Java de base

Université Française d'Égypte – TIC303
Version 1.17 – 25/10/20
Richard Grin
http://richard.grin.free.fr
Richard.Grin@univ-cotedazur.fr

1

Cours POO - TIC303

- http://richard.grin.free.fr/ufe/cours/tic303
- 2 contrôles pendant le cours
  1 note de TP
  1 contrôle final en fin de semestre

2

Plan du cours

- Bases du langage Java
- Héritage et polymorphisme, interfaces
- Exceptions
- Collections
- Généricité
- Classes internes
- Expressions lambda et streams

3

Plan de cette partie

- Présentation de Java, 1er programme, compilation et exécution
- Notions de base sur la programmation objet
- Les classes (constructeurs, méthodes, variables, encapsulation, `this`, méthodes et variables de classe)
- Types de données (primitifs, objets, types énumérés, tableaux, classes de base)
- Instructions de contrôle (if, while, for)
- Compléments sur les méthodes

4

Principales propriétés de Java

- Orienté objet, à classes
- Fourni avec le JDK (*Java Development Kit*) :
  - outils de développement
  - ensemble très riche de paquetages
- Portable grâce à l'exécution par une machine virtuelle
- Sûr
  - fortement typé
  - nombreuses vérifications pendant l'exécution

5

Version étudiée

- Java 15, sorti en septembre 2020

6

Premier programme Java

7

Le code source du premier programme

Code dans une classe

point d'entrée d'exécution

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

- Classe `Helloworld` est `public`, donc le fichier qui la contient doit s'appeler `HelloWorld.java`

8

Portabilité de Java

Code source, écrit en Java — Java Code (.java) — `UneClasse.java`

JAVAC Compiler

Bytecode, langage de la JVM, machine virtuelle Java — byte Code (.class) — `UneClasse.class`

JVM JVM JVM

WINDOWS LINUX MAC

Bytecode `UneClasse.class` fonctionne sur n'importe quelle JVM

9

Compilation avec *javac*

- `javac HelloWorld.java`
  crée `HelloWorld.class` dans le même répertoire que `HelloWorld.java`
- On peut donner un chemin absolu ou relatif :
  `javac util/Liste.java`

Où sera placé le fichier `Liste.class` ?

10

Exécution avec *java*

- `java HelloWorld`
  exécute le *bytecode* de la méthode `main` de la classe `HelloWorld`
- `HelloWorld` est un nom de classe et pas un nom de fichier :
  - pas de chemin
  - pas de suffixe `.class`

11

Où doit se trouver le fichier `.class` ?

- `java HelloWorld`
  `HelloWorld.class` doit se trouver dans le *classpath* (une liste de répertoires, comme la variable PATH)
- Par défaut le répertoire courant
- *classpath* modifié par l'option `-classpath` :
  `java –classpath rep1/rep2 HelloWorld`

12

Compléments sur la compilation et l'exécution

13

Une classe `Point`

Commentaire javadoc

```
/** Modélise un point de coordonnées x, y */
public class Point {
  private int x, y;
  public Point(int x1, int y1) {    // constructeur
    x = x1;
    y = y1;
  }
  public double distance(Point p) { // méthode
    return Math.sqrt((x-p.x)*(x-p.x) + (y-p.y)*(y-p.y));
  }
  public static void main(String[] args) {
    Point p1 = new Point(1, 4);
    Point p2 = new Point(5, 1);
    System.out.println("Distance : " + p1.distance(p2));
  }
}
```

Fichier `?.java`

$\sqrt{(x-p.x)^2+(y-p.y)^2}$

`x` : x de `p1`
`p.x` : x de `p2`

Quelle valeur ?

Message envoyé à `p1` : « Calcule la distance qui te sépare de `p2` »

14

Une classe `Point`

```
/** Modélise un point de coordonnées x, y */
public class Point {
  private int x, y;
  public Point(int x1, int y1) {    // constructeur
    x = x1;
    y = y1;
  }
  public double distance(Point p) { // méthode
    return Math.sqrt((x-p.x)*(x-p.x) + (y-p.y)*(y-p.y));
  }
  public static void main(String[] args) {
    Point p1 = new Point(1, 4);
    Point p2 = new Point(5, 1);
    System.out.println("Distance : " + p1.distance(p2));
  }
}
```

Seulement pour tester

15

2 classes dans 1 fichier

```
/** Modélise un point de coordonnées x, y */
class Point {
  private int x, y;
  public Point(int x1, int y1) {
    x = x1; y = y1;
  }
  public double distance(Point p) {
    return Math.sqrt((x-p.x)*(x-p.x) + (y-p.y)*(y-p.y));
  }
}

/** Teste la classe Point */
class TestPoint {
  public static void main(String[] args) {
    Point p1 = new Point(1, 4);
    Point p2 = new Point(5, 1);
    System.out.println("Distance : " + p1.distance(p2));
  }
}
```

Quelle classe sera `public` ?

16

2 classes dans 1 fichier

```
/** Modélise un point de coordonnées x, y */
public class Point {
  private int x, y;
  public Point(int x1, int y1) {
    x = x1; y = y1;
  }
  public double distance(Point p) {
    return Math.sqrt((x-p.x)*(x-p.x) + (y-p.y)*(y-p.y));
  }
}

/** Teste la classe Point */
class TestPoint {
  public static void main(String[] args) {
    Point p1 = new Point(1, 4);
    Point p2 = new Point(5, 1);
    System.out.println("Distance : " + p1.distance(p2));
  }
}
```

Quel sera le nom du fichier ?

17

2 classes dans 1 fichier

```
/** Modélise un point de coordonnées x, y */
public class Point {
  private int x, y;
  public Point(int x1, int y1) {
    x = x1; y = y1;
  }
  public double distance(Point p) {
    return Math.sqrt((x-p.x)*(x-p.x) + (y-p.y)*(y-p.y));
  }
}

/** Teste la classe Point */
class TestPoint {
  public static void main(String[] args) {
    Point p1 = new Point(1, 4);
    Point p2 = new Point(5, 1);
    System.out.println("Distance : " + p1.distance(p2));
  }
}
```

Que génère la compilation `javac Point.java` ?

`Point.class` et `TestPoint.class`

18

2 classes dans 1 fichier

```
/** Modélise un point de coordonnées x, y */
public class Point {
  private int x, y;
  public Point(int x1, int y1) {
    x = x1; y = y1;
  }
  public double distance(Point p) {
    return Math.sqrt((x-p.x)*(x-p.x) + (y-p.y)*(y-p.y));
  }
}

/** Teste la classe Point */
class TestPoint {
  public static void main(String[] args) {
    Point p1 = new Point(1, 4);
    Point p2 = new Point(5, 1);
    System.out.println("Distance : " + p1.distance(p2));
  }
}
```

Il faut lancer l'exécution de quelle classe ?

Quelle commande taper ?

19

2 classes dans 2 fichiers

Fichier `Point.java`

```
/** Modélise un point de coordonnées x, y */
public class Point {
  private int x, y;
  public Point(int x1, int y1) {
    x = x1; y = y1;
  }
  public double distance(Point p) {
    return Math.sqrt((x-p.x)*(x-p.x) + (y-p.y)*(y-p.y));
  }
}
```

Fichier `TestPoint.java`

```
/** Tester la classe Point */
class TestPoint {
  public static void main(String[] args) {
    Point p1 = new Point(1, 4);
    Point p2 = new Point(5, 1);
    System.out.println("Distance : " + p1.distance(p2));
  }
}
```


20

Compilation des 2 fichiers (1/2)

- `javac Point.java TestPoint.java` (ou `javac *.java`)
- `javac TestPoint.java` suffit pour compiler les 2 classes car `TestPoint` *dépend* de `Point` :

```
class TestPoint {
  public static void main(String[] args) {
    Point p1 = new Point(1, 4);
    ...
```

Est-ce que `javac Point.java`, compile `TestPoint.java` ?

