

Programmation Orientée Objets avec Java

Stéphane Malandain, Yassin Rekik



Java™

Chapitre 1

Les bases du langage Java

Les bases du Java

2

- Généralités
 - Histoire
 - Caractéristiques
- Sémantique
 - Paradigmes
 - Promotion numérique
 - Conversion
- Syntaxe
 - Types Primitifs
 - Opérations
 - Structures de contrôle
 - Structures de données
 - Entrées / Sorties
 - Exceptions
 - Méthodes

- Simula 67, Smalltalk (1970), Eiffel (1986)
 - 1996 – 2000 : Java 1
 - 2000 – 2008 : Java 1.4
 - 2002 – 2015 : Java 5
 - 2005 – 2018 : Java 6
 - 2011 – 2022 : Java 7
 - 2014 – 2019 : Java 9
 - 2018 - ? : Java 10
 - 2018 – 2026 : Java 11 (LTS)
 - Mars 2019 : Java 12
 - Mars 2020 : Java 14
 - Mars 2021 : Java 16
 - Mars 2022 : Java 18
 - Sept 2022 : Java 19
 - Mars 2023 : Java 20
- | | |
|-----------|-----------|
| Juin 2024 | : Java 22 |
| Sept 2024 | : Java 23 |

Caractéristiques

- Populaire
- Industriel
- Large écosystème
- Très grande communauté
- Portable et interprété
 - « Code once, run everywhere »
- Simple
 - Pas de pointeurs
 - Mécanisme de ramasse-miettes
 - Pas de gestion de la mémoire
 - Gestion des exceptions
- Langage statique compilé

HelloWorld Java

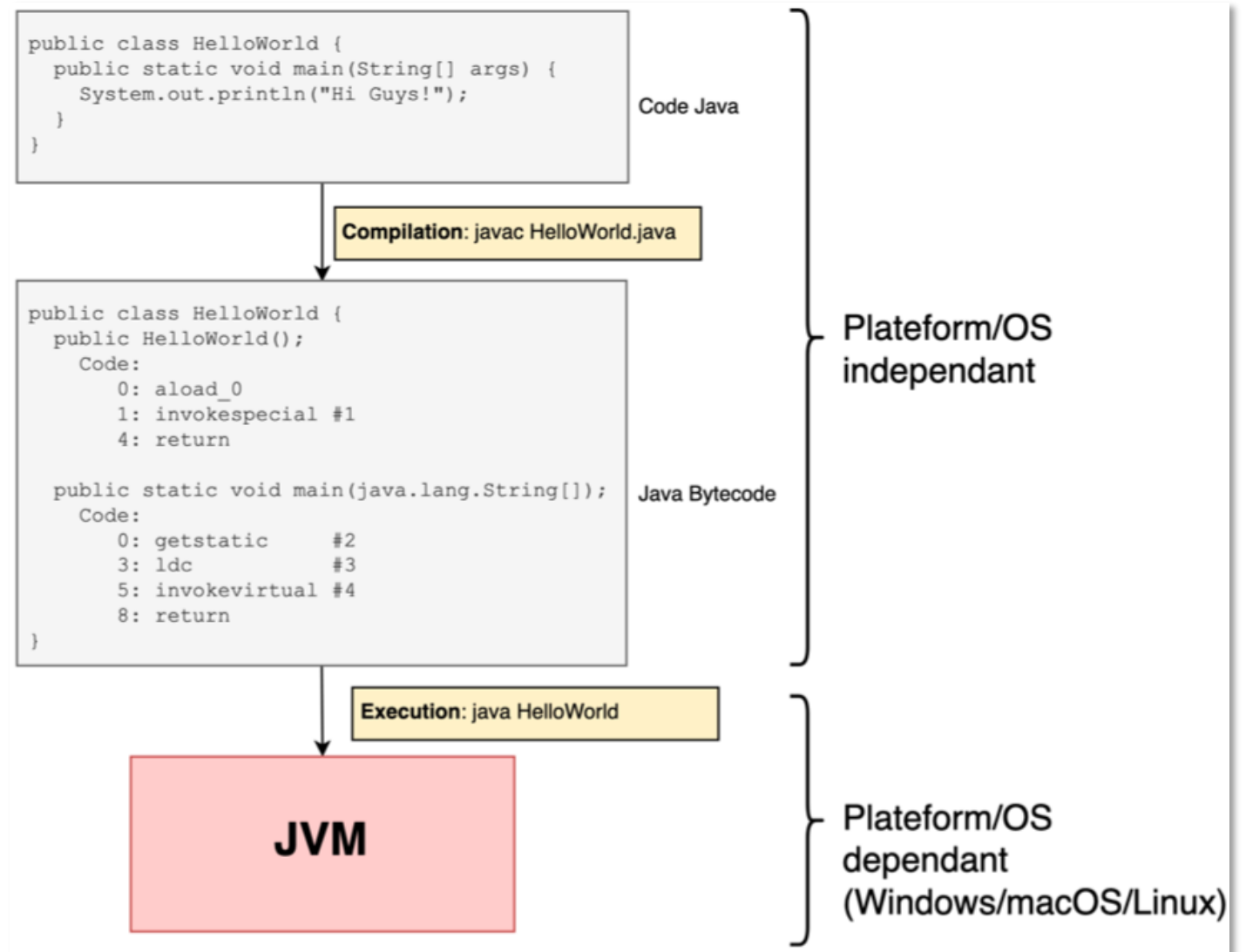
- HelloWorld.java

```
1  public class HelloWorld {  
2      public static void main(String[] args) {  
3          System.out.println("Hello !");  
4      }  
5  }
```

