

Java de base

Université Française d'Égypte – TIC303

Version O 0.9 – 21/6/13

Richard Grin

<http://richard.grin.free.fr>
grin@unice.fr

Emploi du temps TIC303

- <http://richard.grin.free.fr/ufr/cours/tic303>
- 60 heures de cours et TPs sur 2 semaines du ****??**** au ****??****

R. Grin

Introduction à Java

2

À faire avant le 1^{er} TP

- Installer l'environnement (décrit au début du TP 0)
- Commencer à faire le TP 0
- Suivre ce qui est préconisé : ranger le $n^{\text{ième}}$ TP dans **java/tp/tpn**

R. Grin

Introduction à Java

3

Plan du cours (1/2)

- Java de base (ce support) : types primitifs, techniques de flux de contrôle, objets et classes
- Héritage et polymorphisme, classes abstraites et interfaces
- Exceptions
- Classes internes
- Collections (illustration des classes abstraites et des interfaces)
- Généricité

R. Grin

Introduction à Java

4

Plan du cours (2/2)

- Entrées-sorties
- Génération de documentation avec javadoc
- Distribution de logiciels, fichiers jar

R. Grin

Introduction à Java

5

Plan de cette partie

- Présentation de Java
- Notions de base sur la programmation objet
- Types en Java
- Syntaxe de Java
- Paquetages
- Quelques principes de programmation

R. Grin

Introduction à Java

6

Principales propriétés de Java

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

R. Grin

Introduction à Java

7

Premier programme Java

R. Grin

Introduction à Java

8

Le code source du premier programme

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

point d'entrée
d'exécution

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

R. Grin

Introduction à Java

9

La compilation fournit du *bytecode*

Programme écrit en Java

Programme source
UneClasse.java

Compilateur

Programme en *bytecode*,
langage de la JVM
(machine virtuelle Java),
indépendant de l'ordinateur hôte

Bytecode
UneClasse.class

R. Grin

Introduction à Java

10

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** ?

R. Grin

Introduction à Java

11

Exécution avec *java*

- `java HelloWorld`
exécute le *bytecode* de la méthode **main** de la classe **HelloWorld**

R. Grin

Introduction à Java

12

Le paramètre de *java* est un nom de classe

- `java HelloWorld`
HelloWorld est un nom de classe et pas un nom de fichier
- Donc
 - on ne peut pas donner un chemin
 - pas de suffixe **.class**

R. Grin

Introduction à Java

13

Où doit se trouver le fichier **.class** ?

- `java HelloWorld`
HelloWorld.class doit se trouver dans le *classpath*
- *classpath* donné par l'option **-classpath** :
`java -classpath rep1/rep2 HelloWorld`
- Par défaut c'est le répertoire courant

R. Grin

Introduction à Java

14

Votre environnement de développement

- Éditeur de texte (*emacs* ou autre ; indentation automatique obligatoire ; pas gedit !)
- Compilateur (*javac*)
- Interpréteur de *bytecode* (*java*)
- Javadoc des API du JDK (sous navigateur Web)
- Après l'étude des paquetages (interdit avant le TP 4 !), *IDE* (Environnement de Développement Intégré) conseillé ; NetBeans, Eclipse,...
- Autoformation à l'IDE choisi

R. Grin

Introduction à Java

15

3 éditions de Java

- **Java SE** : *Java 2 Standard Edition* ; la version de ce cours est la version 7
- **Java EE** : *Enterprise Edition*, enrichie pour des applications installées sur les serveurs
- **Java ME** : *Micro Edition*, pour des programmes embarqués (carte à puce, téléphone portable)

R. Grin

Introduction à Java

16

Compléments sur la compilation et l'exécution

R. Grin

Introduction à Java

17

Variables d'environnement

- **PATH** : doit inclure le répertoire qui contient les utilitaires Java (**javac**, **java**, **javadoc**,...)
- **Configurez cette variable avant le 1^{er} TP dans vos fichiers de configuration !**
- Évitez la variable CLASSPATH

R. Grin

Introduction à Java

18

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));
    }
}

```

Fichier ?.java

 $(x-p.x)^2 + (y-p.y)^2$

Message envoyé à p1 :
« ça Qu'est-ce que ça fait ?
te sépare de p2 »

R. Grin

Introduction à Java

Une classe Point

```

/** 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));
    }
    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

R. Grin

Introduction à Java

20

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 ?

R. Grin

Introduction à Java

21

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 ?

R. Grin

Introduction à Java

22

2 classes dans 1 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));
    }
}
/** 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

R. Grin

Introduction à Java

23

2 classes dans 1 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));
    }
}
/** 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 ?

java TestPoint

R. Grin

Introduction à Java

24

2 classes dans 2 fichiers

```
/** 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 **Point.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));
    }
}
```

Fichier **TestPoint.java**

R. Grin

Introduction à Java

25

Compilation des 2 fichiers

- On peut compiler les 2 fichiers :
`javac Point.java TestPoint.java`
 (ou `javac *.java`)
- Compiler `TestPoint.java` suffit :
`javac TestPoint.java`
- A-t-on besoin de `Point.java` pour compiler `TestPoint.java` ?
- Non, on n'a besoin que de `Point.class`

R. Grin

Introduction à Java

26

Chargement dynamique des classes

- Durant l'exécution, les classes sont chargées dans la JVM au fur et à mesure des besoins

R. Grin

Introduction à Java

27

Notions de base sur la programmation objet

R. Grin

Introduction à Java

28

Langage orienté objet

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

R. Grin

Introduction à Java

29

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 (nom du client, date,...)