21

Compilation des 2 fichiers (2/2)

- A-t-on besoin de `Point.java` pour compiler `TestPoint.java` ?
- Non, mais on a besoin de `Point.class`

22

JShell

- Interpréteur d'instructions
- L'utilisateur tape des extraits de code qui sont immédiatement exécutés
- Pratique pour tester l'utilisation de classes

23

Exemple

```
C:\Users\toto> jshell
|  Welcome to JShell -- Version 15
|  For an introduction type: /help intro
jshell> String s = "abcdefg"
s ==> "abcdefg"
jshell> s.substring(1,3)
$2 ==> "bc"
jshell> /exit
|  Goodbye
C:\Users\toto>
```

`substring` extrait une sous-chaîne

Les caractères d'une `String` sont numérotés à partir de 0

Que signifient les 2 paramètres de la méthode `substring` ?

24

NOTIONS DE BASE SUR LA PROGRAMMATION OBJET

25

Langage orienté objet

- Manipule des objets
- Programmes découpés suivant les types des objets manipulés (classes)
- En C, programmes découpés suivant les fonctions

26

Les classes

- Les données sont regroupées avec les traitements qui les utilisent
- Une classe **`Facture`** regroupe
  - tout ce qu'on peut faire avec une facture
  - toutes les données nécessaires à ces traitements (client, date,…)

27

Qu'est-ce qu'un objet ?

- Entité identifiable, concrète ou abstraite
- Un objet réagit à des messages qu'on lui envoie ; la façon dont il réagit constitue le comportement de l'objet
- Il ne réagit pas toujours de la même façon à un même message ; sa réaction dépend de l'état dans lequel il est

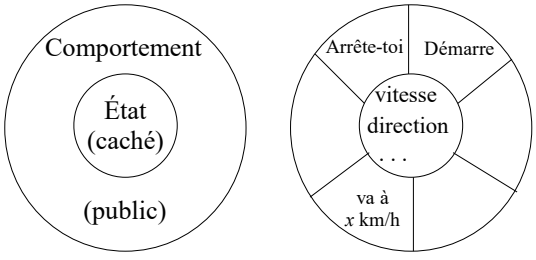
28

Notion d'objet en Java

- Un objet a
  - une adresse en mémoire (identifie l'objet)
  - un comportement (ou interface)
  - un état
- Le comportement est donné par des fonctions ou procédures, appelées méthodes
- L'état interne est donné par des valeurs de variables
- L'objet est décrit par une classe

29

Un objet



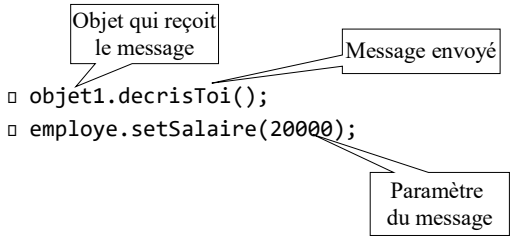
Un objet Une voiture

30

Interactions entre objets

- Les objets interagissent en s'envoyant des messages synchrones Signification de synchrone ?
- Messages qu'on peut envoyer à l'objet ↔ Méthodes de la classe d'un objet
- Quand un objet reçoit un message, il exécute la méthode correspondante

Exemples de messages



Paradigme objet

- La programmation objet est un paradigme, une manière de « modéliser le monde » :
  - des objets ayant un état interne et un comportement
  - collaborent en s'échangeant des messages
  - pour fournir les fonctionnalités qu'on demande à l'application

D'autres paradigmes

- Programmation impérative (Pascal, C) : décrit les opérations à exécuter sous la forme d'une séquence d'instructions qui modifie l'état du programme (les variables)
- Programmation fonctionnelle (Scheme, Lisp) : un programme est un enchainement / emboitement de fonctions ; pas d'affectation de valeurs à des variables

D'autres langages objet

- C++
- C#
- Smalltalk
- JavaScript

Les classes en Java

Regrouper les objets

- On peut le plus souvent dégager des types d'objets
- Par exemple, tous les livres dans une application de gestion d'une bibliothèque
- Une classe correspond à un type d'objets
- Objet du type défini par une classe : instance de la classe

Eléments d'une classe

```
public class Livre {
  private String titre, auteur;
  private int nbPages;

  public Livre(String unTitre, String unAuteur) {
    titre = unTitre;
    auteur = unAuteur;
  }
  public String getAuteur() {      // accesseur
    return auteur;
  }
  public void setNbPages(int nb) { // modificateur
    nbPages = nb;
  }
}
```

Variables d'instance

Constructeurs

Méthodes

Rôles d'une classe

- Type qui décrit une structure (variables d'état) et un comportement (méthodes)
- Générateur d'objets (par ses constructeurs)
- Module pour décomposer une application en entités plus petites
- Unité d'encapsulation : les membres `public` sont vus de l'extérieur mais les membres `private` sont cachés

Conventions pour les identificateurs

- Seuls les noms de classes commencent par une majuscule :
  `Cercle`, `Object`
- Majuscule devant les mots contenus dans un identificateur :
  `UneClasse`, `uneMethode`, `uneAutreVariable`

Commentaires

- Sur une seule ligne :
```
// Voici un commentaire
int prime = 1500; // prime fin de mois
```
- Sur plusieurs lignes :
```
/* Première ligne du commentaire
   suite du commentaire */
```
- Documentation automatique par ***javadoc***
```
/**
 * Cette méthode calcule ...
 * Elle utilise ...
 */
```


Les constructeurs

Constructeurs d'une classe

- Un ou plusieurs constructeurs
- Un constructeur sert à
  - créer les instances (les objets) de la classe
  - initialiser l'état de ces instances
- Un constructeur
  - a le même nom que la classe
  - n'a pas de type retour

Création d'une instance

```
public class Employe {
   private String nom, prenom;
   private double salaire;
  // Constructeur
  public Employe(String n, String p) {
    nom = n;
    prenom = p;
  }
  . . .
  public static void main(String[] args) {
    Employe e1;
    e1 = new Employe("Dupond", "Pierre");
    e1.setSalaire(1200);
    . . .
  }
}
```

variables d'instance

Plusieurs constructeurs (surcharge)

```
public class Employe {
   private String nom, prenom;
   private double salaire;
  // 2 Constructeurs
  public Employe(String n, String p) {
    nom = n;
    prenom = p;
  }
  public Employe(String n, String p, double s) {
    nom = n;
    prenom = p;
    salaire = s;
  }
  . . .
  e1 = new Employe("Dupond", "Pierre");
  e2 = new Employe("Durand", "Jacques", 1500);
```


Plusieurs constructeurs (surcharge)

```
public class Employe {
   private String nom, prenom;
   private double salaire;
  // 2 Constructeurs
  public Employe(String n, String p) {
    nom = n;
    prenom = p;
  }
  public Employe(String n, String p, double s) {
    nom = n;
    prenom = p;
    salaire = s;
  }
  . . .
  e1 = new Employe("Dupond", "Pierre");
  e2 = new Employe("Durand", "Jacques", 1500);
```

Duplication de code

Il n'est pas bon de dupliquer du code. Savez-vous pourquoi ?