Compilation / Exécution

6

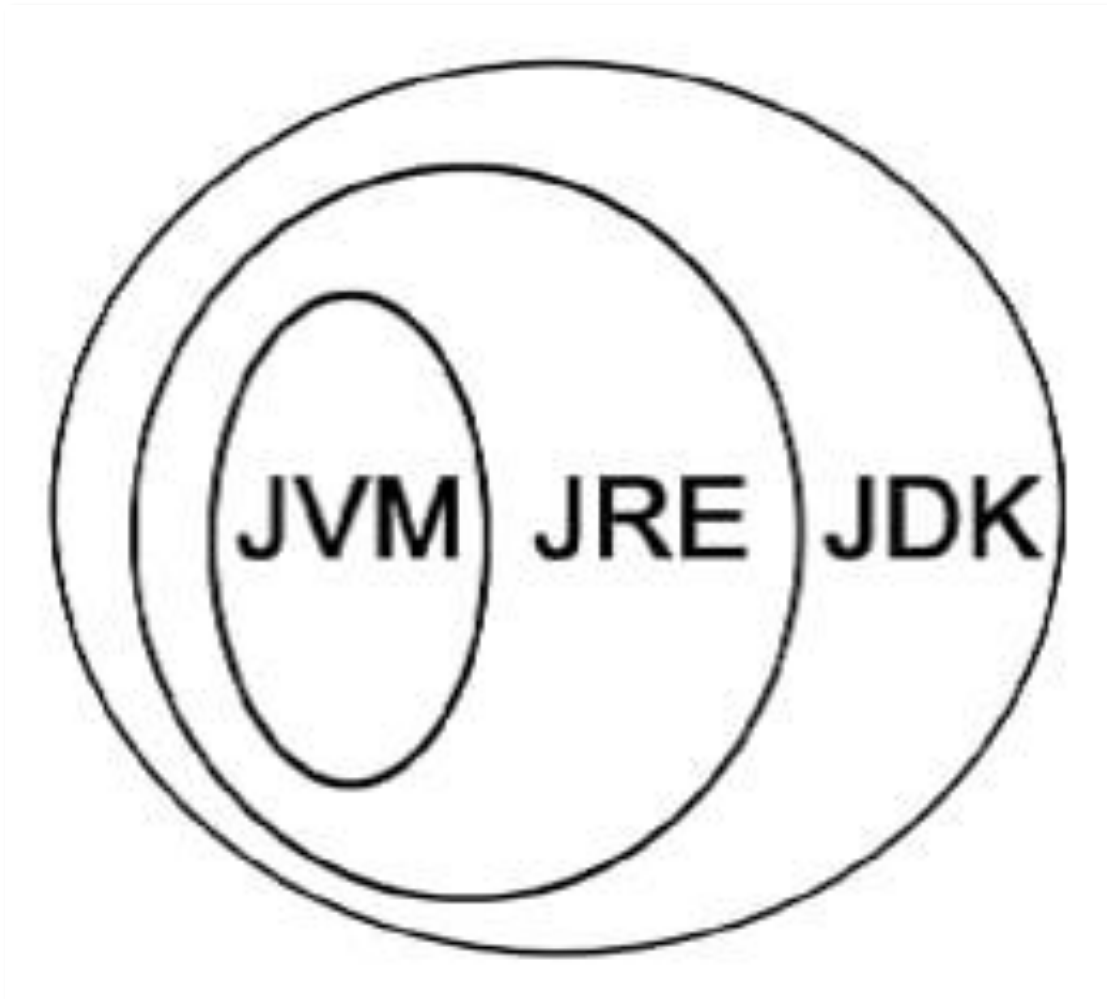
- Compilation / exécution



Compilation / Exécution

7

- Figure 2 : Ecosystème JAVA



JRE = JVM + Library Class

JDK = JRE + Development Tools

- Java est un langage compilé
- Le bytecode est un langage interprété par la JVM
 - Transformation du code source en code machine ligne après ligne
 - Exécution lente !
- Le compilateur Just-In-Time (JIT) est un module du JRE
 - Transforme lors de l'exécution le bytecode fréquemment utilisé en code machine
 - Exécution rapide !
- Le compilateur Ahead-Of-Time (AOT, jaotc du JDC) permet de transformer du bytecode en code machine
 - Compilation pré-exécution
 - Exécution rapide
- HotSpot d'Oracle est la JVM la plus courante

Paradigme

- Impératif
- Orienté Objet (avec types primitifs)
- Style déclaratif (depuis la version 7)

- Algorithme : Compter la somme des éléments positifs d'une liste

Style **impératif** en pseudo-java

```
1  int i = 0;
2  int result = 0;
3  while (i < liste.size() ) {
4      int valeur = liste[i];
5      if (valeur >= 0) {
6          result += valeur;
7      }
8      i += 1
9  }
10 print ( result )
11
```

- Algorithme : Compter la somme des éléments positifs d'une liste

Style **déclaratif** en pseudo-java

```
1 int result = liste.filter( (valeur) -> valeur >= 0 ).sum()  
2 print(result)
```

La syntaxe

Documentation Oracle de Java

13

<https://docs.oracle.com/en/java/javase/23/>

- Vérification des types à la compilation
 - Les variables sont précédées de leur type

```
1  int i=12 ;  
2  String s = "Salut !" ;  
3  float f = 56.2 ;
```

- Depuis Java 10, inférence des types avec `var`

```
1  var i=12 ;  