R. Grin

Introduction à Java

30

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

R. Grin

Introduction à Java

31

Notion d'objet en Java

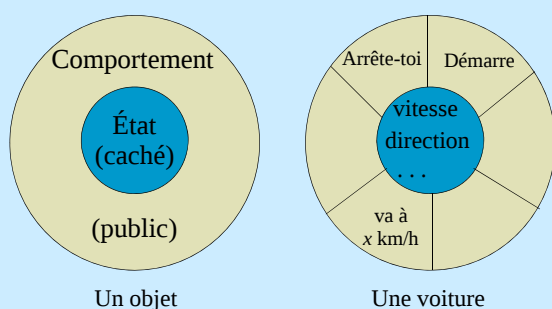
- Un objet a
 - une adresse en mémoire (identifie l'objet)
 - un **comportement** (ou **interface**)
 - un **état interne**
- Le comportement est donné par des fonctions ou procédures, appelées **méthodes**
- L'état interne est donné par des valeurs de **variables**

R. Grin

Introduction à Java

32

Un objet



R. Grin

Introduction à Java

33

Interactions entre objets

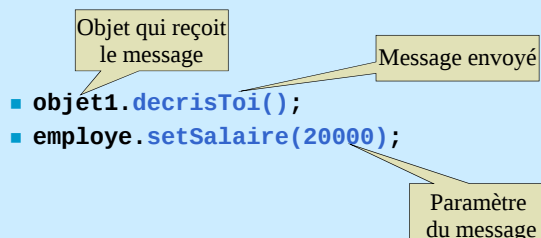
- Les objets interagissent en s'envoyant des **messages synchrones**
- Méthodes de la classe d'un objet ↔ Messages qu'on peut envoyer à l'objet
- Quand un objet reçoit un message, il exécute la méthode correspondante

R. Grin

Introduction à Java

34

Exemples de messages



R. Grin

Introduction à Java

35

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

R. Grin

Introduction à Java

36

D'autres paradigmes

- Programmation impérative (Pascal, C)
- Programmation fonctionnelle (Scheme, Lisp)

R. Grin

Introduction à Java

37

Langages objet

- C++
- C#
- Smalltalk

R. Grin

Introduction à Java

38

Les classes en Java

R. Grin

Introduction à Java

39

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

R. Grin

Introduction à Java

40

Éléments d'une classe

```
public class Livre {
    private String titre, auteur;
    private int nbPages;
    // Variables d'instance

    public Livre(String unTitre, String unAuteur) {
        titre = unTitre;
        auteur = unAuteur;
    }
    // Constructeurs

    public String getAuteur() { // accesseur
        return auteur;
    }
    // Méthodes

    public void setNbPages(int nb) { // modificateur
        nbPages = nb;
    }
}
```

R. Grin

Introduction à Java

41

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

R. Grin

Introduction à Java

42

Conventions pour les identificateurs

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

R. Grin

Introduction à Java

43

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 ...`
`*/`

R. Grin

Introduction à Java

44

Les constructeurs

R. Grin

Introduction à Java

45

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

R. Grin

Introduction à Java

46

Création d'une instance

```
public class Employe {
    private String nom, prenom; } variables d'instance
    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);
        ...
    }
}
```

R. Grin

Introduction à Java

47

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);
}
```

R. Grin

Introduction à Java

48

Plusieurs constructeurs (surcharge)

```
public class Employe {
    private String nom;
    private String 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);
```

R. Grin

Introduction à Java

49

Désigner un constructeur par **this()**

```
public class Employe {
    private String nom;
    private String 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);
```

R. Grin

Introduction à Java

50

Constructeur par défaut

- Lorsque le code d'une classe ne contient pas de constructeur, un constructeur est automatiquement ajouté par le compilateur
- Pour une classe **Classe**, ce constructeur sera :
`[public] Classe() { }`

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

R. Grin

Introduction à Java

51

Les méthodes

R. Grin

Introduction à Java

52

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

R. Grin

Introduction à Java

53

Types de méthode (2)

- Des méthodes **private** servent de « sous-programmes » utiles aux autres méthodes de la classe

R. Grin

Introduction à Java

54

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) {
        ...
    }
}
```

Classe qui décrit une tâche à accomplir

R. Grin

Introduction à Java

55

Surcharge d'une méthode

- Méthode qui a le même nom mais pas la même signature qu'une autre méthode :
calculerSalaire(int)
calculerSalaire(int, double)
- Interdit de surcharger en changeant seulement le type retour :
~~int calculerSalaire(int)~~
~~double calculerSalaire(int)~~

R. Grin

Introduction à Java

56

toString()

- Conseillé d'inclure une méthode **toString** dans toutes les classes
- Elle renvoie une **String** qui décrit l'instance
- Description **compacte et précise** pour être utile lors de la mise au point des programmes
- **System.out.println(objet)** affiche la valeur retournée par **objet.toString()**

R. Grin

Introduction à Java

page 57

Exemple

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

Livre [titre=Les Misérables, auteur=Victor Hugo, nbPages=320]

