* manipulés par les instructions `throw` (déclenchement d'exception) et `try catch` (rattrapage d'exception).

1.2.9 Chaînes de caractères

Une chaîne de caractères est un objet instance de la classe `String`. Un simple caractère ou un tableau de caractères peuvent être utilisés pour créer une chaîne :

```
String s1 = new String('a');   // chaîne "a"
char data[] = {'a', 'b', 'c'}; // tableau de caractères
String s2 = new String(data);  // chaîne "abc"
```

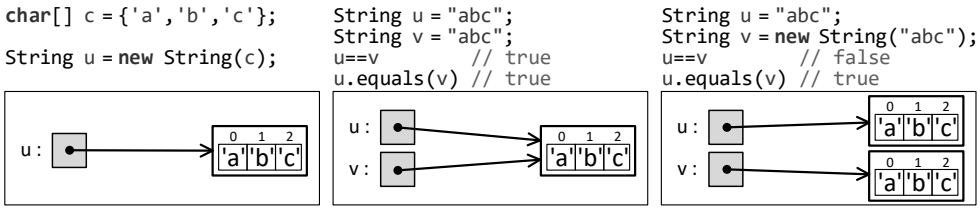
Une chaîne de caractères peut être désignée plus simplement par un littéral spécifique délimité par le caractère " :

```
String s3 = "abc"; // chaîne "abc"
```

Égalité de chaînes Pour tester l'*égalité de chaînes*, il faut utiliser la méthode `equals` et non pas l'opérateur == :

```
boolean test; String s1 = /*...*/ ; String s2 = /*...*/ ;
test = (s1 == s2);      // test d'égalité d'adresses d'objets
test = (s1.equals(s2)); // test d'égalité des chaînes
```

Partage de chaînes Une chaîne de caractères est un objet *non modifiable*. À chaque littéral de type `String` correspond un *unique* objet qui le représente. Pour créer deux objets différents qui représentent une même chaîne, on peut utiliser l'opérateur new :



Opérateur de concaténation L'opérateur + s'applique à *au moins une* opérande de type chaîne de caractères. Si aucune opérande n'est de type `String`, alors le compilateur interprète l'opérateur + comme l'*addition*. Illustration :

```
String s = 1+2+"3"+4+5; // s contient "3345"
```

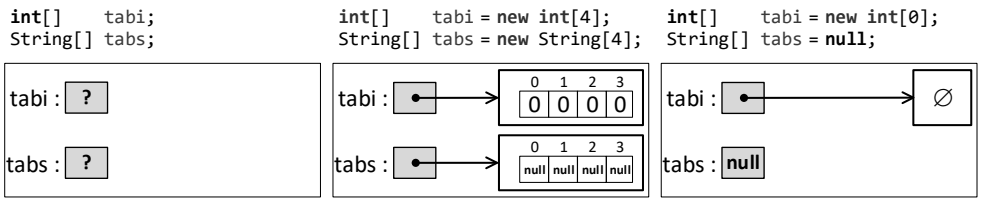
Fonctions de manipulation La classe `String` comporte de nombreuses fonctions de manipulation de chaînes :

```
int     length()                        // taille de la chaîne
char    charAt(int i)                   // caractère d'indice i
char[]  toCharArray()                   // transformation en tableau
boolean equalsIgnoreCase(String s)      // equals, casse-indépendant
int     compareTo(String s)             // comparaison
int     compareToIgnoreCase(String s)   // idem, casse-indépendant
String  substring(int i, int f)         // sous-chaîne [i : f[
boolean startsWith(String prefix)       // test d'existence de préfixe
boolean endsWith(String suffix)         // test d'existence de suffixe
String  toLowerCase()                   // mise en minuscules
String  toUpperCase()                   // mise en majuscules
String  replace(char c1, char c2)       // remplacement de caractère
String  replaceAll(String e, String r)// remplacement de sous-chaîne
static  String valueOf(boolean b)       // transfo. booléen -> String
static  String valueOf(char c)          // transfo. caract. -> String
static  String valueOf(int i)           // transfo. entier  -> String
```

1.2.10 Tableaux

Un tableau est un objet qui représente une collection de valeurs indicées. Sa dimension ne doit être fournie qu'à sa *création* (par l'intermédiaire de l'opérateur new) et non pas à sa *déclaration*. L'*attribut* `length` d'un tableau contient sa taille. Les cases sont indicées à partir de 0. L'opérateur *crochet* (`[·]`) permet d'y accéder.