Désigner un constructeur par `this()`

```
public class Employe {
   private String nom, prenom;
   private double salaire;
  // Ce constructeur appelle l'autre constructeur
  public Employe(String n, String p) {
    this(n, p, 0);
  }
  public Employe(String n, String p, double s) {
    nom = n;
    prenom = p;
    salaire = s;
  }
  . . .
  e1 = new Employe("Dupond", "Pierre");
  e2 = new Employe("Durand", "Jacques", 1500);
```


Désigner un constructeur par `this()`

```
public class Employe {
   private String nom, prenom;
   private double salaire;
  // Ce constructeur appelle l'autre constructeur
  public Employe(String n, String p) {
    nom = n;
    prenom = p;
  }
  public Employe(String n, String p, double s) {
    this(n, p);
    salaire = s;
  }
  . . .
  e1 = new Employe("Dupond", "Pierre");
  e2 = new Employe("Durand", "Jacques", 1500);
```

Constructeur par défaut

- Lorsque le code d'une classe ne contient pas de constructeur, un constructeur sans paramètres est automatiquement ajouté par le compilateur
- Pour une classe `Classe`, ce constructeur sera :

```
[public] Classe() { }
```

Même accessibilité que la classe (`public` ou non)

Les méthodes

- Une méthode correspond à un message qu'on envoie à une instance de la classe

Types de méthode (1)

- Pour accéder aux variables depuis l'extérieur de la classe :
  - accesseurs pour lire les valeurs des variables (*getter*)
  - modificateurs pour modifier leur valeur (*setter*)
- La plupart des méthodes permettent d'offrir des services plus complexes aux autres instances

Types de méthode (2)

- Des méthodes `private` servent de « sous-programmes » utilitaires aux autres méthodes de la classe

Exemples de méthodes

```
public class Employe {
  . . .
  public void setSalaire(double unSalaire) {
    if (unSalaire >= 0.0)
      salaire = unSalaire;
  }
  public double getSalaire() {
    return salaire;
  }
  public boolean accomplir(Tache t) {
    ...
  }
}
```

Demander à `t` la description de la tâche, pour l'accomplir en choisissant la meilleure façon, selon le contexte

Classe qui décrit une tâche à accomplir

Surcharge d'une méthode

Overload en anglais

- Méthode qui a
  - le même nom
  - mais pas la même signature qu'une autre méthode

```
calculerSalaire(int)
calculerSalaire(int, double)
```


55

`toString()`

- Conseillé d'inclure une méthode `toString` dans toutes les classes
- Elle retourne un texte (type `String`) qui décrit l'instance
- Description compacte et précise pour être utile pendant la mise au point
- `System.out.println(objet)` affiche la valeur retournée par `objet.toString()`

56

Exemple

```
public class Livre {
  ...
  public String toString() {
    return "Livre{titre=" + titre
      + ", auteur=" + auteur
      + ", nbPages=" + nbPages
      + "}";
  }
}
```

```
Livre{titre=Les Misérables, auteur=Victor Hugo, nbPages=320}
```


57

Les variables

58

Types de variables

- Variable d'instance :
  - conserve l'état d'une instance
  - déclarée en dehors de toute méthode
  - partagée par toutes les méthodes de la classe
- Variable locale :
  - conserve une valeur utilisée par la méthode
  - déclarée à l'intérieur d'une méthode
  - accessible seulement dans le bloc ({ … }) dans lequel elle a été déclarée

59

Déclaration des variables

- Toute variable doit être déclarée avant d'être utilisée :

```
double prime;
Employe e1;
Point centre;
```

- La déclaration indique le type de la variable : le type des données qui seront contenues dans la variable

60

Initialisation d'une variable

- Une variable doit recevoir une valeur avant d'être utilisée
- L'utilisation d'une variable locale non initialisée provoque une erreur

```
int x, y;
y = x + 1;
```

Quel est le problème ?

- Une variable d'instance est initialisée automatiquement à la valeur par défaut de son type (0 pour le type `int`, par exemple)

61

Initialisation d'une variable (2)

- On peut initialiser une variable en la déclarant :

```
double prime = 2000.0;
Employe e1 = new Employe("Dupond", "Jean");
double salaire = prime + 5000.0;
```


62

Déclaration / création

```
public static void main(String[] args) {
  Employe e1;
  e1.setSalaire(12000);
  …
}
```

provoque une erreur `NullPointerException`

OK ?

- Il ne faut pas confondre
  - déclaration d'une variable
  - création d'un objet référencé par cette variable

63

Déclaration / création

```
public static void main(String[] args) {
  Employe e1;
  e1 = new Employe("Dupond", "Pierre");
  e1.setSalaire(12000);
  …
}
```


64

Désigner les variables d'une instance

```
Cercle c1 = new Cercle(p1, 10);
System.out.println(c1.rayon); // affiche 10
```

Le plus souvent pas possible en dehors de la classe `Cercle`

Pourquoi ?

65

`var`

- On peut se passer de déclarer le type des variables locales
- Leur type est alors inféré (déduit du contexte) par le compilateur
- Utiliser seulement lorsque ça simplifie le code et améliore la lisibilité

66

Exemples

- `int x = 8;`
  peut être remplacé par Aucun intérêt
  `var x = 8;`
  Pour le compilateur, `x` sera bien de type `int`
- Intéressant quand l'objet de type complexe est retourné par une méthode, et si ça ne nuit pas à la lisibilité du code :

```
Map.Entry<? extends String, ? extends Number>
    entry = iterateur.next();
```

peut être remplacé par

```
var entry = iterateur.next();
```


Accès aux membres d'une classe

Degrés d'encapsulation

- Pour les membres (variables et méthodes) et les constructeurs d'une classe (pas pour les variables locales)
- `private` : seule la classe y a accès
- `public` : toutes les classes sans exception y ont accès
- Par défaut (ni `public`, ni `private`), seules les classes du même paquetage y ont accès

Granularité de la protection des attributs d'une classe

- La protection se fait classe par classe (pas objet par objet)
- Un objet a accès à tous les attributs d'un objet de la même classe, même les attributs privés

Protection de l'état interne d'un objet

- L'état d'un objet (variables d'instance) doit être `private`
- Si on veut autoriser la lecture d'une variable, on lui associe un accesseur avec la protection que l'on veut
- Si on veut autoriser la modification d'une variable, on lui associe un modificateur, qui permet la modification tout en contrôlant la validité de la modification

Désigner l'instance qui reçoit le message, « `this` »

this

- Désigne l'instance qui a reçu le message (instance courante)
- « this.salaire » désigne le salaire de l'instance courante
- Lorsqu'il n'y a pas d'ambiguïté, « this. » est optionnel

73

Exemple de this implicite

```
public class Employe {
  private double salaire;
  . . .
  public void setSalaire(double unSalaire) {
    salaire = unSalaire;
  }
  public double getSalaire() {
    return salaire;
  }
  . . .
}
```

Implicitement this.salaire

Implicitement this.salaire

74

this explicite

- this est utilisé surtout dans 2 occasions :
  - pour distinguer une variable d'instance et un paramètre :

```
public void setSalaire(double salaire) {
  this.salaire = salaire;
}
```

  - un Employe passe une référence de lui-même à un Comptable :

Signification de ce code ?

```
salaire = comptable.calculeSalaire(this);
```

Comptable, calcule le salaire de **moi**

75

this explicite

- this est utilisé surtout dans 2 occasions :
  - pour distinguer une variable d'instance et un paramètre :

```
public void setSalaire(double salaire) {
  this.salaire = salaire;
}
```

  - un Employe passe une référence de lui-même à un Comptable :

```
salaire = comptable.calculeSalaire(this);
```

Dans quelle classe pourrait-on trouver ce code ?