R. Grin

Introduction à Java

page 58

Les variables

R. Grin

Introduction à Java

59

Types de variables

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

R. Grin

Introduction à Java

60

Déclaration des variables

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

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

R. Grin

Introduction à Java

61

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

R. Grin

Introduction à Java

62

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;
```

R. Grin

Introduction à Java

63

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

R. Grin

Introduction à Java

64

Déclaration / création

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

R. Grin

Introduction à Java

65

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**

R. Grin

Introduction à Java

66

Accès aux membres d'une classe

R. Grin

Introduction à Java

67

Degrés d'encapsulation

- Pour les membres (variables et méthodes) et les constructeurs d'une classe
- **private** : seule la **classe** y a accès
- **public** : toutes les classes sans exception y ont accès
- Sinon, **par défaut**, seules les classes du même paquetage y ont accès

R. Grin

Introduction à Java

68

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**

R. Grin

Introduction à Java

69

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**

R. Grin

Introduction à Java

70

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

R. Grin

Introduction à Java

71

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

R. Grin

Introduction à Java

72

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

R. Grin

Introduction à Java

73

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 objet passe une référence de lui-même à un autre objet :

Comptable, calcule le salaire de moi

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

R. Grin

Introduction à Java

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 objet passe une référence de lui-même à un autre objet :

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

R. Gr

Dans quelle classe peut-on trouver ce code ?

Dans la classe **Employe**

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 objet passe une référence de lui-même à un autre objet :

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

Dans quelle classe peut-on trouver la méthode **calculeSalaire** ?

Dans la classe **Comptable**

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 objet passe une référence de lui-même à un autre objet :

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

Quelle sera la signature de la méthode **calculeSalaire** ?

ou **double calculeSalaire(Employe)**

Méthodes et variables de classe

R. Grin

Introduction à Java

78

Variable de classe (**static**)

- Partagée par toutes les instances d'une classe
- **Initialisation** dans la déclaration exécutée **une seule fois**, quand la classe est chargée en mémoire

R. Grin

Introduction à Java

79

Exemple de variable de classe

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

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

R. Grin

Introduction à Java

80

Méthode de classe (**static**)

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

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

R. Grin

Introduction à Java

81

Désigner une méthode de classe

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

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

R. Grin

Introduction à Java

82

Rappel : initialiser une variable d'instance

- En la déclarant :
`private int x = 10;`
- Dans un constructeur : `x = 10;`
- Si la variable est un tableau, on ne peut l'initialiser que dans un constructeur :

```
for (int i = 0; i < 25; i++) {
    tab[i] = -1;
}
```
- Comment initialiser un tableau **static** ?

R. Grin

Introduction à Java

83

Bloc d'initialisation **static**

- Pour les variables **static** complexes :

```
class UneClasse {
    private static int[] tab = new int[25];
    static {
        for (int i = 0; i < 25; i++) {
            tab[i] = -1;
        }
    }
    . . .
}
```

- Exécuté **une seule fois**, quand la classe est chargée en mémoire

R. Grin

Introduction à Java

84

Représentation graphique d'une classe en notation UML (*Unified Modeling Language*)

Cercle	Cercle
private Point centre private int rayon	- Point centre - int rayon
public Cercle(Point, int) public void setRayon(int) public int getRayon() public double surface()	+ Cercle(Point, int) + void setRayon(int) + int getRayon() + double surface()

(- : private, # : protected, + : public, \$ (ou souligné) : static)

R. Grin

Introduction à Java

85

Types de données en Java

R. Grin

Introduction à Java

86

Types de données en Java

- Types primitifs
- Objets (instances de classe)

R. Grin

Introduction à Java

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)

R. Grin

Introduction à Java

88

Erreurs de calculs

- $16.8 + 20.1$ donne **36.900000000000006**
- Pour les traitements de comptabilité utiliser la classe **java.math.BigDecimal**

R. Grin

Introduction à Java

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    // depuis Java 7
```

R. Grin

Introduction à Java

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 ; €)

R. Grin

Introduction à Java

91

Autres constantes

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

R. Grin

Introduction à Java

92

Opérateurs sur les types primitifs

- Les plus utilisés :

= + - * / % ++ -- += -= *= /=

== != > < >= <=

&& || ! (et, ou, négation)

Division entière si les
opérandes sont des entiers

R. Grin

Introduction à Java

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
```

R. Grin

Introduction à Java

94

Exemples d'utilisation des opérateurs