2  var s = "Salut !" ;  
3  var f = 56.2 ;
```

La console jshell (Java 9 ->)

15

- Appelé aussi REPL : Read, Eval, Process and Loop.

```
[malandai@StephBook ~ % jshell
| Welcome to JShell -- Version 18.0.2
| For an introduction type: /help intro

[jshell> String firstname="steph"
firstname ==> "steph"

[jshell> String lastname="malandain"
lastname ==> "malandain"

[jshell> System.out.p
print(      printf(      println(
[jshell> System.out.println(firstname+ " " + lastname);
steph malandain

jshell> █
```

Structure générale d'un programme

16

```
1  /*
2   * Programme HelloWorld
3   * Le fichier doit s'appeler HelloWorld.java
4   *
5   * Compilation :    javac HelloWorld.java
6   * Exécution :     java HelloWorld
7   */
8  public class HelloWorld {
9      public static void main(String[] args) {
10         System.out.println("Hello !");
11     }
12 }
13
```


Explications

17

- `Public class HelloWorld` : le programme principal est une classe à visibilité publique
- Méthode `main` :
 - Code du programme principal
 - `static` : n est pas lié à un objet
 - `void` : ne retourne rien
 - `String[] args` : permet de recevoir des arguments

Commentaires

18

```
1 // Commentaires simples
2
3 /* Commentaires
4  * multi-lignes */
5
6 /*
7  * Commentaires importants (par convention)
8  */
9
10 /**
11  * Commentaires pour la génération de la javadoc
12  */
13
```

Les types primitifs

19

- Un type primitif commence par une minuscule
- Chaque type primitif à son équivalent en classe (Byte, Integer, Double, ...)
- Un type primitif est stocké sur la pile (stack),
 - il est donc plus performant qu'une classe stockée sur le tas (heap)
- Préférez toujours un type primitif !