76

this explicite

- this est utilisé surtout dans 2 occasions :
  - pour distinguer une variable d'instance et un paramètre :

```
public void setSalaire(double salaire) {
  this.salaire = salaire;
}
```

  - un Employe passe une référence de lui-même à un Comptable :

```
salaire = comptable.calculeSalaire(this);
```

Dans quelle classe peut-on trouver la méthode calculeSalaire ?

77

Appel d'une méthode de la même classe

- Il suffit de nommer la méthode
- Par exemple

```
public int m1() {
  ...
  int y = m2(); // équivalent à y = this.m2();
  ...
}
public int m2() { ... }
```


78

Méthodes et variables de classe

79

Variable de classe (`static`)

- Partagée par toutes les instances d'une classe

80

Exemple de variable de classe

```
public class Employe {
  private String nom, prenom;
  private double salaire;
  private static int nbEmployes = 0;

  public Employe(String nom, String prenom) {
    this.nom = nom;
    this.prenom = prenom;
    nbEmployes++;
  }
  . . .
}
```

Initialisation exécutée une seule fois, quand classe `Employe` chargée en mémoire

A quoi sert `nbEmploye` ?

81

Méthode de classe (`static`)

- Exécution indépendante d'une instance particulière de la classe
- Correspond à un message envoyé à la classe
- Exemple :

```
public static int getNbEmployes() {
  return nbEmployes;
}
```

- Erreur à la compilation si la méthode utilise une variable d'instance

82

Désigner une méthode de classe

- Depuis une autre classe, préfixer par le nom de la classe :

```
int n = Employe.getNbEmploye();
```

- Depuis la même classe, on peut seulement donner le nom de la méthode :

```
int n = getNbEmploye();
```


83

TYPES DE DONNÉES EN JAVA

84

Type

- Sert à indiquer ce que peut contenir une variable
- Utilisé par le compilateur pour vérifier les valeurs affectées à une variable
- `v = expression`
  le type de l'expression doit être compatible avec le type de la variable

85

Exemple

```
String v1 = "Hello";
...
int v2 = v1 + 8;
```

Cette dernière ligne provoque une erreur à la compilation : incompatible types: String cannot be converted to int

86

2 sortes de types en Java

- Les types primitifs (par exemple `int`)
- Les autres types ; par exemple les classes

87

Types primitifs

- `boolean` (`true`/`false`)
- Nombres entiers : `byte`, `short`, `int`, `long`
- Nombres non entiers, à virgule flottante : `float`, `double`
- Caractère (un seul) : `char` (2 octets) ; codage Unicode (pas ASCII)

88

Erreurs de calculs

Explication ?

- 16.8 + 20.1 donne 36.900000000000006
- Pour éviter cette sorte d'erreur, utiliser la classe `java.math.BigDecimal`

89

Constantes nombres

```
35          // de type int
2589L       // de type long
456.7 ou 4.567e2 // 456,7 ; de type double
.123587E-25F // de type float
1_567_458
```


90

Constante de type caractère

- Entouré par « ' »
- CR et LF interdits (caractères de fin de ligne)

```
'A'
'\t' '\n' '\r' '\'' '\"' '\\'
'\u20ac'
```

(\u suivi du code Unicode hexadécimal à 4 chiffres ; €)

'\u0604' correspond à ﷺ

91

Autres constantes

- Type booléen
  - `false`
  - `true`
- Référence inexistante ; pour *tous les types non primitifs*
  - `null`

92

Opérateurs sur les types primitifs

- Les plus utilisés :

```
= + - * / % ++ -- += -= *= /=
== != > < >= <=
```

&& || ! (et, ou, négation)

Reste de la division entière (modulo)

Division entière si les opérandes sont des entiers

93

Exemples d'utilisation des opérateurs

```
int x = 9, y = 2;
int z = x / y;      // z = ?
                    // z = 4
z = x++ / y;        // x = ? z = ?
                    // z = 4 puis x = 10
z = --x / y;        // x = ? z = ?
                    // x = 9 puis z = 4
```


94

Exemples d'utilisation des opérateurs

```
if (z == y)         // il faut 2 "=" !
  x += y;           // x = x + y
if (y != 0 && x / y > 8)
                    // pas d'erreur si y = 0
```


95

Types des résultats des calculs avec des nombres

- Un calcul entre entiers donne un résultat de type `int` (ou `long`, si un des opérandes est de type `long`)
- Par exemple, `byte` + `byte` donne un `int`

96

Traitement différent pour les objets et les types primitifs

- Les variables contiennent
  - des valeurs de types primitifs
  - des références aux objets

97

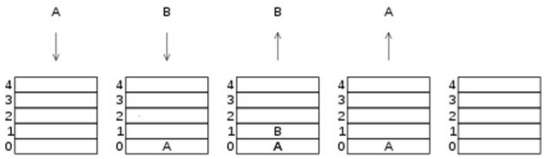
Exemple d'utilisation des références

```
int m() {
  A a1 = new A();
  A a2 = a1;
  ...
}
```

- Comment est exécutée cette méthode `m()` ?

98

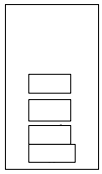
Pile



- On ajoute en empilant
- On enlève en dépilant
- Donc le 1er sorti de la pile est le dernier entré

99

Exécution d'un programme



Pile d'exécution

- Les espaces mémoire associés aux méthodes s'empilent et se dépilent dans la pile d'exécution
- L'espace mémoire associée à une méthode contient les valeurs des variables locales et des paramètres de la méthode

100

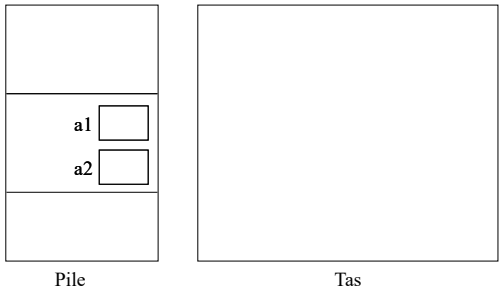
Mémoire en Java

- 2 espaces mémoire bien distincts :
  - la pile d'exécution
  - le tas où sont rangés les objets ;
    les variables qui « contiennent » un objet contiennent en fait une référence vers l'objet qui est dans le tas

101

Références

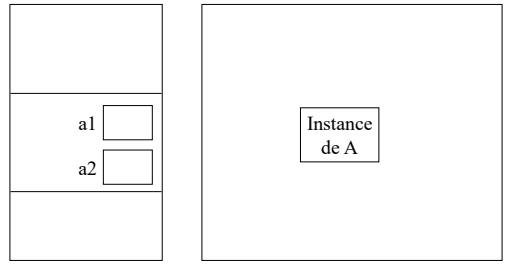
```
int m() {
  A a1 = new A();
  A a2 = a1;
  ...
```



102

```
int m() {
  A a1 = new A();
  A a2 = a1;
  ...
```

Références

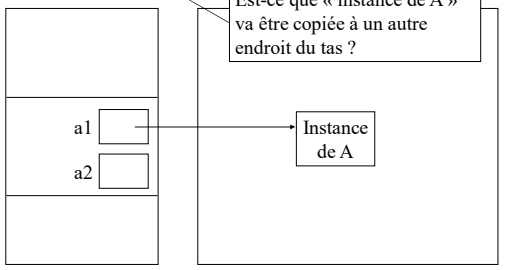


103

```
int m() {
  A a1 = new A();
  A a2 = a1;
  ...
```

Références

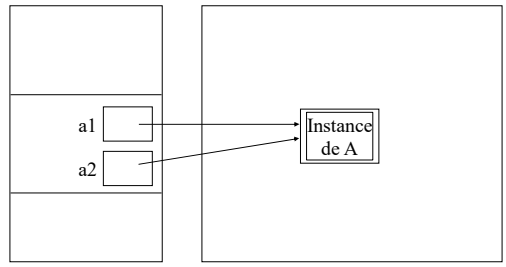
Est-ce que « instance de A » va être copiée à un autre endroit du tas ?