Tableau à une dimension Le type d'un tableau est défini par le *type* de ses cases suivi d'une paire de crochets. Un tableau *déclaré* n'est pas *créé*. C'est l'opérateur new qui permet de le créer à partir de sa *dimension*. Comme pour des attributs, les cases ont des valeurs initiales par défaut qui dépendent de leur type :



Un tableau peut être alloué et initialisé directement grâce aux raccourcis suivants :

```
int[] tabi = {8, -1, 3, 0};//déclaration + création + initialisation
tabi = new int[]{7, 6, 25};//création avec new + initialisation
```

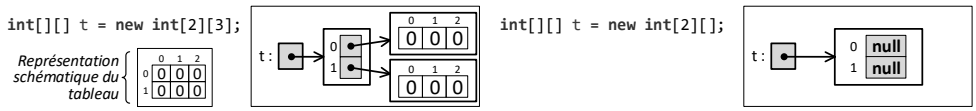
Un tableau peut être parcouru grâce à un indice variant de 0 (inclus) à la taille du tableau (exclue) fournie par `length` :

```
String[] tab; /* ... */
for (int i=0; i<tab.length; i++)
    tab[i] = "abc"; // toutes les cases sont mises à jour
```

La boucle de type *for each* s'applique à un tableau (parcours en *lecture seule*) :

```
String[] tab; /* ... */ String concat="";
for (String s : tab)
    concat += s; // toutes les cases sont lues
```

Tableau à deux dimensions Un tableau à 2 dimensions est un tableau à 1 dimension de tableaux à 1 dimension (*tableau de lignes*). Son type est défini par le *type* des cases suivi de deux paires de crochets. Sa création nécessite de connaître *au minimum* le nombre de lignes à allouer :



Raccourcis de création :

```
int[][] tabi = { {8, -1, 3, 0} , {1, -1,  8, 5} };
tabi         = new int[][]{ {8, -1} , {1, -1}, {8, 5} };
```

Le parcours d'un tableau à 2 dimensions correspond typiquement à un parcours des cases (par une boucle *for*) de chaque *ligne* (par une boucle *for* englobante) :

```
for (int numL=0; numL<tab.length; numL++) // lignes
   for (int numC=0; numC<tab[numL].length; numC++) // colonnes
      concat += tab[numL][numC];
```

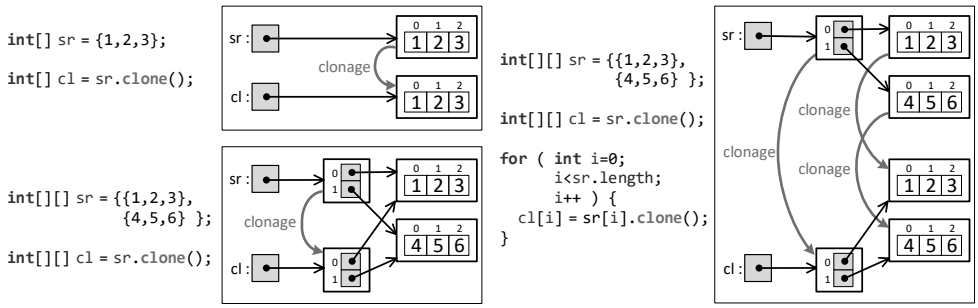
Variante avec une boucle de type *for each* :

```
for (String[] ligne : tab) // lignes
   for (String case : ligne) // colonnes
      concat += case;
```

Clonage de tableaux Les tableaux sont des objets *clonables*. Ils disposent de la méthode clone suivante, qui fournit une *copie superficielle* :

```
public Object clone() { /*...*/ }
```

Le clone d'un tableau d'objets partage donc les mêmes références d'objets que sa source. C'est en particulier le cas d'un tableau à plusieurs dimensions. Dans ce cas, il peut être utile de cloner récursivement les sous-tableaux.