Nombres entiers

`byte, short, int, long`

Nombres réels

`float, double`

Caractère

`char`

Booléen

`boolean`

Les types primitifs

20

- Les types primitifs

Category	Types	Size (bits)	Minimum Value	Maximum Value	Precision	Example
Integer	byte	8	-128	127	From +127 to -128	byte b = 65;
	char	16	0	$2^{16}-1$	All Unicode characters	char c = 'A'; char c = 65;
	short	16	-2^{15}	$2^{15}-1$	From +32,767 to -32,768	short s = 65;
	int	32	-2^{31}	$2^{31}-1$	From +2,147,483,647 to -2,147,483,648	int i = 65;
	long	64	-2^{63}	$2^{63}-1$	From +9,223,372,036,854,775,807 to -9,223,372,036,854,775,808	long l = 65L;
Floating-point	float	32	2^{-149}	$(2-2^{-23}) \cdot 2^{127}$	From 3.402,823,5 E+38 to 1.4 E-45	float f = 65f;
	double	64	2^{-1074}	$(2-2^{-52}) \cdot 2^{1023}$	From 1.797,693,134,862,315,7 E+308 to 4.9 E-324	double d = 65.55;
Other	boolean	--	--	--	false, true	boolean b = true;

Déclaration de variables

21

- Exemple avec le int

```
1  int i1;                // attribution de la valeur 0 par défaut
2  int i2 = 6;
3  int i3 = 3 * 6;        // attribution d'une expression
4  int i4 = 20, i5 = 10;  // attribution multiple
5  int i6 = 12, i7, i8 = 5; // attention, ici i7 vaut 0
```

Seules les lignes 2 et 3 sont recommandées

Déclaration de variables

22

- Non attribution de valeur
- Une valeur arbitraire "neutre" est attribuée par défaut si aucune valeur n'est affectée explicitement à une déclaration de variable
 - 0 pour les nombres entiers
 - 0.0 pour les nombres réels
 - false pour les booléens
 - ' ' pour les caractères
 - null pour les objets

Les types entiers

23

type	octets
byte	1
short	2
int	4
long	8

- Arithmétique modulaire : aucun contrôle des débordements
- L'ensemble des valeurs forme un cycle

```
[jshell> int j = 2147483647 + 1  
j ==> -2147483648
```

Les types entiers

24

- Quelques règles :
 - Littéral (valeur donnée explicitement dans le code) entier de type int
 - Le compilateur refuse tout littéral entier dont la valeur n'appartient pas au domaine de définition du type de la variable

```
1  int i1 = 14;    // 14 est un littéral
2  byte b1 = 127;  // Ok, 127 est converti en byte;
3  byte b2 = 128;  // Erreur, 128 est hors du domaine
4  |   |   |   |  // de définition d'un byte
5  short s1 = 10;
6  short s2 = 20;
```


Les types entiers

25

- Conversion en long : post-fixer le littéral de l ou L

```
1  long l1 = 3000000000; // erreur, 3 milliards n'est pas un int
2  long l2 = 3000000000L; // ok
```

Les types entiers

26

- Promotion numérique :
 - La valeur d'une expression entière est de type int
 - Dans une expression, les opérandes de type byte ou short sont converties en int.

```
1  short s1 = 10;
2  short s2 = 20;
3  short s3 = s1 + s2; // Erreur, le résultat s1+s2 retourne un entier.
4  |         |         |         |         // il est impossible de l'attribuer à un short
5  short s4 = s1; // ok, la valeur d'affectation n'est pas une expression
6
```

Les types réels

27

type	octets
float	4
double	8

- Quelques règles :
 - Littéral réel est de type double
 - Le compilateur refuse de l'attribuer à un float
 - Conversion d'un double en float : post-fixer le littéral de f ou F
 - Pas de promotion numérique : une opération sur deux floats retourne un float