104

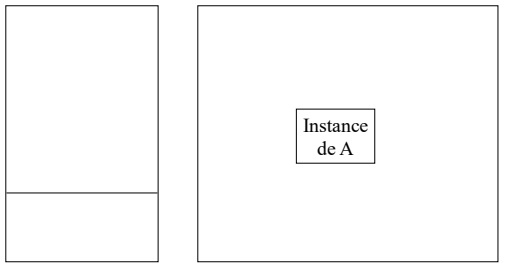
```
int m() {
  A a1 = new A();
  A a2 = a1;
  ...
```

Références



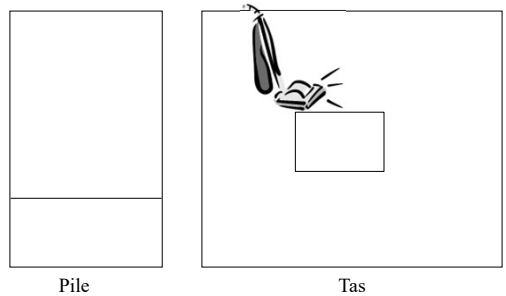
105

Après l'exécution de la méthode `m()`, l'instance de `A` n'est plus référencée mais reste dans le tas



106

...le ramasse-miette (*garbage collector*) interviendra à un moment aléatoire...



107

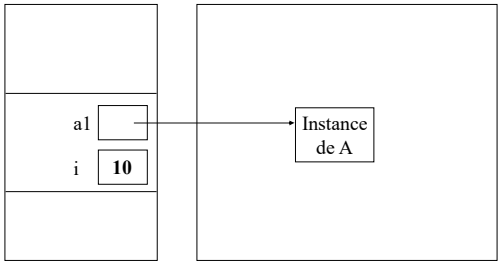
Garbage collector

- Libère la place occupée par les instances non référencées
- Travaille en arrière-plan
- Intervient
  - quand le système a besoin de mémoire
  - ou, de temps en temps, avec une priorité faible

108

Types primitifs

```
int m() {
  A a1 = new A();
  int i = 10;
  ...
```



109

Opérateur `instanceof`

o minuscule

- Exemple : `if (x instanceof Livre)`
- Le résultat est un booléen :
  - `true` si `x` est une instance de la classe `Livre`
  - `false` sinon

110

Forcer un type en Java

- Parfois nécessaire de forcer le compilateur à considérer une expression comme étant d'un certain type
- On utilise alors le *cast* (transtypage) :

```
int x = 10, y = 3;
double z = (double)x / y;
```

Ne caste que `x` ; que donnerait `(double)(x / y)` ?

Pourquoi ce cast ?
Le résultat serait différent sans le cast ?

111

Exemples de *casts*

```
byte b1 = 20;
byte b2 = 15;
byte b3 = b1 + b2;
```
Quel est le problème ?

```
byte b3 = (byte) (b1 + b2);
float f = 1.2;
```
Quel est le problème ?

```
float f = 1.2f; // ou = (float) 1.2;
```


112

Variable `final`

- Si la variable est d'un type primitif, sa valeur ne peut changer :

```
static final int AGE_LEGAL = 18;
```

- Si la variable référence un objet, elle ne pourra référencer un autre objet mais l'état de l'objet pourra être modifié

```
final Employe e = new Employe("Bibi");
. . .
e.nom = "Toto";          // Autorisé !
e.setSalaire(12000);     // Autorisé !
e = new Employe("Bob"); // Interdit
```


113

Constantes `static`

- Par convention, les noms des constantes (« variable » avec `static final`) sont écrites en majuscules
- Exemple : la classe `java.lang.Math` contient la constante `PI` :

```
public static final double PI =
    3.14159265358979323846;
```

- Les mots dans le nom sont séparés par « _ » : `VITESSE_LIMITE`

114

Types énumérés

115

Énumération externe à une classe

- Type défini en énumérant toutes ses valeurs possibles, dans un fichier `CouleurCarte.java` :

```
public enum CouleurCarte {
  TREFLE, CARREAU, COEUR, PIQUE
}
```

- Utilisation :

```
public class Carte {
  private CouleurCarte couleur;
  ...
}
```


116

Énumération interne à une classe

-
```
public class Carte {
  public enum Couleur
    {TREFLE, CARREAU, COEUR, PIQUE};

  private Couleur couleur;
  . . .
  this.couleur = Couleur.PIQUE;
```

- Utilisation depuis une autre classe :

```
carte.setCouleur(Carte.Couleur.TREFLE);
```


117

Tableaux

118

Les tableaux sont des objets

- Créés par `new`
- Ils sont enregistrés dans le tas et les variables contiennent des références aux tableaux
- Une variable d'instance :
  `public final int length`

119

Mais des objets particuliers

- Syntaxe particulière pour
  - la déclaration des tableaux
  - l'initialisation

120

Déclaration et création des tableaux

- Déclaration : la taille n'est pas fixée
  `int[] tabEntiers;`
- Création : on doit donner la taille
  `tabEntiers = new int[5];`
  Les éléments du tableau reçoivent la valeur par défaut du type de base du tableau (0 pour `int`, `null` si type non primitif)
- La taille ne peut pas être modifiée

Initialisation des tableaux

- Syntaxe spéciale pour initialiser un tableau ; taille calculée automatiquement (autorisé seulement dans la déclaration) :

```
int[] tabEntiers = {8, 2*8, 3, 5, 9};
```

```
Employe[] employes = {
  new Employe("Dupond", "Sylvie"),
  new Employe("Durand", "Patrick")
};
```

Taille de `tabEntiers` ? | Taille de `employes` ?

Affectation en bloc

- On peut affecter « en bloc » tous les éléments d'un tableau avec un tableau anonyme :

```
int[] t;
. . .
t = new int[] {1, 2, 3};
```


Tableaux - utilisation

- Affectation des éléments ; l'indice commence à 0 et se termine à `tabEntiers.length - 1`
  `tabEntiers[0] = 12;`
- Taille du tableau — Qu'est-ce que ça fait ?

```
int longueur = tabEntiers.length;
int e = tabEntiers[longueur];
```

`ArrayIndexOutOfBoundException`

- Déclarations dans la signature d'une méthode
  `int[] m(String[] t)`

Paramètres de la ligne de commande : exemple de tableau de chaînes

```
class Arguments {
  public static void main(String[] args) {
    for (int i = 0; i < args.length; i++)
      System.out.println(args[i]);
  }
}
```

```
java Arguments toto bibi
affichera ?
toto
bibi
```


OK ?

```
Employe[] personnel = new Employe[100];
personnel[0].setNom("Dupond");
```

```
Employe[] personnel = new Employe[100];
personnel[0] = new Employe();
personnel[0].setNom("Dupond");
```