Fonctions de manipulation Les classes utilitaires System et Arrays fournissent des fonctions dédiées à la manipulation des tableaux. Exemples :

```
/* Dans la classe System */
public static void arraycopy(Object s, int i, Object d, int j,int l)
       // copie de l cases de s à partir de i dans d à partir de j
```

```
/* Dans la classe Arrays */
public static int[] copyOf(int[] original, int newLength)
public static int[] copyOfRange(int[] original, int from, int to)
       // copie de tableau ou d'extrait de tableau
public static boolean equals(int[] a, int[] a2)
public static boolean deepEquals(Object[] a1, Object[] a2)
       // test d'égalité superficielle ou en profondeur de tableaux
public static String toString(int[] a)
public static String deepToString(Object[] a)
       // représentation superficielle ou en profondeur de a
public static void fill(int[] a, int val)
public static void fill(int[] a, int f, int t, int val)
       // remplissage de a par val, éventuellement entre f et t
public static void sort(int[] a) // tri de tableau
```

1.3 Organisation du code et API standard

1.3.1 Packages Java

Un *package* rassemble des *types* (classes, interfaces, types énumérés). Il permet de rendre relatif au package le nom de ses éléments, et de limiter leur visibilité.

Déclaration Les types déclarés dans un fichier dont la première ligne est la directive package p appartiennent *par définition* au package p. Sans cette directive, ils appartiennent par défaut au *package anonyme* (à éviter).

Visibilité et accès Par défaut, les membres d'un package ne sont accessibles qu'aux autres membres du même package. Si leur déclaration est précédée du modificateur `public`, alors ils sont accessibles de l'extérieur du package. Pour les désigner de l'extérieur, ils doivent être *préfixés* par le nom du package.

Contraintes d'unicité et de nom Il ne peut pas y avoir plus d'un élément public par fichier source, et si un fichier en comporte un, il doit porter son nom :

```
package p; /* fichier A.java, contient des éléments de p */
public class A { /*...*/ } // élément publique
class B { /*...*/ }        // élément restreint à p
```

```
package p; /* fichier X.java, contient des éléments de p */
class C {
   A a; // attribut de classe A, du même package p : Ok.
   B b; // attribut de classe B, du même package p : Ok.
}
```

```
package q; /* fichier Y.java, contient des éléments de q */
class C {
   p.A a; // attribut de classe A, membre publique de p : Ok.
   p.B b; // attribut de classe B, membre restreint à p : ERREUR !
}
```

Import La *directive d'import* placée juste après la déclaration de *package* permet d'accéder directement aux éléments d'un *autre package* :

```
import p.A; // pour tous les éléments publiques de p : import p.*;
class C {
   A a; // A : raccourci de p.A
}
```

1.3.2 Structure de l'API et classes fondamentales

L'API standard de Java [5] contient de nombreuses classes et interfaces organisées en *packages*. Le package fondamental `java.lang` contient en particulier les classes `Object` et `String`. Chaque fichier source java contient implicitement la clause `import java.lang.*` :

Package	Usage
`java.lang`	éléments fondamentaux du langage
`java.io`	gestion des entrées/sorties
`java.nio`	compléments de gestion des entrées/sorties
`java.math`	objets et fonctions mathématiques complémentaires
`java.net`	gestion des connexions réseau
`java.text`	fonctions d'analyse syntaxique (*parsing*)
`java.util`	structures et fonctions utilitaires, dont les collections
`java.time`	Gestion du temps, des durées et des dates (depuis Java 8)
`java.applet`	création d'applications légères pour clients web
`java.beans`	gestion des *JavaBeans* (composants Java réutilisables)
`java.awt`	*Abstract Window Toolkit* : rudiments d'IHM
`java.rmi`	*Remote Method Invocation* : programmation distribuée
`java.security`	sécurité, contrôle d'accès et cryptographie
`java.sql`	JDBC (*Java Database Connectivity*) : gestion des SGBDR
`javax.rmi`	compléments de Java RMI
`javax.swing`	compléments d'IHM pour AWT

Classe `Object` Racine de la *hiérarchie* des classes Java : tous les objets sont de type `Object` et disposent donc de ses méthodes (*e.g.* `equals` et `toString`) :

```
public boolean equals   (Object obj) {...} // critère d'égalité
public String  toString ()           {...} // représentation
```