Les types réels

28

```
1  float f1 = 10.0;    // erreur, 10.0 est un double !
2  |         |         |         |         // il ne peut donc pas être attribué à un float
3  float f2 = 10.0f;    // ok
4  double f3 = 2e3;     // Notation scientifique =  $2 \times 10^3 = 2000.0$ 
5  float f4 = f2 + f2;  // OK, pas de promotion numérique
```

Le type caractère

29

- char occupe 2 octets pour stocker un caractère
- représentation Unicode
- guillemets simples
- interprété comme un entier non signé
- il respecte la promotion numérique

```
1  char c1;           // initialisé à chaîne vide ''
2  char c2 = 'B';
3
4  int i = 12;
5  char c3 = i;        // erreur, conversion implicite int -> char impossible!
6
7  char c4 = 12;        // OK, le compilateur comprend que 12 est dans l'ensemble
8  |   |   |           // de définition d'un char
```

Le type booléen

30

- deux valeurs possibles : `true` et `false`
- occupe un octet
- pas de conversion implicite d'un autre type vers un boolean

```
1    boolean b1 = 2;        // erreur
2    boolean b2;            // b2 vaut false
3    boolean b3 = true;
4    boolean b4 = 3 > 4;    // b4 vaut false
```

Compatibilité et équivalence

31

- types non équivalents entre eux
 - espace mémoire diffère
 - signé / non signé
- mais ils sont comparables voir compatibles
 - conversion implicite possible si aucune perte de précision
 - si la conversion implique une perte possible de précision :
 - => conversion explicite (`cast`) au risque de perdre de l'information

Conversions implicites

32

- Conversion implicites légales
 - une conversion implicite est légale s'il n'y a pas de perte de précision (conversion non dégradante)
 - conversion possible selon la hiérarchie suivante :
- `byte -> short -> int -> long -> float -> double`
- `char -> int -> long -> float -> double`
 - Les conversions `byte -> char` ou `short -> char` sont impossibles

Conversions implicites

33

- Conversions implicites légales

```
1  short s = 3;
2  float f = s; // ok, conversion short -> float
3
4  void h(int i, float f) { ... }
5
6  short s; float f;
7  h(s,s); // ok, arg1 -> int, arg2 -> float
8  h(f,f); // erreur, f -> int impossible
9  h(s, 10.0) // erreur, 10.0 est de type double
10
```

Conversions implicites

34

- Deux types de conversions implicites dans les expressions :
 - Promotion numérique sur les opérateurs
 - Byte, short, char convertis en int
 - Ajustement de types
 - Si $T1 \subset T2$ alors T1 peut être converti en T2
 - ex. int vers long
 - Hiérarchie des conversions
 - int -> long -> float -> double
 - ex. long vers float est possible, mais pas l'inverse

Conversions implicites

35

(short + short) + double

|

|

(int + int) + double

int + double

|

double + double

double

// promotion numérique short -> int

// ajust. int -> double

// résultat

Conversions explicites (cast)

36

- Exemple de conversion explicite

```
1  short s1 = 32767; // ok, 32767 => valeur max d'un short
2  short s2 = 32768; // erreur, 32768 hors de l'ensemble de
3  |         |         |         | // définition des short
4  short s3 = (short)32768; // conversion explicite. un cycle
5  |         |         |         | // s'effectue lors d'un dépassement.
6  |         |         |         | // s3 vaut -32768
```

Conversions

37

- Quizz : Quelles sont les valeurs de n et p ?

```
1    byte b = Byte.MAX_VALUE;  
2    int n = b + 1;  
3  
4    int i = Integer.MAX_VALUE;  
5    long p = i + 1;  
6
```

Déclaration d'une constante

38

- Une bonne pratique veut que toute référence non modifiée doit être une constante.
 - mot-clé : final
 - si visibilité globale : majuscule
 - si locale : convention des majuscules abandonnée (pour des raisons de lisibilité)

```
1    final double PI = 3.14159
2    final var userId = 1;
```

Convention de nommage

39

- identifiants de variables et fonctions en notation camelCase
 - `value`, `globalTax`, `veryBigVariableName`
- identifiants de fonctions / méthodes
 - par convention, commence par un verbe (suivi par adjectif, nom, ...)
 - `Run`, `runFast`, `isNumeric`, `hasValue`, `setXXX`, `getXXX`, ...
- idem pour le nom des classes, mais commence par une lettre majuscule
 - `String`, `HelloWorld`, `System`, `MyClass`
- utilisation impossible
 - des mots réservés du langage,
 - des littéraux (`true`, `false`, `null`),
 - du nom des types primitifs et des classes.