Création employé

Tableaux à plusieurs dimensions

- Déclaration

```
int[][] notes;
```

- Chaque élément du tableau contient une référence vers un tableau
- Création

```
notes = new int[30][3];
notes = new int[30][];
```

Chacun des 30 étudiants a au plus 3 notes

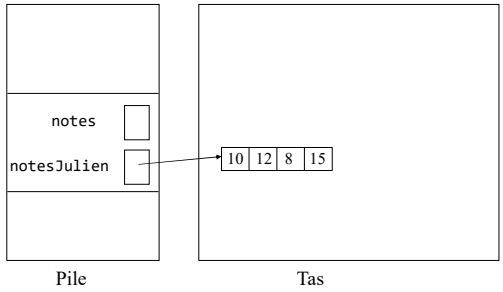
Chacun des 30 étudiants a un nombre de notes variable

Il faut donner au moins les premières dimensions

127

Tableaux dans le tas

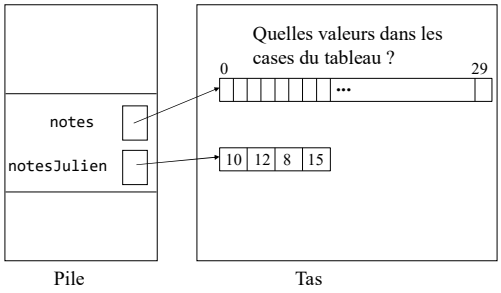
```
int[][] notes;
notes = new int[30][];
notes[0] = notesJulien;
```



128

Tableaux dans le tas

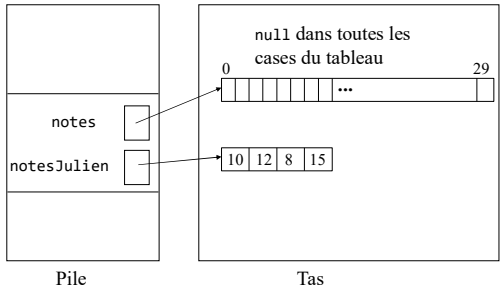
```
int[][] notes;
notes = new int[30][];
notes[0] = notesJulien;
```



129

Tableaux dans le tas

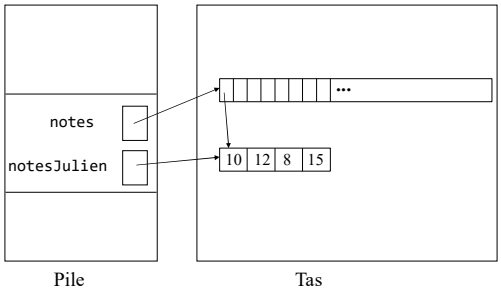
```
int[][] notes;
notes = new int[30][];
notes[0] = notesJulien;
```



130

Tableaux dans le tas

```
int[][] notes;
notes = new int[30][];
notes[0] = notesJulien;
```



131

Initialisation de tableaux à plusieurs dimensions

- Déclaration, création et initialisation

```
int[][] notes = { {10, 11, 9} // 3 notes
                  {15, 8}     // 2 notes
                  . . .
                };
```

- Affectation

```
notes[10][2] = 12;
```


132

Afficher les valeurs d'un tableau

- Utile pour la mise au point : méthodes `public static String toString` de la classe `java.util.Arrays`
- `Arrays.toString(t)` retourne « `[12.5, 134.76]` »

Classes de base

- *Classes pour les chaînes de caractères*
- *Classes qui enveloppent les types primitifs*

Chaînes de caractères

2 classes

- `String` pour les chaînes constantes
- `StringBuilder` pour les chaînes variables
- On utilise le plus souvent `String`, sauf si la chaîne doit être fréquemment modifiée

Affectation d'une valeur littérale

- Syntaxe particulière pour `String` :

```
chaine = "Bonjour";
```

Qu'est-ce qu'on aurait dû normalement écrire pour créer une instance de `String` ?

Nouvelle affectation avec les `String`

```
String chaine = "Bonjour";
chaine = "Hello";
```

Objet `String` pas modifié

- La dernière instruction correspond aux étapes suivantes :
  1) Une nouvelle valeur (*Hello*) est créée
  2) `chaine` référence la nouvelle chaîne *Hello*
  3) La place occupée par la chaîne *Bonjour* pourra être récupérée par le ramasse-miette

Concaténation de chaînes

```
String s = "Bonjour" + " les amis";
```

- Si un des 2 opérandes de **+** est une `String`, l'autre est traduit en `String` :

```
int x = 5;
s = "Valeur de x = " + x;
```

  - un objet est converti en utilisant la méthode `toString()` de sa classe

Égalité de `String`s

- La méthode `equals` teste si 2 `String`s contiennent la même valeur :

```
String s1, s2;
s1 = "Bonjour ";
s2 = "les amis";
if ((s1 + s2).equals("Bonjour les amis"))
  System.out.println("Egales");
```

- « == » ne doit pas être utilisé pour comparer 2 chaînes

Comparaison de `String`s

- `s.compareTo(t)`
  retourne un entier qui a le « signe de `s-t` » : positif si `s` vient après `t` dans l'ordre lexicographique (alphabétique pour les lettres)
- `"bonjour".compareTo("les amis")`
  renvoie une valeur de quel signe ?

Quelques méthodes de `String`

- Extraire une sous-chaîne : `substring`
- Rechercher des sous-chaînes :
  `indexOf`
  `LastIndexOf`
- `startsWith` , `endsWith` , `toUpperCase`, `toLowerCase`
- Il y en a beaucoup d'autres…

Exemple

javadoc :

```
public int indexOf(String str, int fromIndex)
Returns the index within this string of the first occurrence of
the specified substring, starting at the specified index.
```

```
String s = "bonjour les amis";
System.out.println(s.indexOf("les"));
System.out.println(s.indexOf("o"));
System.out.println(s.indexOf("o", 3));
System.out.println(s.indexOf("o", 6));
```

Que sera-t-il affiché ?

Méthodes pour extraire un caractère

- Les caractères d'une `String` sont numérotés de `0` à `length() - 1`
- « `char charAt(int i)` » extrait le i^{ème} `char`
- Attention, des caractères Unicode sont codés avec 2 `char` ; `charAt` peut alors ne ramener qu'une partie d'un caractère Unicode
- Heureusement ça n'est pas le cas des caractères occidentaux ou arabes

Expressions régulières

- Vous connaissez ?

145

Chaînes modifiables

- `StringBuilder` possède des méthodes qui modifient le receveur du message, ce qui évite la création de nouvelles instances :
  - `append` et `appendCodePoint`
  - `insert`
  - `replace`
  - `delete`

146

`String` et `StringBuilder`

- Utiliser plutôt la classe `String` qui possède de nombreuses méthodes
- Si la chaîne de caractères doit être souvent modifiée, passer à `StringBuilder`

147

Exemple

```
String[] t;
... // met des valeurs dans le tableau
StringBuilder sb = new StringBuilder();
for (int i = 0; i < t.length; i++) {
    sb.append(t[i]);
}
String chaine = sb.toString();
```

Que fait ce code ?

Est-ce qu'on aurait pu utiliser `String` à la place de `StringBuilder` ?

Quel est l'intérêt de passer par `StringBuilder` ?

148

Constante `String` (1/2)

- Une constante `String` entourée de « " » ne peut s'écrire que sur une seule ligne
- Ce code ne compile pas :

```
String html = "
  <html>
    <body>
      <p>Hello, world</p>
    </body>
  </html>
";
```


149

Constante `String` (2/2)

- Il faudra écrire

```
String html = "  <html>\n" +
"    <body>\n" +
"      <p>Hello, world</p>\n" +
"    </body>\n" +
"  </html>";
```

Pas trop lisible ni pratique à taper !