Classes de types primitifs Classes *wrappers* ou *classes enveloppes* définies dans le package `java.lang` qui encapsulent des valeurs primitives constantes :

Types primitifs	Classes correspondantes
`boolean`	`Boolean`
`char`	`Character`
`byte`, `short`, `int`, `long`	`Byte`, `Short`, `Integer`, `Long`, héritent de `Number`
`float`, `double`	`Float`, `Double`, héritent de `Number`

Chacune de ces classes dispose d'un constructeur qui encapsule une valeur primitive, et de fonctionnalités dédiées. Par exemple, si on note `type` un type primitif (*e.g.* `int`) et `Type` la classe correspondante (*e.g.* `Integer`), alors la classe `Type` dispose dans tous les cas du constructeur et des méthodes de conversion suivantes :

```
Type(type t)                    // constructeur : encapsule t
static Type valueOf(type t)     // alternative au constructeur
type typeValue()                // restitution de valeur encapsulée
static type parseType(String s) // conversion String -> type
```

La valeur encapsulée est constante. Il n'est donc pas possible de la modifier. Par conséquent, l'instance peut être partagée sans risque. Les types numériques sont représentés par des classes qui *héritent toutes* de la classe `Number`. Elles disposent donc toutes des fonctionnalités de conversion suivantes :

```
byte   byteValue()   // conversion vers byte
double doubleValue() // conversion vers double
float  floatValue()  // conversion vers float
int    intValue()    // conversion vers int
long   longValue()   // conversion vers long
short  shortValue()  // conversion vers short
```

Classes utilitaires Classes comme `System`, `Math` et `Arrays` qui ne sont pas destinées à être instanciées, et qui rassemblent des constantes et des fonctions :

```
/****** Classe java.lang.System ******/

// Canaux d'entrée/sortie standards :
static InputStream in;  // The "standard" input stream.
static PrintStream out; // The "standard" output stream.
static PrintStream err; // The "standard" error output stream.
// Copie de tableaux (cf. section "Tableaux")
// Instant courant :
public static long currentTimeMillis() {/*...*/}// en millisecondes
public static long nanoTime() {/*...*/}         // en nanosecondes
// Arrêt de la JVM :
public static void exit(int status) {/*...*/}
// Appel du garbage collector :
public static void gc() {/*...*/}
```

```
/****** Classe java.lang.Math ******/

static double E;  // constante d'Euler
static double PI; // pi
static double abs(int a) {/*...*/} // valeur absolue
static double sqrt(double a) {/*...*/} // racine carrée
static double cbrt(double a) {/*...*/} // racine cubique
static double cos(double a) {/*...*/} // cosinus
static double log(double a) {/*...*/} // logarithme (ln)
static double log10(double a) {/*...*/} // log. base 10
static double max(double a, double b) {/*...*/} // plus grand
static double min(double a, double b) {/*...*/} // plus petit
static double pow(double a, double b) {/*...*/} // puissance
static double random() {/*...*/} // valeur aléatoire dans [0 - 1[
```

```
/****** Classe java.util.Arrays ******/

// Manipulation de tableaux (cf. section "Tableaux")
```

1.4 Exercices

Exercice 1

Définissez une fonction `max` qui s'applique à 3 entiers et qui renvoie en résultat la plus grande valeur des 3 :

- version 1 : en utilisant l'*instruction* conditionnelle
- version 2 : en utilisant l'*opérateur* conditionnel

Exercice 2

Définissez une fonction `ordered` qui s'applique à 3 entiers et qui renvoie en résultat un booléen qui indique si les 3 valeurs fournies sont triées par ordre croissant :

- version 1 : en utilisant l'*instruction* conditionnelle
- version 2 : en utilisant l'*opérateur* conditionnel
- version 3 : en n'utilisant aucune forme de conditionnelle

Remarque : la version 3 est toujours à privilégier.

Exercice 3

Définissez une fonction `entier` qui s'applique à un flottant de type `double` et qui renvoie en résultat un booléen qui indique si la valeur fournie correspond à une valeur entière :

- version 1 : sans tenir compte de l'approximation des flottants
- version 2 : en tenant compte d'une marge d'erreur de l'ordre d'un millionième

Par exemple, les valeurs flottantes `5.00001` et `5.0000001` ne sont pas considérées comme des valeurs entières dans la version 1. Par contre dans la version 2, la valeur `5.0000001` est considérée comme une valeur entière car ce qui la sépare de `5.0` est inférieur à un millionième.