Convention de nommage

40

- Mots-clés réservés du Java

<code>abstract</code>	<code>continue</code>	<code>for</code>	<code>new</code>	<code>switch</code>
<code>assert</code>	<code>default</code>	<code>goto</code>	<code>package</code>	<code>synchronized</code>
<code>boolean</code>	<code>do</code>	<code>if</code>	<code>private</code>	<code>this</code>
<code>break</code>	<code>double</code>	<code>implements</code>	<code>protected</code>	<code>throw</code>
<code>byte</code>	<code>else</code>	<code>import</code>	<code>public</code>	<code>throws</code>
<code>case</code>	<code>enum</code>	<code>instanceof</code>	<code>return</code>	<code>transient</code>
<code>catch</code>	<code>extends</code>	<code>int</code>	<code>short</code>	<code>try</code>
<code>char</code>	<code>final</code>	<code>interface</code>	<code>static</code>	<code>void</code>
<code>class</code>	<code>finally</code>	<code>long</code>	<code>strictfp</code>	<code>volatile</code>
<code>const</code>	<code>float</code>	<code>native</code>	<code>super</code>	<code>while</code>

Convention de nommage

41

- Règle d'or
 - plus la visibilité d'une variable est grande, plus son nom doit être explicite.
 - il vaut mieux un identifiant long et explicite que des commentaires inutiles.

Classes enveloppes

42

- Chaque type primitif à son équivalent en terme de classe

type primitif	Classe associée (enveloppe)
boolean	Boolean
byte	Byte
short	Short
char	Character
int	Integer
long	Long
float	Float
double	Double

préférez toujours les types primitifs !

Classes enveloppes

43

- Les paramètres d'un type générique ne peuvent pas être un type primitif.
 - (List<Integer> et non List<int>)
- autoboxing et unboxing
 - autoboxing : conversion automatique d'un type primitif en un objet
 - unboxing : opération inverse

```
1 // Autoboxing
2 Integer i = 3; // au lieu de Integer i = new Integer(3);
3
4 // Unboxing
5 int j = i;
```

- Attention :

`void` et `null` ne sont pas des types !

- `void` signifie qu'une fonction ne retourne pas de valeur
- `null` signifie qu'un objet n'est pas instancié

Opérations

Opérateurs et précédenances

46

- Précédence des opérateurs

Operator Precedence	
Operators	Precedence
postfix	<i>expr</i> ++ <i>expr</i> --
unary	++ <i>expr</i> -- <i>expr</i> + <i>expr</i> - <i>expr</i> ~ !
multiplicative	* / %
additive	+ -
shift	<< >> >>>
relational	< > <= >= instanceof
equality	== !=
bitwise AND	&
bitwise exclusive OR	^
bitwise inclusive OR	
logical AND	&&
logical OR	
ternary	? :
assignment	= += -= *= /= %= &= ^= = <<= >>= >>>=

Opérateurs et précédences

47

```
1  int i = 1;
2
3  i++ * 3 + 2 * 2;
4  1 * 3 + 4 // i++ retourne 1 mais i vaut maintenant 2
5  3+4
6  7 // résultat de l'expression
```

Structures de contrôle

Branchement conditionnel

49

- Syntaxe générale du **if**

```
1  if (condition) {  
2  |    ...  
3  } else if (condition2) {  
4  |    ...  
5  } else if (condition3) {  
6  |    ...  
7  } else {  
8  |    ...  
9  }
```

Branchement conditionnel

50

- Syntaxe générale du **switch**

```
1  switch (expression) {  
2      case constant1:  
3          ...  
4          break;  
5      case constant2:  
6          ...  
7          break;  
8      case constant3:  
9          ...  
10         break;  
11         default:  
12             ...  
13     }
```

Branchement conditionnel

51

- Exemple

```
1  int i = 2;
2  switch(i) {
3      case 1:
4          System.out.println("Un");
5      case 2:
6          System.out.println("Deux");
7      case 3:
8          System.out.println("Trois");
9      default:
10         System.out.println("...");
11 }
```

Affichage

Deux

Trois

...