150

Bloc de texte

- Permet d'écrire une constante `String` sur plusieurs lignes, en conservant l'indentation
- Un bloc de texte
  - commence par « `"""` », suivi par un passage à la ligne
  - se termine par « `"""` » en fin de ligne

151

Exemple

```
public static void main(String[] args) {
  String html = """
    <html>
      <body>
        <p>Hello, world</p>
      </body>
    </html>
    """;
}
```

Valeur de la variable html

```
<html>
  <body>
    <p>Hello, world</p>
  </body>
</html>
```


152

Indentation

```
public static void main(String[] args) {
  String html = """
....<html>
....  <body>
....    <p>Hello, world</p>
....  </body>
....</html>
    """;
}
```

Les espaces représentés par des « . » sont ignorés pour la valeur de la variable

Valeur de la variable html

```
<html>
  <body>
    <p>Hello, world</p>
  </body>
</html>
```


153

Classes enveloppes de type primitif - « Autoboxing/Unboxing »

154

Classes enveloppes des types primitifs

- Certaines manipulations nécessitent de travailler avec des objets et pas avec des types primitifs
- Classes pour envelopper les types primitifs : `Byte`, `Short`, `Integer`, `Long`, `Float`, `Double`, `Boolean`, `Character`
- Exemple (la meilleure façon de récupérer un `Integer`) :
  `Integer i = Integer.valueOf(8);`

155

Conversion en `int` d'une `String`

```
public class AfficheParam {
  public static void main(String[] args) {
    int i = Integer.parseInt(args[0]);
    System.out.println(i*2);
  }
}
```

Donnez un exemple de commande pour exécuter cette classe

156

Listes et types primitifs

- Une liste ne peut contenir de type primitif ; avec les anciennes versions de Java on écrivait :

```
liste.add(new Integer(89));
int i = liste.get(n).intValue();
```


157

Boxing/unboxing

- Le « *boxing* » (mise en boîte) automatise le passage des types primitifs vers les classes qui les enveloppent
- L'opération inverse s'appelle « *unboxing* »
- On écrit :

```
liste.add(89);
int i = liste.get(n);
```


158

INSTRUCTIONS DE CONTRÔLE

159

Interdit de cacher une variable locale

- Une variable locale ne peut en cacher une autre :

```
int somme(int init) {
  int i = init;
  int j = 0;
  for (int i = 0; i < 10; i++) {
    j += i;
  }
  int init = 3;
}
```

Interdit !

160

`if ... else`

```
int x = y + 5;
if (x % 2 == 0) {
  type = 0;
  x++;
}
else
  type = 1;
```

Si y = 2, type = ?

Un bloc serait préférable

161

`if ... else`

```
int x = y + 5;
if (x % 2 == 0) {
  type = 0;
  x++;
}
else {
  type = 1;
}
```


162

Expression conditionnelle

expressionBooléenne ? *expression1* : *expression2*

```
int y = (x % 2 == 0) ? x + 1 : x;
```

est équivalent à

```
int y;
if (x % 2 == 0)
  y = x + 1
else
  y = x;
```

Quelle valeur pour `y` si x = 2 ?

Distinction de cas suivant une valeur

```
switch(expression) {
case val1:
  instructions;
  break;
...
case valn:
  instructions;
  break;
default: instructions;
}
```

Attention, sans `break` les instructions du cas suivant sont exécutées !

expression est de type `char`, `byte`, `short`, ou `int`, ou énumération, ou `String`

Exemple de `switch`

```
char lettre;
int nbVoyelles = 0, nbA = 0,
    nbT = 0, nbAutre = 0;
. . .
switch (lettre) {
case 'a' : nbA++;
case 'e' :              // pas d'instruction !
case 'i' : nbVoyelles++;
           break;
case 't' : nbT++;
           break;
default :  nbAutre++;
}
```

Si, dans une boucle, `lettre` prend comme valeur chacune des lettres de « attention », quelles valeurs auront les variables ?

`switch` pour une expression

- Depuis Java 14, `switch` peut être une expression : on peut affecter la valeur d'un `switch` à une variable
- `switch` doit alors retourner une valeur pour toutes ses branches (sauf si une exception est lancée)
- La valeur retournée par une branche est indiquée par `yield`

Exemple

```
int nbJours =
  switch(mois) {
    case "avril": case "juin" :
    case "septembre" : case "novembre":
      yield 30;
    case "janvier": case "mars": case "mai":
    case "juillet": case "août": case "décembre":
      yield 31;
    case "février":
      ...
    default:
      ...
  };
```


Nouvelle syntaxe pour `switch`

- L'oubli de `break` étant fréquent, une nouvelle syntaxe (avec « `->` » à la place de « `:` ») a été ajoutée dans Java 14, avec un `break` automatique à la fin de chaque cas
- Une variable déclarée dans une branche de `switch` n'est pas connue des autres branches

Cas simple de `switch` avec `->`

```
switch(expression) {
   case val1 -> expression1;
   case val2 -> expression2;
   ...
   case valN -> expressionN;
   default -> expressionD;
}
```

expression1, *expression2*, … *expressionN*, *expressionD* sont les valeurs retournées par le `switch`

169

Compléments sur `switch` avec `->`

- Tous les cas doivent être couverts
- Plusieurs cas peuvent être réunis :
  `case val1, val2 -> expression1;`
- Une expression peut être remplacée par un bloc de plusieurs instructions :

```
case val1 -> {
    instruction1;
    yield valeurRetour1;
}
```

Dans un bloc, `yield` indique la valeur retournée par le `switch`

170

Exemple

```
int nbJours = switch(mois) {
  case 1, 3, 5, 7, 8, 10, 12 -> 31;
  case 4, 6, 9, 11 -> 30;
  case 2 -> estBissextile(annee) ? 29 : 28;
  default ->
      throw new IllegalArgumentException("Mois invalide");
}
```


171

Répétitions « `while` » et « `do while` »

```
while(expressionBooléenne)
   bloc-instructions ou instruction
```

```
do
   bloc-instructions ou instruction
while(expressionBooléenne)
```

Boucle exécutée au moins une fois

172

Exemple

```
public class Diviseur {
  public static void main(String[] args) {
    int i = Integer.parseInt(args[0]);
    int j = 2;
    while (i % j != 0) {
      j++;
    }
    System.out.println("PPD de "
                       + i + " : " + j);
  }
}
```

Que fait ce code ?

Comment lancer l'exécution ?

Pas très performant ! Dites pourquoi et essayez d'améliorer.

Code avec une boucle `do` ?

173

Exemple avec `do`

```
public class Diviseur {
  public static void main(String[] args) {
    int i = Integer.parseInt(args[0]);
    int j = 2;
    do {
      j++;
    } while (i % j != 0)
    System.out.println("PPD de "
                       + i + " : " + j);
  }
}
```

Problème ?

`java Diviseur 2` affiche quoi ?

Comment corriger ?

174

Répétition for

```
for(init; test; incrément){
   instructions;
}
```

est équivalent à

```
init;
while (test) {
   instructions;
   incrément;
}
```


175

Exemple de for

```
int somme = 0;
for (int i = 0; i < tab.length; i++) {
  somme += tab[i];
}
System.out.println(somme);
```

Qu'est-ce qui sera affiché ?