Exercice 4

Définissez une fonction `fact` qui s'applique à un entier de type `int` et qui renvoie en résultat la factorielle de la valeur fournie (type `int` aussi) :

- version 1 : en utilisant la récursivité
- version 2 : en utilisant une itération
- version 3 : en n'utilisant ni récursivité, ni itération

Si la valeur fournie est négative, le résultat est -1. Si la valeur fournie est supérieure à 12, le résultat excède la capacité de `int`. Dans ce cas, le résultat est -2.

Exercice 5

Définissez une fonction `maxtab` qui s'applique à un tableau d'entiers non vide et qui renvoie en résultat la plus grande valeur du tableau.

Exercice 6

Définissez une fonction `inverse` qui s'applique à un tableau d'entiers et qui inverse l'ordre de ses éléments. Illustration :

```
int[] tab = { 5, -3, 8, 7, 4, -2, 4, 0 };
inverse(tab); // tab : { 0, 4, -2, 4, 7, 8, -3, 5 }
```

Attention : vous ne devez pas créer de tableau pour réaliser cette fonction.

Exercice 7

Définissez une fonction `cptab` qui s'applique à un tableau d'entiers et qui renvoie en résultat une copie du tableau fourni :

- version 1 : en utilisant une boucle de parcours
- version 2 : en utilisant la méthode `arraycopy` de la classe `System`
- version 3 : en utilisant la méthode `copyOf` de la classe `Arrays`
- version 4 : en utilisant la méthode `clone` des tableaux

Exercice 8

Définissez une fonction `cptab2dim` qui s'applique à un tableau à deux dimensions d'entiers et qui renvoie en résultat une copie *en profondeur* du tableau :

- version 1 : en utilisant une double boucle de parcours
- version 2 : en utilisant une simple boucle de parcours et en réutilisant la fonction `cptab` de l'exercice précédent

Exercice 9

Définissez une fonction `extrairePairs` qui s'applique à un tableau à deux dimensions d'entiers `tab` et qui renvoie en résultat un tableau à une dimension comportant exactement les nombres pairs de `tab`. Illustration :

```
int[] t1 = extrairePairs(new int[][]{{5,-3,8,7},{4,-2,4,0}});
           // t1 : [8,4,-2,4,0]
int[] t2 = extrairePairs(new int[][]{{5,-3,8,7},{9,-3,5,1}});
           // t2 : [8]
int[] t3 = extrairePairs(new int[][]{{5,-3,9,7},{9,-3,5,1}});
           // t3 : []
```

Exercice 10

Définissez une fonction contient qui s'applique à deux chaînes de caractères et qui indique si la deuxième est un extrait de la première, indépendamment de la casse. Exemple :

```
contient("Brocoli", "Brocolis") // faux
contient("Brocoli", "Brocoli")  // vrai
contient("Brocoli", "bro")      // vrai
contient("Brocoli", "COL")      // vrai
contient("Brocoli", "CLO")      // faux
```

Rappel des méthodes de String utiles pour cet exercice :

```
boolean equalsIgnoreCase()     // equals indépendant de la casse
int length()                   // taille de la chaîne
String substring(int d, int f) // extrait de d (inclus) à f (exclu)
```

Exercice 11

Définissez une fonction separe qui s'applique à une chaîne de caractères et qui renvoie une chaîne constituée d'abord des voyelles puis de tous les autres caractères sauf des espaces. Toutes les lettres doivent apparaître en majuscules dans le résultat. Exemple :

```
separe("Allez␣les␣bleus␣!") // "AEEEULLZLSBLS!"
```

Rappel des méthodes de String utiles pour cet exercice :

```
String toUpperCase() // mise en majuscules
int length()         // taille de la chaîne
char charAt(int i)   // caractère à l'indice i
```

Exercice 12

Un palindrome est un mot que l'on peut lire indifféremment de gauche à droite ou de droite à gauche, comme par exemple *ressasser*. Dans un premier temps, définissez une fonction palindrome qui s'applique à une chaîne de caractères et qui indique si cette chaîne correspond à un palindrome.