```
if (z == y)          // et pas un seul "=" !
    x += y;          // x = x + y
if (y != 0 && x / y > 8)
    // pas d'erreur si y = 0
```

R. Grin

Introduction à Java

95

Opérateur instanceof

o minuscule

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

R. Grin

Introduction à Java

96

Types des résultats des calculs avec des nombres

- Tout 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**

R. Grin

Introduction à Java

97

Traitement différent pour les objets et les types primitifs

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

R. Grin

Introduction à Java

98

Exemple d'utilisation des références

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

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

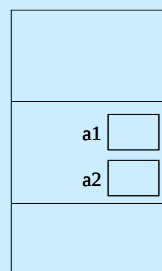
R. Grin

Introduction à Java

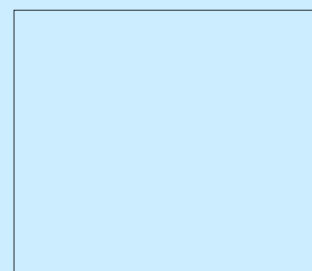
99

Références

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



Pile



Tas

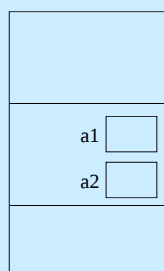
R. Grin

Introduction à Java

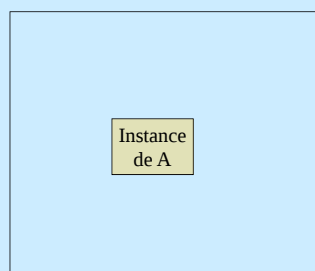
100

Références

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



Pile



Instance de A

Tas

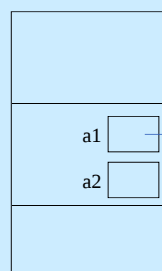
R. Grin

Introduction à Java

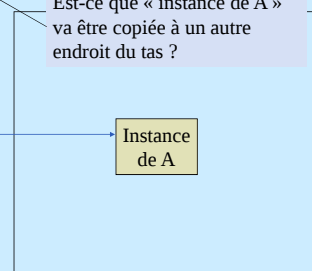
101

Références

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



Pile



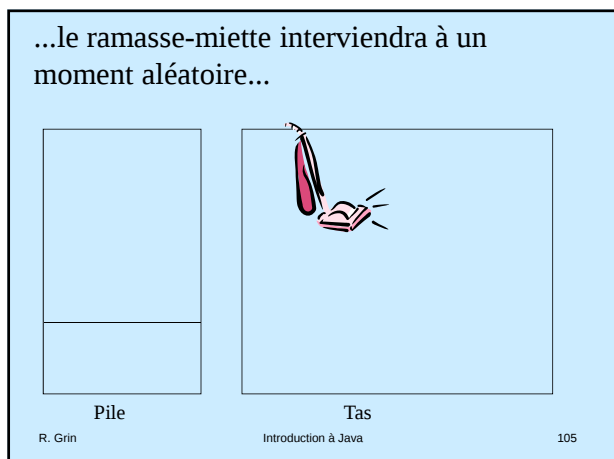
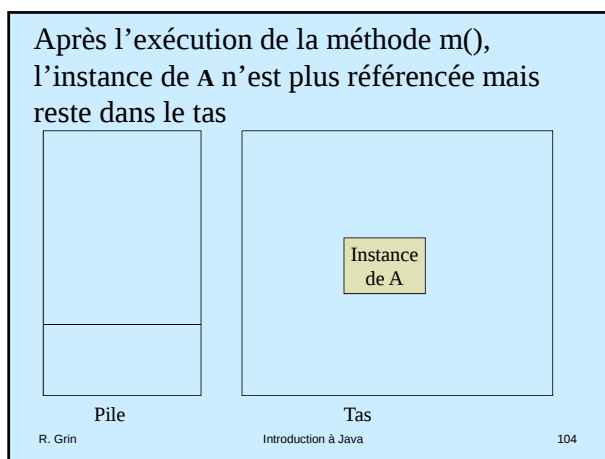
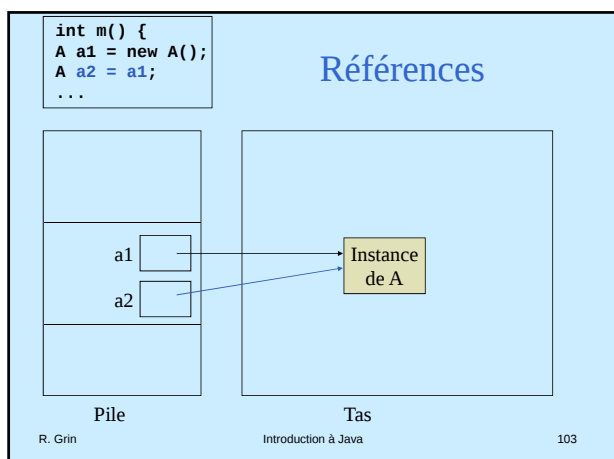
Tas

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

R. Grin

Introduction à Java

102



- ### Ramasse-miettes (*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
- R. Grin Introduction à Java 106

- ### Variable **final**
- Si la variable est d'un type primitif, sa valeur ne peut changer :
`static final double PI = 3.14;`
 - 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
```
- R. Grin Introduction à Java 107

- ### Forcer un type en Java
- Parfois nécessaire de forcer le programme 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
- R. Grin Introduction à Java 108

Casts autorisés

- En Java, 2 seuls cas sont autorisés pour les casts :
 - entre types primitifs
 - entre classe mère et classe fille

Étudié dans le cours
sur l'héritage

R. Grin

Introduction à Java

109

Casts entre types primitifs

- Une affectation entre types primitifs peut utiliser un cast *implicite* si elle ne peut pas provoquer de perte de valeur
- Par exemple, un **short** peut être affecté à un **int**
- Sinon, elle doit comporter un cast explicite (erreur à la compilation si pas de cast)

R. Grin

Introduction à Java

110

Exemples de Casts

- `byte b1 = 20;`
`byte b2 = 15;`
`byte b3 = b1 + b2;`
- `byte b3 = (byte) (b1 + b2);`
- `float f = 1.2;`
- `float f = 1.2f;`

R. Grin

Introduction à Java

111

Problèmes de Casts

- Un cast explicite peut donner un résultat aberrant *sans aucun message d'erreur* :

```
int i = 130;
b = (byte)i; // b = -126 !
```

R. Grin

Introduction à Java

112

Types énumérés

R. Grin

Introduction à Java

113

Types énumérés

- Type défini en énumérant toutes ses valeurs possibles

R. Grin

Introduction à Java

114

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);
```

R. Grin

Introduction à Java

115

## Énumération externe à une classe

```
public enum CouleurCarte { CouleurCarte.java
 TREFLE, CARREAU, COEUR, PIQUE;
}

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

R. Grin

Introduction à Java

116

## Utilisables avec ==

```
CouleurCarte couleurCarte;
...
if(couleurCarte == CouleurCarte.PIQUE)
```

R. Grin

Introduction à Java

117

## Utilisables dans un switch

```
switch(couleurCarte) {
 case PIQUE:
 break;
 case TREFLE:
 break;
 default:
 ...
}
```

Ne pas préfixer par **CouleurCarte**

R. Grin

Introduction à Java

118

## Tableaux

R. Grin

Introduction à Java

119

## Les tableaux sont des objets

- Créés par **new**
- Les variables contiennent des **références** aux tableaux
- Une variable d'instance :  

```
public final int length
```

R. Grin

Introduction à Java

120

## Mais des objets particuliers

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

R. Grin

Introduction à Java

121

## Déclaration et création des tableaux

- **Déclaration** : la taille n'est pas fixée  
`int[] tabEntiers;`
- **Création** : on **doit** donner la taille  
`tabEntiers = new int[5];`  
 Chaque élément du tableau reçoit la valeur par défaut du type de base du tableau
- La taille ne peut pas être modifiée