Branchement conditionnel

52

- Exemple

```
1  int i = 2;
2  switch(i) {
3      case 1:
4          System.out.println("Un");
5          break;
6      case 2:
7          System.out.println("Deux");
8          break;
9      case 3:
10         System.out.println("Trois");
11         break;
12     default:
13         System.out.println("...");
14 }
```

Affichage

Deux

Expression conditionnelle

53

- Syntaxe générale :
 - `valeur = condition ? valeur_si_vrai : valeur_si_faux;`

```
1  // Instruction
2  String type;
3  if (age >= 18) {
4      type = "Majeur";
5  } else {
6      type = "Mineur";
7  }
8
9  // Expression
10 String type = age >= 18 ? "Majeur" : "Mineur";
```

Boucle pré-conditionnelle

54

- Répétition d'instruction tant qu'une condition est vérifiée a priori

```
while (condition) {  
    ...  
}
```

```
1  int i = ???  
2  while (i<4) {  
3      System.out.println("i: %d",i);  
4      i += 1;  
5  }
```

i	Affichage
2	i:2
	i:3
3	i:3
4	
5	

Boucle post-conditionnelle

55

- Répétition d'instruction tant qu'une condition est vérifiée à posteriori
 - Le bloc est exécuté au moins une fois

```
do{  
    ...  
} while (condition);
```

```
1  int i = ???  
2  do {  
3      System.out.println("i: %d",i);  
4      i += 1;  
5  } while (i < 4);
```

i	Affichage
2	i:2
	i:3
3	i:3
4	i:4
5	i:5

Boucle d'itération

56

```
for (initialisation; condition; itération) {  
    ...  
}
```

```
1  String[] greetings = {"Bonjour", "Coucou", "Salut"};  
2  
3  // Affiche les 3 chaînes de caractères du tableau  
4  for ( int i = 0; i < greetings.length; i+=1 ) {  
5      System.out.println(greetings[i] );  
6  }
```


Boucle de parcours

57

- garantie du parcours de tous les éléments
 - la plus élégante et la plus sûre

```
for (type identifiant: collection) {  
    // identifiant va prendre successivement toutes  
    // les valeurs de la collection  
}
```

```
1  String[] greetings = {"Bonjour", "Coucou", "Salut"};  
2  
3  for ( String hello: greetings ) {  
4      System.out.println( hello );  
5  }
```

Boucle de parcours

58

- Style déclaratif

```
1  String[] greetings = {"Bonjour", "Coucou", "Salut"};
2
3  // Style déclaratif:
4  Arrays.asList(greetings).forEach( hello -> System.out.println(hello) );
5
6  // ou aussi
7  Arrays.asList(greetings).forEach( System.out::println );
8
```

Structures de données

Chaînes de caractères (String)

60

- objets immuables (ou immutables)
- il est possible de changer sa référence (si non final)
- objet immutable : une fois instancié, l'état de l'objet ne peut pas changer

```
1    // String str = new String("abc"); inutile
2    String str = "abc"; // autoboxing
3    str = str + str; // str vaut maintenant "abcabc"
```

Chaînes de caractères (String)

61

Exemples de méthodes d'instance

```
1 String test = "Abcdef";  
2 test.contains("bc"); // retourne true  
3 test.contains("ab"); // retourne false  
4 test.concat("gh");   // retourne Abcdefgh mais ne modifie  
5 | | | | |           // pas l'instance test  
6 test.toUpperCase()   // retourne "ABCDEF"  
7 test.split("c")       // retourne { "Ab", "def" }
```

Exemples de méthodes de classe

```
1 String.join(" - ", "This", "course", "is", "awesome");  
2 // retourne "This - course - is - awesome"  
3  
4 String.valueOf(22); // retourne "22"
```

- Les tableaux sont des objets de la classe `Arrays`
 - il est **impossible** de supprimer ou de rajouter des éléments
 - il est possible de modifier le contenu par contre.