Un palindrome n'est pas nécessairement réduit à un mot. Il peut s'agir d'une phrase ou même un texte complet. Dans ce cas, on ignore les espaces, les apostrophes et la ponctuation, et on ne tient pas compte de la casse, des accents, des trémas et autres cédilles. Exemples :

- *Engage le jeu que je le gagne.*
- *Oh, cela te perd répéta l'écho !*

Adaptez la fonction `palindrome` afin qu'elle puisse reconnaître les palindromes ainsi définis. Pour cette version étendue, vous pouvez utiliser la méthode `replaceAll` de `String` :

```
// Remplace les sous-chaînes correspondant à e par r :
String replaceAll(String e, String r) // e : expression régulière
```

```
// Exemple :
"AbCdEf".replaceAll("[ABCDEF]", "--") // "--b--d--f"
```

Exercice 13

Complétez la méthode genTab1 suivante qui crée un tableau d'entiers de n cases contenant des valeurs aléatoires comprises entre 0 et 100 :

```
public static int[] genTab1(int nbCases) { /* ... */ }
```

Exemple d'utilisation :

```
int[] tab1Test = genTab1(10); // génération d'un tableau de 10 cases
System.out.println(Arrays.toString(tab1Test)); // affichage
```

Exemple d'affichage obtenu à l'exécution (les nombres varient à chaque exécution) :

```
[44, 42, 90, 65, 86, 19, 7, 69, 99, 94]
```

Pour définir les valeurs aléatoires, vous devez utiliser la méthode statique `random` de la classe `Math` qui fournit une valeur flottante comprise entre 0 et 1.

Exercice 14

Complétez la méthode genTab2 suivante qui fournit un tableau à deux dimensions d'entiers de nbl lignes et nbc colonnes, et dont chaque case contient des valeurs aléatoires comprises entre 0 et 100 :

```
public static int[][] genTab2(int nbl, int nbc) { /* ... */ }
```

Exemple d'utilisation :

```
int[][] tab2Test = genTab2(3,5); // tableau de 3x5 cases
System.out.println(Arrays.deepToString(tab2Test)); // affichage
```

Exemple d'affichage obtenu à l'exécution (les nombres varient à chaque exécution) :

```
[[80, 94, 72, 78, 95], [59, 50, 97, 64, 80], [34, 97, 41, 66, 95]]
```

Pour programmer cette méthode, vous devez utilisez la méthode genTab1 de l'exercice précédent.

Exercice 15

Complétez la méthode swap suivante qui échange le contenu des cases d'indices i et i+1 d'un tableau fourni en paramètre :

```
public static void swap(int[] tab, int i) { /* ... */ }
```

Exemple d'utilisation :

```
int[] tab = {2, 7, 0, 5, 3, 1, 9};
System.out.println(Arrays.toString(tab)); // affichage avant
swap(tab, 2); // échange des cases d'indice 2 et 3
System.out.println(Arrays.toString(tab)); // affichage après
```

À l'exécution, on obtient l'affichage suivant (les valeurs 0 et 5 sont échangées) :

```
[2, 7, 0, 5, 3, 1, 9]
[2, 7, 5, 0, 3, 1, 9]
```

Exercice 16

Complétez la méthode bulle suivante qui s'applique à un tableau à une dimension d'entiers et à un entier n. Cette méthode déplace dans la case d'indice n la plus grande valeur contenue dans le tableau entre les indices 0 et n (inclus) :

```
public static void bulle(int[] tab, int n) { /* ... */ }
```

Exemple d'utilisation :

```
int[] tabBulle = {2, 7, 0, 5, 3, 1, 9};
System.out.println(Arrays.toString(tabBulle)); // affichage avant
bulle(tabBulle, 4); // maximum de [0 ; 4] placée à l'indice 4
System.out.println(Arrays.toString(tabBulle)); // affichage après
```

À l'exécution, on obtient l'affichage suivant. La valeur 7 est placée dans la case d'indice 4. Il s'agit bien de la plus grande valeur parmi les cases indicées de 0 à 4 :

```
[2, 7, 0, 5, 3, 1, 9]
[2, 0, 5, 3, 7, 1, 9]
```

Pour programmer cette méthode, vous devez utilisez la méthode swap de l'exercice précédent.