R. Grin

Introduction à Java

122

## 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")
};
```

R. Grin

Introduction à Java

123

## 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};
```

R. Grin

Introduction à Java

124

## Tableaux - utilisation

- **Affectation** des éléments ; l'indice commence à  
**0** et se termine à `tabEntiers.length - 1`  
`tabEntiers[0] = 12;`

- **Taille** du tableau

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

`ArrayIndexOutOfBoundsException`

- **Déclarations dans la signature d'une méthode**

```
int[] m(String[] t)
```

R. Grin

Introduction à Java

125

## 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
affiche ?
toto
bibi
```

R. Grin

Introduction à Java

126

OK ?

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

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

Création employé

R. Grin

Introduction à Java

127

## 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

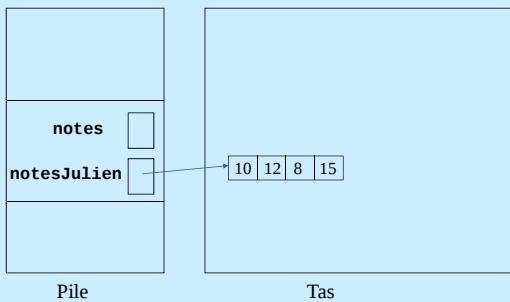
R. Grin

Introduction à Java

128

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

## Tableaux dans le tas



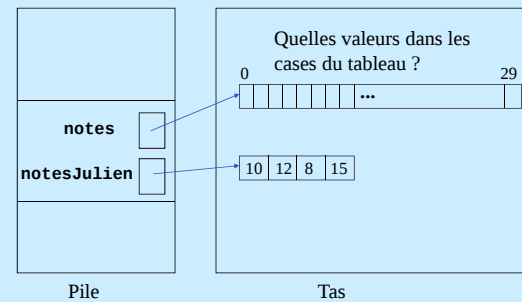
R. Grin

Introduction à Java

129

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

## Tableaux dans le tas



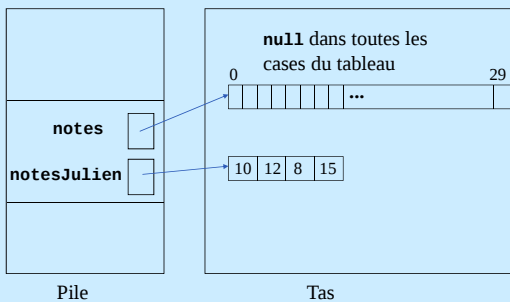
R. Grin

Introduction à Java

130

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

## Tableaux dans le tas



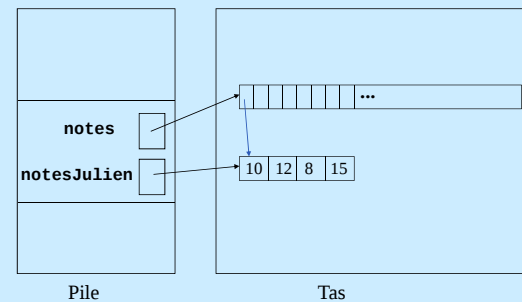
R. Grin

Introduction à Java

131

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

## Tableaux dans le tas



R. Grin

Introduction à Java

132

## 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;
```

R. Grin

Introduction à Java

133

## Exemple

```
int[][] t;
t = new int[2][];
int[] t0 = {0, 1};
t[0] = t0;
t[1] = new int[] {2, 3, 4, 5};
for (int i = 0; i < t.length; i++) {
 for (int j = 0; j < t[i].length; j++) {
 System.out.print(t[i][j] + " ");
 }
 System.out.println();
}
```

Déclaration

Création/affectation

On peut affecter un tableau entier

Que sera-t-il affiché ?

R. Grin

Introduction à Java

134

## Exemple

```
int[][] t;
t = new int[2][];
int[] t0 = {0, 1};
t[0] = t0;
t[1] = new int[] {2, 3, 4, 5};
for (int i = 0; i < t.length; i++) {
 for (int j = 0; j < t[i].length; j++) {
 System.out.print(t[i][j] + " ");
 }
 System.out.println();
}
```

```
0; 1;
2; 3; 4; 5;
```

R. Grin

Introduction à Java

135

## Afficher les valeurs d'un tableau

- Utile pour la mise au point : méthodes **static String toString** de la classe **Arrays**
- **Arrays.toString(t)** retourne **[12.5, 134.76]**
- Pour les tableaux à plusieurs dimensions : **Arrays.deepToString(t)**

R. Grin

Introduction à Java

136

## Classes de base

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

R. Grin

Introduction à Java

137

## 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

R. Grin

Introduction à Java

138

## Affectation d'une valeur littérale

- Syntaxe particulière pour **String** :  
`chaîne = "Bonjour";`

R. Grin

Introduction à Java

139

## Nouvelle affectation avec les **String**

```
String chaîne = "Bonjour";
chaîne = "Hello";
```

Objet **String**  
pas modifié

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

R. Grin

Introduction à Java

140

## 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

R. Grin

Introduction à Java

141

## Égalité de **Strings**

- La méthode **equals** teste si 2 **Strings** 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

R. Grin

Introduction à Java

142

## Comparaison de **Strings**

- **s.compareTo(t)**  
 retourne un entier qui a le « signe de **s-t** »
- **"bonjour".compareTo("les amis")**  
 renvoie une valeur de quel signe ?

R. Grin

Introduction à Java

143

## Quelques méthodes de **String**

- Extraire une sous-chaîne : **substring**
- Rechercher des sous-chaînes :  
**indexOf**  
**lastIndexOf**
- **startsWith**, **endsWith**,  
**toUpperCase**, **toLowerCase**
- et beaucoup d'autres...

