

Java de base 2

Université de Nice - Sophia Antipolis

Version 7.2 – 24/3/12

Richard Grin

<http://deptinfo.unice.fr/~grin>

R. Grin

Introduction à Java

3

Plan de cette partie

- Types de données : types primitifs et non primitifs, le `tas`, types énumérés, tableaux, casts
- Classes de base
- Syntaxe du langage Java
- Paquetages
- Compléments sur *classpath*, `javac` et `java`
- Compléments sur le codage des caractères

R. Grin

Introduction à Java

2

Types de données en Java

R. Grin

Introduction à Java

3

Plan de la section « Types de données »

- Types primitifs et types objets, casts
- Types énumérés
- Tableaux

R. Grin

Introduction à Java

4

Types de données en Java

- Toutes les données manipulées par Java ne sont pas des objets
- 2 grands groupes de types de données :
 - types primitifs
 - objets (instances de classe)

R. Grin

Introduction à Java

5

Types primitifs

- **boolean** (`true/false`)
- Nombres entiers : **byte** (1 octet), **short** (2 octets), **int** (4 octets), **long** (8 octets)
- Nombres non entiers, à virgule flottante : **float** (4 octets), **double** (8 octets)
- Caractère (un seul) : **char** (2 octets) ; codé par le codage Unicode (et pas ASCII)

R. Grin

Introduction à Java

6

Caractéristiques des types numériques entiers

- **byte** : compris entre -128 et 127
- **short** : compris entre -32.768 et 32.767
- **int** : compris entre -2.147.483.648 et 2.147.483.647
- **long** : valeur absolue maximum $9,2 \times 10^{18}$ (arrondie)

R. Grin

Introduction à Java

7

Caractéristiques des types numériques non entiers

- **float** : environ 7 chiffres significatifs ; valeur absolue (arrondie) inférieure à $3,4 \times 10^{38}$ et précision maximum (arrondie) de $1,4 \times 10^{-45}$
- **double** : environ 17 chiffres significatifs ; valeur absolue inférieure à $1,8 \times 10^{308}$ (arrondie) et précision maximum de $4,9 \times 10^{-324}$ (arrondie)

R. Grin

Introduction à Java

8

Constantes nombres

- Une constante « entière » est de type **long** si elle est suffixée par « L » et de type **int** sinon
- Une constante « flottante » est de type **float** si elle est suffixée par « F » et de type **double** sinon

```
35
2589L          // constante de type long
456.7 ou 4.567e2 // 456,7 de type double
.123587E-25F  // de type float
012           // 12 en octal = 10 en décimal
0xA7         // A7 en hexa = 167 en décimal
```

R. Grin

Introduction à Java

9

Ajout de JDK 7

- Il est possible d'insérer le caractère « souligné » dans l'écriture des constantes nombres entiers : `1_567_688` ou `0x50_45_14_56`
- On peut aussi écrire un nombre en binaire : `0b010100000100010100010100010110` ou `0b01010000_01000101_00010100_01010110`

R. Grin

Introduction à Java

10

Constantes de type caractère

- Un caractère Unicode entouré par « ' »
- CR et LF interdits (caractères de fin de ligne)

```
'A'
'\t' '\n' '\r' '\\' '\'' '\"'
'\u20ac' (\u suivi du code Unicode hexadécimal
d'un caractère ; représente €)
'α'
```

R. Grin

Introduction à Java

11

Autres constantes

- Type booléen
 - **false**
 - **true**
- Référence inexistante (indique qu'une variable de type non primitif ne référence rien) ; convient pour *tous les types non primitifs*
 - **null**

R. Grin

Introduction à Java

12

Valeurs par défaut

- Si elles ne sont pas initialisées, les variables d'instance ou de classe (pas les variables locales d'une méthode) reçoivent par défaut les valeurs suivantes :

boolean	false
char	'\u0000'
Entier (byte short int long)	0 0L
Flottant (float double)	0.0F 0.0D
Référence d'objet	null

R. Grin

Introduction à Java

13

Opérateurs sur les types primitifs

- Voici les plus utilisés :

= + - * / % ++ -- += -= *= /=
== != > < >= <=
&& || ! (et, ou, négation)

Division entière si les opérandes sont des entiers

- **x++** : la valeur actuelle de **x** est utilisée dans l'expression et *juste après* **x** est incrémenté
- **++x** : la valeur de **x** est incrémentée et ensuite la valeur de **x** est utilisée
- **x = 1; y = x++ * x++; // y** a la valeur **6** (2 x 3) !