Syntaxe

- Notation crochet `[]` pour la déclaration de tableau
- Attribution:
 - `{ valeur1, valeur2, ... }`
 - `New Type[TAILLE]`

```
1  char[] tab1 = {a,b,c};  
2  double[] tab2 = new double[20];
```

- Instanciation de tableaux
 - prends la valeur `null` si non affecté
 - prends un ensemble de valeurs "neutres" du type s'il est instancié avec une taille

```
1  int[] is; // is = null
2  int[] empty = {}; // tableau vide
3  int[] fibonacci = {1,1,2,3,5,8,13};
.
```

- Instanciation de tableaux avec taille arbitraire

```
1  float[] fs = new float[4]; // fs = {0.0, 0.0, 0.0, 0.0}
2  Float[] fs2 = new Float[4]; // fs2 = {null, null, null, null}
3  Person[] persons = new Person[10]; // tableau de 10 valeurs null
```

- Plusieurs dimensions possibles

```
1 double[][] matrix = new double[2][3]; // tableau à 2 dim.
```

- Copie / référencement

```
1 double[][] matrix2 = matrix; // matrix et matrix2 pointent
2 

|  |  |  |  |  |  |  |
|--|--|--|--|--|--|--|
|  |  |  |  |  |  |  |
|--|--|--|--|--|--|--|

 // sur la même instance
3 double[][] matrix3 = matrix.clone(); // copie le tableau
```

- Un unique attribut `length` pour connaître le nombre d'éléments

```
1 int[] is = new int[12];
2 is.length; // retourne 12
```


Communication homme-machine

Affichage (terminal)

66

- La classe `System` met à disposition le flux de sortie `out` et fournit plusieurs méthodes
 - `Print` : affichage sans retour à la ligne
 - `Println` : avec retour à la ligne
 - `format` : printf C-style
 - beaucoup d'autres à découvrir sur doc.oracle.com: System

```
1 System.out.print("Affiche l'argument sans retour à la ligne");
2 System.out.println("Affiche l'argument avec retour à la ligne");
3 System.out.println("Concaténation: "+ 10 + "!");
4 System.out.format("Valeur de l'argument: %d",2);
```

Lecture (terminal)

67

- la lecture d'information via le terminal se fait à l'aide d'un objet de type `Scanner`
- flux d'entrée `System.in` en argument

```
1  import java.util.Scanner;
2
3  Scanner scanner = new Scanner(System.in);
4
5  System.out.print("Enter your name: ");
6  String name = scanner.next();
7
8  System.out.print("Enter your age: ");
9  int age = scanner.nextInt();
10
11 scanner.close();
12
13 System.out.println("Bonjour " + name + ", vous avez " + age + " ans");
14
```

Méthodes statiques

- une méthode est une fonction rattachée à un type
- elles sont liées à la notion de classes que nous aborderons prochainement
- les méthodes statiques sont utilisables cependant comme des fonctions

```
1  // fichier Math.java
2
3  public class Math {
4      public static int doubleThat(int i) { return i*2; }
5      public static int square(int i) { return i*i; }
6
7      public static void main(String[] args) {
8          System.out.println( square(10) ); // Affiche 100
9      }
10 }
```

Exception (base)

70

- Traiter une exception

```
1  try {  
2      int i = scanner.nextInt();  
3      System.out.println(i);  
4  } catch (InputMismatchException e) {  
5      System.err.println(e.toString());  
6  } finally {  
7      scanner.close();  
8  }
```

Exception (base)

71

- Provoquer une exception

```
1  public static int countPeople() {  
2      if ( !isDatabaseConected() ) {  
3          throw new RuntimeException("Database out of order")  
4      }  
5      ...  
6  }  
7  }
```

- Quizz : Réalisez une méthode `askInt()` qui demande un entier à l'utilisateur et le retourne. Tant que l'utilisateur n'entre pas une valeur entière, la demande lui est refaite.
- 2 versions à réaliser : l'une avec une boucle, l'autre sans.
- Exemple d'utilisation :

```
Entrez une valeur: 24ads
Valeur erronée
Entrez une valeur: aued
Valeur erronée
Entrez une valeur: 54
La méthode a retourné 54
```


- Complétez le modèle ci-dessous

```
1  public class Lecture {  
2  
3      ...  
4  
5  
6      public static void main(String[] args) {  
7  
8          System.out.println("La méthode a retourné "+ askInt() );  
9  
10     }  
11 }
```