R. Grin

Introduction à Java

144

## Méthodes pour extraire un caractère

- Les caractères d'une **String** sont numérotés de **0** à **length() - 1**
- Avant Java 5, « **char charAt(int i)** » pour extraire le *i*<sup>ème</sup> **char**
- Depuis Java 5, il vaut mieux éviter de travailler avec les **char** car, pour certaines langues, un caractère peut nécessiter 2 **char** ; **charAt** peut alors ne ramener qu'une partie d'un caractère

R. Grin

Introduction à Java

145

## Expressions régulières

- Étudiées dans le cours sur les entrées-sorties

R. Grin

Introduction à Java

146

## Chaînes modifiables

- **StringBuilder** possède des méthodes qui modifient le receveur du message et évitent la création de nouvelles instances :
  - **append** et **appendCodePoint**
  - **insert**
  - **replace**
  - **delete**
- Attention, **StringBuilder** ne redéfinit pas **equals**

R. Grin

Introduction à Java

147

## 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**

R. Grin

Introduction à Java

148

## Exemple

```
String[] t;
...
StringBuilder sb = new StringBuilder(t[0]);
for (int i = 1; i < t.length; i++) {
 sb.append(t[i]);
}
String chaine = sb.toString();
```

Que fait ce code ?

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

R. Grin

Introduction à Java

149

## A retenir !

- La création d'instances est **coûteuse**
- Il faut essayer de l'éviter

R. Grin

Introduction à Java

150

## Classes enveloppes de type primitif - « Autoboxing/Unboxing »

R. Grin

Introduction à Java

151

## 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 : **new Integer(8)**
- Les instances de ces classes ne sont pas modifiables

R. Grin

Introduction à Java

152

## Facilités des classes enveloppes

- Méthodes pour faire des conversions avec les types primitifs (et avec la classe **String**)
- Constantes, en particulier, **MAX\_VALUE** et **MIN\_VALUE**

R. Grin

Introduction à Java

153

## Conversion en entier d'une **String**

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

R. Grin

Introduction à Java

154

## Listes et types primitifs

- Une liste ne peut contenir de type primitif et on doit écrire :

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

R. Grin

Introduction à Java

155

## 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 peut écrire :
 

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

R. Grin

Introduction à Java

156

## Blocs d'instructions

R. Grin

Introduction à Java

157

## Interdit de cacher une variable locale

- En Java 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 += 1;
 }
 int init = 3;
}
```

**Interdit !**

R. Grin

Introduction à Java

158

## Instructions de contrôle

R. Grin

Introduction à Java

159

## if ... else

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

Quelle valeur pour **type** si y = 2 ?

Un bloc serait préférable

R. Grin

Introduction à Java

160

## 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;
```

R. Grin

Introduction à Java

161

## 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 de type énumération, ou **String**

R. Grin

Introduction à Java

162

## 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++;
}
```

Avec les lettres de  
« attention »,  
quelles valeurs auront  
les variables ?

R. Grin

Introduction à Java

163

## Répétitions « tant que »

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

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

Boucle  
exécutée au  
moins une fois

R. Grin

Introduction à Java

164

## 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);
 }
}
```