R. Grin

Introduction à Java

14

Exemples d'utilisation des opérateurs

```
int x = 9, y = 2;
int z = x/y;      // z = 4 (division entière)
z = x++ / y;     // z = 4 puis x = 10
z = --x / y;     // x = 9 puis z = 4
if (z == y)      // et pas un seul "=" !
    x += y;      // x = x + y
if (y != 0 && x/y > 8) // raccourci,
    // pas d'erreur si y = 0
    x = y = 3;   // y = 3, puis x = y,
                // (à éviter)
```

R. Grin

Introduction à Java

15

Types des résultats des calculs avec des nombres entiers

- Tout calcul entre entiers donne un résultat de type **int** (ou **long** si au moins un des opérandes est de type **long**)

R. Grin

Introduction à Java

16

Erreurs de calculs (1)

- Les types numériques « flottants » (non entiers) respectent la norme IEEE 754
- Malgré toutes les précautions prises, on ne peut empêcher les erreurs de calculs dues au fait que les nombres décimaux sont stockés sous forme binaire dans l'ordinateur :
16.8 + 20.1 donne 36.9000000000000006
- Pour les traitements de comptabilité on utilisera la classe **java.math.BigDecimal**

R. Grin

Introduction à Java

17

Erreurs de calculs (2)

- **System.out.print(2000000000 + 2000000000);**
affiche
-294967296
- Car **2000000000 + 2000000000** est de type **int** et il y a dépassement de capacité du type (sans aucun message d'erreur affiché, ni à la compilation, ni à l'exécution !)
- La solution : suffixer un des 2 nombres par **L** (ou le caster en **long**)

Richard Grin

Types de données

18

Ordre de priorité des opérateurs

Postfixés	[] . (params) expr++ expr--
Unaires	++expr --expr +expr -expr ~ !
Création et cast	new (type)expr
Multiplicatifs	* / %
Additifs	+ -
Décalages bits	<< >> >>>
Relationnels	< > <= >= instanceof
Egalité	= !=
Bits, et	&
Bits, ou exclusif	^
Bits, ou inclusif	
Logique, et	&&
Logique, ou	
Conditionnel	? :
Affectation	= += -= *= /= %= &= ^= = <<= >>= >>>=

Les opérateurs d'égales priorités sont évalués de gauche à droite, sauf les opérateurs d'affectation, évalués de droite à gauche

R. Grin

Introduction à Java

19

Opérateur instanceof

o minuscule

- La syntaxe est :
objet instanceof nomClasse
- Exemple : `if (x instanceof Livre)`
- Le résultat est un booléen :
 - `true` si `x` est de la classe `Livre`
 - `false` sinon
- On verra des compléments sur `instanceof` dans le cours sur l'héritage

R. Grin

Introduction à Java

20

Utilisation des opérateurs sur les bits

```
final byte BIT2 = 2; // = 0000 0010
byte drapeaux = 5; // bits 1 et 3 positionnés
// 2ème bit à 1 (sans toucher aux autres bits) :
drapeaux |= BIT2; // | = ou logique
// 2ème bit à 0 (sans toucher aux autres bits) :
drapeaux &= ~BIT2; // ~ = inverse les bits ; & = et logique
// Inverse la valeur du 2ème bit (sans toucher aux
// autres bits) :
drapeaux ^= BIT2; // ^ = xor (1 ssi un seul des 2 à 1)
// Teste la valeur du 2ème bit :
if (drapeaux == (drapeaux | BIT2))
    System.out.println("Bit 2 positionné");
```

R. Grin

Introduction à Java

21

Décalages de bits

- `<< n` : décalage à gauche ; idem multiplication par 2^n
- `>> n` : décalage à droite avec extension de signe (les bits à gauche sont remplis avec le bit de signe) ; comme tous les entiers sont signés en Java, le signe (le bit le plus à gauche) est conservé ; idem division par 2^n
- `>>> n` : décalage à droite en considérant que le nombre n'est pas signé