176

« for each »

```
String[] noms = new String[50];
...
// Lire « pour chaque nom dans noms »
for (String nom : noms) {
  System.out.println(nom);
}
```

- Plus lisible qu'une boucle *for* ordinaire
- Mais on ne dispose pas de la position dans le tableau (pas de « variable de boucle »)

177

Instructions liées aux boucles

- `break` sort de la boucle et continue après la boucle
- `continue` passe à l'itération suivante

178

Exemple de continue et break

```
int somme = 0;
for (int i = 0; i < tab.length; i++) {
  if (tab[i] == 0) break;
  if (tab[i] < 0) continue;
  somme += tab[i];
}
System.out.println(somme);
```

Qu'affiche ce code avec le tableau
1 ; –2 ; 5 ; -1 ; 0 ; 8 ; -3 ; 10 ?

179

COMPLÉMENTS SUR LES MÉTHODES

180

Passage des arguments des méthodes

- Passage par valeur :
  la valeur de l'argument est recopiée dans l'espace mémoire de la méthode
- Attention, pour les objets, la valeur passée est une référence ; c'est cette référence qui est recopiée, pas l'objet

181

Exemple de passage de paramètres

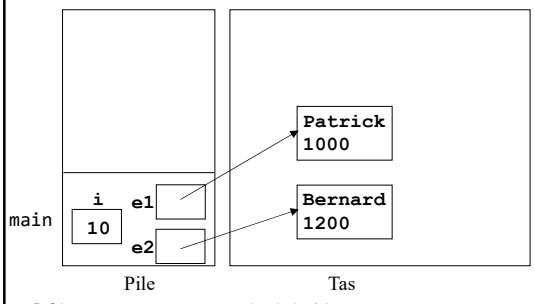
```
public static void m(int ip, Employe e1p, Employe e2p) {
  ip = 100;
  e1p.salaire = 800;
  e2p = new Employe("Pierre", 900);
}
public static void main(String[] args) {
  Employe e1 = new Employe("Patrick", 1000);
  Employe e2 = new Employe("Bernard", 1200);
  int i = 10;
  m(i, e1, e2);
  System.out.println(i + '\n' + e1.salaire
                     + '\n' + e2.nom);
}
```

Que sera-t-il affiché ?

182

Passage de paramètres

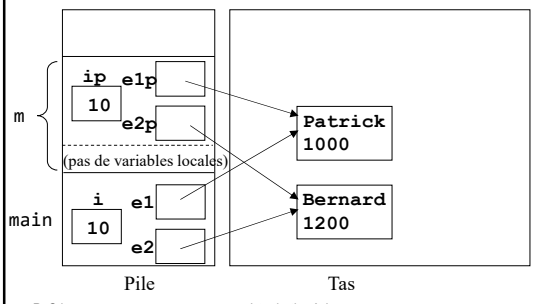
```
main :
Employe e1 = new Employe("Patrick", 1000);
Employe e2 = new Employe("Bernard", 1200);
int i = 10;
```



183

Passage de paramètres

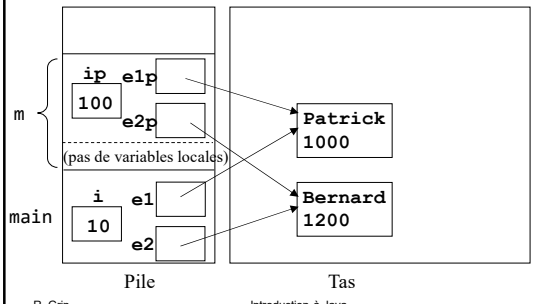
```
main :
m(i, e1, e2);
```



184

Passage de paramètres

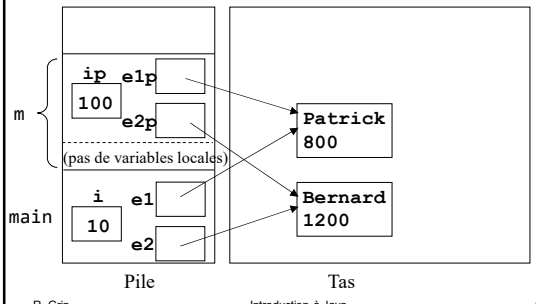
```
m() :
ip = 100;
e1p.salaire = 800;
e2p = new Employe("Pierre", 900);
```



185

Passage de paramètres

```
m() :
ip = 100;
e1p.salaire = 800;
e2p = new Employe("Pierre", 900);
```

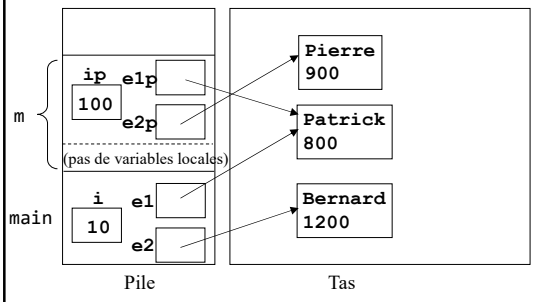


186

Passage de paramètres

m() :

```
ip = 100;
e1p.salaire = 800;
e2p = new Employe("Pierre", 900);
```

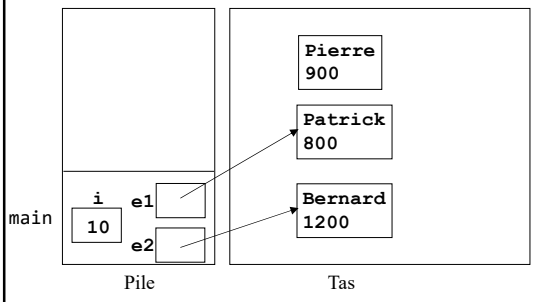


187

Passage de paramètres

main :

```
System.out.println(i + '\n' + e1.salaire
                   + '\n' + e2.nom);
```



188

Paramètre `final`

- Ne peut être modifié dans la méthode
- Attention ! si le paramètre n'est pas d'un type primitif, l'objet référencé peut être modifié
  `int m(final Employe e1)`
- Alors, que ne peut pas faire la méthode à cause de `final` ?

Le salaire de l'employé `e1` pourra être modifié

189

Nombre variable d'arguments

- Le type du dernier paramètre peut être suivi de « `...` »
- `m(int p1, String... params)`
  la méthode `m` peut prendre de 0 à plusieurs arguments de type `String`
- Traduit par le compilateur en :
  `m(int p1, String[] params)`

190

Exemple

```
public static int max(int... valeurs) {
  if (valeurs.length == 0)
    throw new IllegalArgumentException(
              "Au moins 1 valeur requise");
  int max = valeurs[0];
  for (int i = 1; i < valeurs.length; i++) {
    if (valeurs[i] > max)
      max = valeurs[i];
  }
  return max;
}
```

Que fait cette méthode ?

Peut-on appeler `int m = max();`

Utilisation :
`int m = max(5, 8, 12, 7);`

Amélioration pour éviter une erreur à l'exécution ?

191

Exemple amélioré

```
public static int max(int valeur,
                      int... autresValeurs) {
  int max = valeur;
  for (int val : autresValeurs) {
    if (val > max)
      max = val;
  }
  return max;
}
```


192

Contrôle du compilateur pour la valeur de retour

- Ne compile pas :

```
double soustraction(double a, double b) {
  if (a > b)
    return a - b;
}
```

- Pourquoi ?

Récursivité des méthodes

```
static long factorielle(int n) {
  if (n == 0)
    return 1;
  else
    return n * factorielle(n - 1);
}
```

Valeur de factorielle(0) ?

Valeur de factorielle(1) ?

Valeur de factorielle(2) ?

Valeur de factorielle(3) ?