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

R. Grin

Introduction à Java

165

## Répétition for

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

est équivalent à

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

R. Grin

Introduction à Java

166

## Exemple de for

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

R. Grin

Introduction à Java

167

## « for each »

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

R. Grin

Introduction à Java

168

## Parcours d'un tableau

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

R. Grin

Introduction à Java

169

## Instructions liées aux boucles

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

R. Grin

Introduction à Java

170

## 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 ?

R. Grin

Introduction à Java

171

## Compléments sur les méthodes

R. Grin

Introduction à Java

172

## 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, et pas l'objet

R. Grin

Introduction à Java

173

## Exemple de passage de paramètres

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

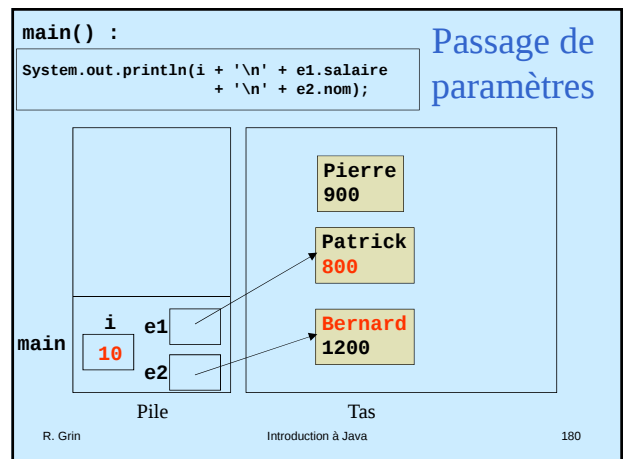
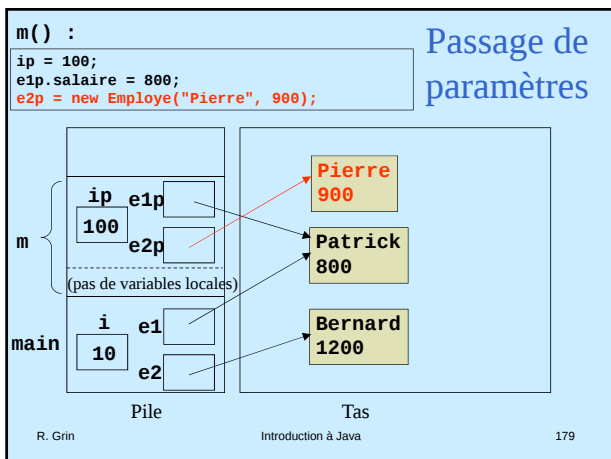
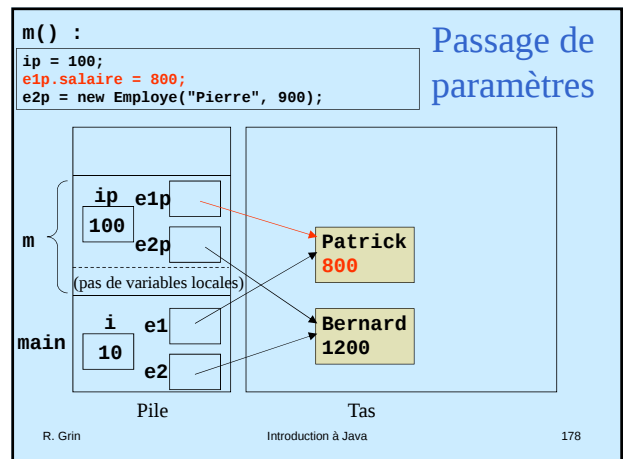
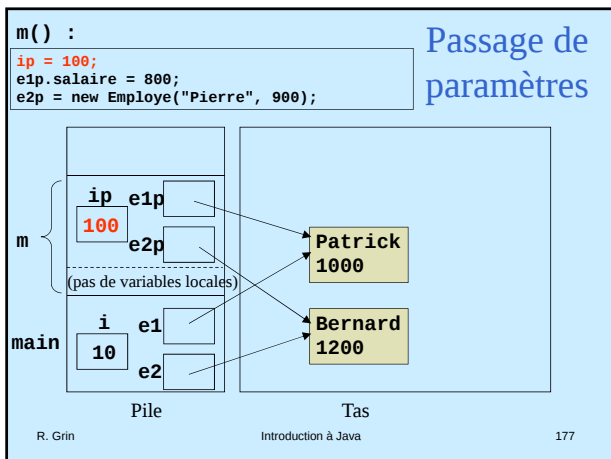
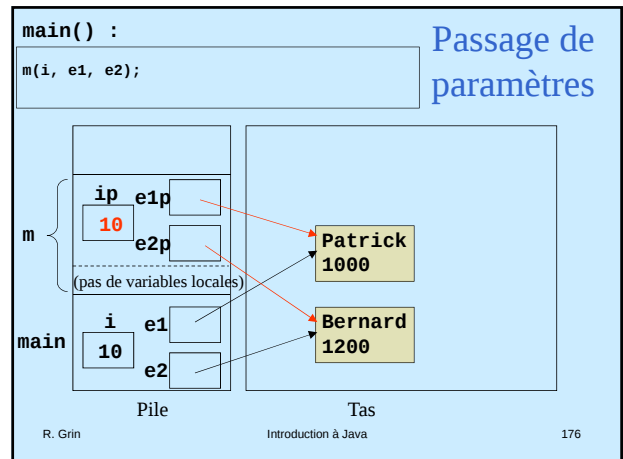
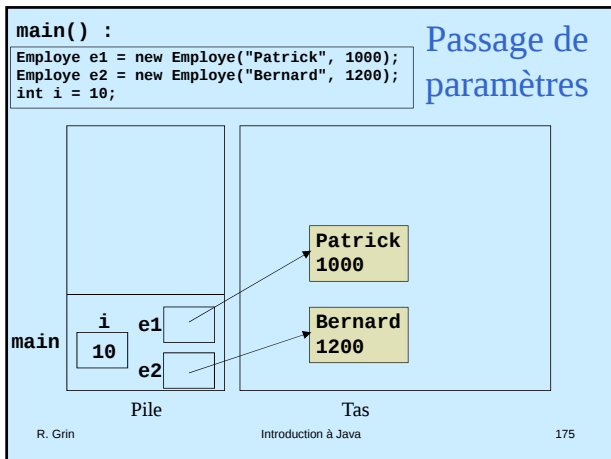
Que  
sera-t-il  
affiché ?

10  
800.0  
Bernard

R. Grin

Introduction à Java

174



## 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 Employee e1)`

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

R. Grin

Introduction à Java

181

## Nombre variable d'arguments

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

R. Grin

Introduction à Java

182

## 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;
}
```

Amélioration pour éviter une erreur à l'exécution ?

R. Grin

Introduction à Java

183

## Exemple amélioré

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

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

R. Grin

Introduction à Java

184

## Contrôle du compilateur pour la valeur de retour

- Ne compile pas :  

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

R. Grin

Introduction à Java

185

## Réversivité des méthodes

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

R. Grin

Introduction à Java

186

## Paquetages

R. Grin

Introduction à Java

187

## Définition d'un paquetage

- Regroupement de classes
- Niveau de modularité supplémentaire pour
  - réunir des classes suivant un centre d'intérêt commun
  - la protection des attributs et des méthodes

R. Grin

Introduction à Java

188

## Quelques paquetages du JDK

- **java.lang** : classes de base de Java
- **java.util** : utilitaires, collections
- **java.io** : entrées-sorties
- **java.awt** : interface graphique

R. Grin

Introduction à Java

189

## Nommer une classe

- Le **nom complet** d'une classe (*qualified name*) est le nom de la classe préfixé par le nom du paquetage :  
**java.util.ArrayList**
- Une classe **du même paquetage** peut être désignée par son nom « terminal » (**ArrayList**)
- Une classe d'un autre paquetage doit être désignée par son nom complet

R. Grin

Introduction à Java

190

## Importer une classe d'un paquetage

- Permet de désigner une classe d'un autre paquetage par son nom terminal