R. Grin

Introduction à Java

22

Exemples de décalages de bits

- Plus rapide que `n / 2 : n >> 1`
- `13 << 2 = 52`
- `-13 << 2 = -52`
- `13 >> 2 = 3`
- `-13 >> 2 = -4` (c'est bien `-13 / 4` car `-4` est bien le plus grand entier tel que `-4 x 4 < -13`)
- `13 >>> 2 = 3`
- `-13 >>> 2 = 1073741820` (car pas d'extension de signe)

R. Grin

Introduction à Java

23

Traitement différent pour les objets et les types primitifs

- Java manipule différemment les types primitifs et les objets
- Les variables contiennent
 - des valeurs de types primitifs
 - des références aux objets

R. Grin

Introduction à Java

24

La pile et le tas

- L'espace mémoire alloué à une variable locale est situé dans la pile
- Si la variable est d'un type primitif, sa valeur est placée dans la pile
- Sinon la variable contient une référence à un objet ; la valeur de la référence est placée dans la pile mais l'objet référencé est placé dans le tas
- Lorsque l'objet n'est plus référencé, un « ramasse-miettes » (*garbage collector*, GC) libère la mémoire qui lui a été allouée

R. Grin

Introduction à Java

25

Exemple d'utilisation des références

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

- Que se passe-t-il lorsque la méthode `m()` est appelée ?

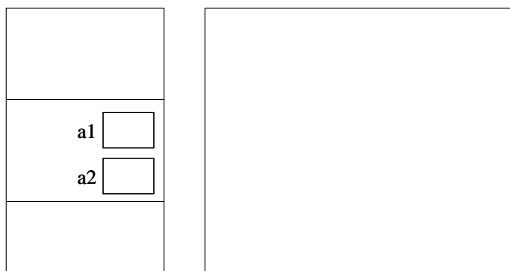
R. Grin

Introduction à Java

26

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

Références



Pile

Tas

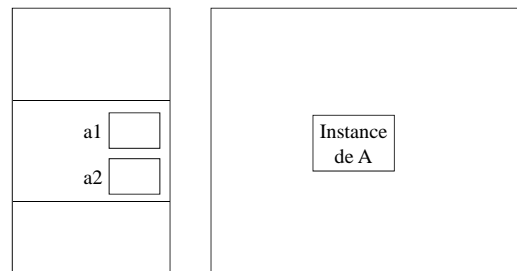
R. Grin

Introduction à Java

27

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

Références



Pile

Tas

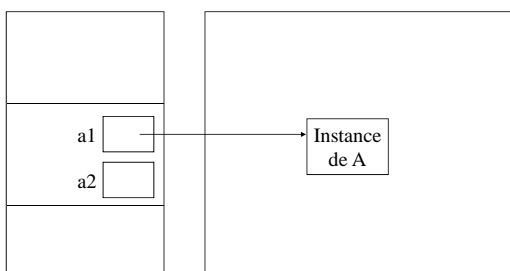
R. Grin

Introduction à Java

28

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

Références



Pile

Tas

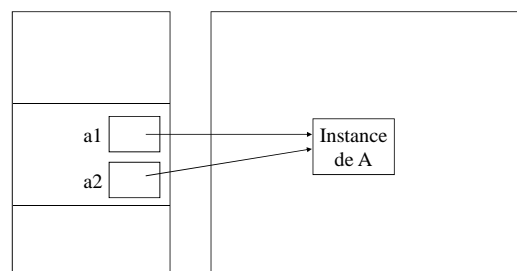
R. Grin

Introduction à Java

29

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

Références



Pile

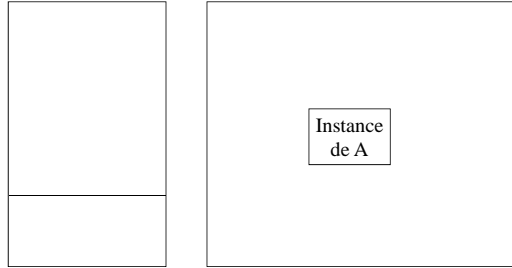
Tas

R. Grin

Introduction à Java

30

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

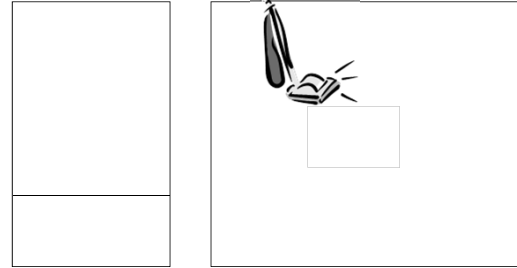


R. Grin

Introduction à Java

31

...le ramasse-miette interviendra à un
moment aléatoire...



R. Grin

Introduction à Java

32

Ramasse-miettes

- Le ramasse-miettes (*garbage collector*) est une tâche qui
 - libère la place occupée par les instances non référencées
 - compacte la mémoire occupée
- Il travaille en arrière-plan et intervient
 - quand le système a besoin de mémoire
 - ou, de temps en temps, avec une priorité faible

R. Grin

Introduction à Java

33

Modificateur **final**

- Le modificateur **final** indique que la valeur de la variable ne peut être modifiée : on pourra lui donner une valeur une seule fois dans le programme (voir les contraintes dans les transparents suivants)

R. Grin

Introduction à Java

34

Variable de classe **final**

- Une variable de classe **static final** est constante dans tout le programme ; exemple :
`static final double PI = 3.14;`
- Une variable de classe **static final** peut ne pas être initialisée à sa déclaration mais elle doit alors recevoir sa valeur dans un bloc d'initialisation **static**

R. Grin

Introduction à Java

35

Variable d'instance **final**

- Une variable *d'instance* (pas **static**) **final** est constante pour chaque instance ; mais elle peut avoir 2 valeurs différentes pour 2 instances
- Une variable d'instance **final** peut ne pas être initialisée à sa déclaration mais elle doit avoir une valeur à la sortie de tous les constructeurs

R. Grin

Introduction à Java

36

Variable locale **final**

- Une variable locale peut aussi être **final** : sa valeur ne pourra être donnée qu'une seule fois

R. Grin

Introduction à Java

37

Variable **final**

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

38

Forcer un type en Java

- Java est un langage fortement typé
- Dans certains cas, il est nécessaire de forcer le programme à considérer une expression comme étant d'un type qui n'est pas son type réel ou déclaré
- On utilise pour cela le *cast* (transtypage) :

(type-forcé) expression

```
int x = 10, y = 3;  
// on veut 3.3333.. et pas 3.0  
double z = (double)x / y; // cast de x suffit
```

R. Grin

Introduction à Java

39

Casts autorisés

- En Java, 2 seuls cas sont autorisés pour les *casts* :
 - entre types primitifs,
 - entre classes mère/ancêtre et classes filles

Détaillé lors de l'étude de l'héritage, plus loin dans ce cours

R. Grin

Introduction à Java

40

Cast entre types primitifs

- Un *cast* entre types primitifs peut occasionner une perte de données :
conversion d'un **int** vers un **short**
- Un *cast* peut provoquer une simple perte de précision :
la conversion d'un **long** vers un **float** peut faire perdre des chiffres significatifs mais pas l'ordre de grandeur

R. Grin

Introduction à Java

41

Cast implicite 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** sans nécessiter de *cast* explicite :

```
short s = ...;  
int i = s;
```

R. Grin

Introduction à Java

42

Cast explicite entre types primitifs

- Si une affectation peut provoquer une perte de valeur, elle doit comporter un *cast* explicite :

```
int i = ...;  
short s = (short)i;
```
- L'oubli de ce *cast* explicite provoque une erreur à la compilation
- Il faut écrire « `float f = 1.2f;` » et pas « `float f = 1.2;` »

R. Grin

Introduction à Java

43

Exemples de casts (1)

- `short s = 1000000;` // erreur !
- Cas particulier d'une affectation statique (repérable par le compilateur) d'un `int` « petit » :
`short s = 65;` // pas d'erreur
- Pour une affectation non statique, le *cast* explicite est obligatoire :

```
int i = 60;  
short b = (short)(i + 5);
```
- Les *casts* de types « flottants » vers les types entiers tronquent les nombres :
`int i = (int)1.99;` // `i = 1`, et pas 2