```
import java.util.ArrayList;

public class Classe {
 ...
 ArrayList liste = new ArrayList();
}
```

- Seulement une facilité d'écriture

R. Grin

Introduction à Java

191

## Importer toutes les classes d'un paquetage

- On peut importer **toutes** les classes d'un paquetage :  
**import java.util.\*;**
- Les classes du paquetage **java.lang** sont **automatiquement** importées

R. Grin

Introduction à Java

192

## Importer des constantes **static**

- Pour alléger le code, on peut importer des variables ou méthodes statiques d'une classe : **import static**
- ```
import static java.lang.Math.*;

public class Machin {
    . . .
    x = max(sqrt(abs(y)), sin(y));
}
```
- À utiliser avec précaution pour ne pas nuire à la lisibilité du code

R. Grin

Introduction à Java

193

Ajout d'une classe dans un paquetage

- **package** *nom-paquetage*;
au début du fichier source de la classe
- Par défaut, une classe appartient au « **paquetage par défaut** » :
 - paquetage sans nom
 - toutes les classes situées dans le même répertoire y appartiennent

R. Grin

Introduction à Java

194

Sous-paquetage

- Un paquetage peut avoir des sous-paquetages
- Par exemple, **java.awt.event** est un sous-paquetage de **java.awt**
- L'importation des classes d'un paquetage n'importe pas les classes des sous-paquetages :

```
import java.awt.*;
import java.awt.event.*;
```

R. Grin

Introduction à Java

195

Nom d'un paquetage

- Conseillé de préfixer ses propres paquetages par son adresse Internet :
eg.ufe.toto.liste

R. Grin

Introduction à Java

196

Placement d'un paquetage

- Les fichiers **.class** doivent se situer dans l'arborescence d'un des répertoires du *classpath*
- à un endroit qui correspond au nom du paquetage
- Les classes de **eg.ufe.toto.liste** doivent se trouver dans un sous-répertoire **eg/ufe/toto/liste** d'un des répertoires du *classpath*

R. Grin

Introduction à Java

197

Encapsulation d'une classe dans un paquetage

- Si la définition de la classe commence par **public class** la classe est accessible de partout
- Sinon, la classe n'est accessible que depuis les classes du même paquetage

R. Grin

Introduction à Java

198

Compiler les classes d'un paquetage

```
javac -d rep Classe.java
```

- « -d » indique le répertoire *racine* où javac rangera les fichiers « .class »
- Le fichier .class d'une classe du paquetage **nom1.nom2** sera rangé dans le répertoire *rep/nom1/nom2*

R. Grin

Introduction à Java

199

Compiler les classes d'un paquetage

- L'option -d, est souvent associée à l'option -classpath :

```
javac -classpath rep -d rep Classe.java
```

R. Grin

Introduction à Java

200

Exécuter une classe d'un paquetage

- Donner le nom complet de la classe :

```
java p1.p2.C
```

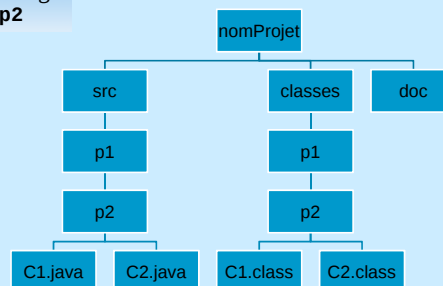
R. Grin

Introduction à Java

201

Placements préconisés

Paquetage
p1.p2



R. Grin

Introduction à Java

202

Commandes à lancer

- On se place dans le répertoire racine

– pour compiler :

```
javac -d classes -classpath classes  
-sourcepath src src/p1/p2/*.java
```

– pour exécuter :

```
java -classpath classes p1.p2.C1
```

R. Grin

Introduction à Java

203

classpath

R. Grin

Introduction à Java

204

Classpath

- Contient les endroits où trouver les classes :
 - répertoires
 - fichiers **.jar** ou **.zip**
 - **rep/*** désigne tous les fichiers **.jar** du répertoire **rep**

R. Grin

Introduction à Java

205

Exemples de Classpath

- Sous Unix, séparateur « : » :
`.:~/java/mespackages:~/mesutil/cl.jar`
- Sous Windows, séparateur « ; » :
`.;c:\java\mespackages;c:\mesutil\cl.jar`

R. Grin

Introduction à Java

206

Rappels : Quelques principes de programmation

R. Grin

Introduction à Java

207

Ce qu'il faut rechercher

- Une plus grande facilité de programmation
- Mais surtout
 - une maintenance plus aisée
 - une extensibilité accrue

R. Grin

Introduction à Java

208

Comment ?

- **Modularité** : décomposer en éléments plus simples
- **Encapsulation** : cacher ce qu'il n'est pas indispensable de voir
- **Lisibilité** : faciliter la compréhension des programmes
- **Réutilisabilité** : écrire des modules réutilisables dans les futurs développements (difficile)

R. Grin

Introduction à Java

209

Modularité

- Un programme est modulaire s'il est découpé en modules (plus ou moins) indépendants
- Un bon découpage doit satisfaire les 2 critères :
 - forte cohésion des éléments d'un module
 - faible couplage entre deux modules différents
- Ces 2 principes favorisent l'utilisation, la réutilisation et la maintenance des modules :
 - plus de souplesse : un module - une fonctionnalité
 - les modifications d'un module ont moins d'impacts sur les autres modules

R. Grin

Introduction à Java

210

Encapsulation

- Ne montrer et ne permettre de modifier que ce qui est nécessaire à une bonne utilisation
 - on montre l'interface (services offerts) d'un module
 - on cache l'implémentation (comment sont rendus les services)
- Avantages :
 - simplification de l'utilisation (la complexité ne dépend que de l'interface publique)
 - meilleure robustesse du programme
 - simplification de la maintenance de l'application

R. Grin

Introduction à Java

211

Attribution des fonctionnalités

- Comment choisir l'objet qui doit être le responsable de l'exécution d'une action ?
- Déterminer les informations nécessaires à l'exécution
- L'objet qui possède le plus d'information est le mieux placé pour exécuter l'action
- Localisation => modularité et encapsulation facilitées

R. Grin

Introduction à Java

212