R. Grin

Introduction à Java

44

Exemples de casts (2)

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

 provoque une erreur (`b1 + b2` est un `int`)
- La dernière ligne doit s'écrire  
`byte b3 = (byte)(b1 + b2);`

R. Grin

Introduction à Java

45

## Problèmes de casts (1)

- Une simple perte de précision ne nécessite pas de *cast* explicite, mais peut conduire à des résultats comportant une erreur importante :  

```
long l1 = 123456789;
long l2 = 123456788;
float f1 = l1;
float f2 = l2;
System.out.println(f1); // 1.23456792E8
System.out.println(f2); // 1.23456784E8
System.out.println(l1 - l2); // 1
System.out.println(f1 - f2); // 8 !
```

R. Grin

Introduction à Java

46

## Problèmes de casts (2)

- Attention, dans le cas d'un *cast* explicite, la traduction peut donner un résultat totalement aberrant sans aucun avertissement ni message d'erreur :

```
int i = 130;
b = (byte)i; // b = -126 !
int c = (int)1e+30; // c = 2147483647 !
```

R. Grin

Introduction à Java

47

## Casts entre entiers et caractères

- Ils font correspondre un entier et un caractère qui a comme code Unicode la valeur de l'entier
- La correspondance `char` → `int`, `long` s'obtient par *cast* implicite
- Le code d'un `char` peut aller de 0 à 65.535 donc `char` → `short`, `byte` nécessite un *cast* explicite (`short` ne va que jusqu'à 32.767)
- Les entiers sont signés et pas les `char` donc `long`, `int`, `short` ou `byte` → `char` nécessite un *cast* explicite

R. Grin

Introduction à Java

48

## Types énumérés

R. Grin

Introduction à Java

49

## Types énumérés

- Ils ont été ajoutés par le JDK 5.0
- Ils permettent de définir un nouveau type en énumérant toutes ses valeurs possibles (par convention, les valeurs sont en majuscules)
- Plus sûrs que d'utiliser des entiers pour coder les différentes valeurs du type (vérifications à la compilation)
- Utilisés comme tous les autres types

R. Grin

Introduction à Java

50

## Exemple d'erreur sans énumération

- Une méthode `setDate(int, int, int)`
- Que signifie `setDate(2010, 5, 8)` ?
- 8 mai 2010 ? 5 août 2010 ?
- Il est facile d'inverser les significations des paramètres si on utilise cette méthode
- Problème sérieux si on veut indiquer la date de largage du satellite !
- Ce type d'erreur sera repéré à la compilation si on crée un type énuméré pour les mois

R. Grin

Introduction à Java

51

## Énumération « interne » à une classe

- On peut définir une énumération à l'intérieur d'une classe :

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

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

- Depuis une autre classe :

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

R. Grin

Introduction à Java

52

## Énumération « externe » à une classe

- En fait les types énumérés sont des classes qui héritent de la classe `java.lang.Enum`
- Comme les classes ils peuvent être définis indépendamment d'une classe
- Le fichier qui contient une énumération publique doit avoir le nom de l'énumération avec le suffixe « .java »

R. Grin

Introduction à Java

53

## Exemple d'énumération externe

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

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

R. Grin

Introduction à Java

54

## Les valeurs

- `toString()` retourne le nom de la valeur sous forme de `String` ; par exemple, `CouleurCarte.TREFLE.toString()` retourne "TREFLE"
- `static valueOf(String)` renvoie la valeur de l'énumération correspondant à la `String`
- La méthode `static values()` retourne un tableau contenant les valeurs de l'énumération ; le type des éléments du tableau est l'énumération

R. Grin

Introduction à Java

55

## Utilisable avec ==

- Dans le code de la classe `Carte` :

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

Obligatoirement  
préfixé par  
`CouleurCarte`

R. Grin

Introduction à Java

56

## Utilisable dans un switch

- Dans le code de la classe `Carte` :

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

Il ne faut pas  
préfixer par  
`CouleurCarte`

R. Grin

Introduction à Java

57

## Compléments sur les types énumérés

- Comme les classes les énumérations peuvent comporter des méthodes et des constructeurs

R. Grin

Introduction à Java

58

## Exemple

- Associer des `String` aux valeurs :

```
public enum Couleur {
 TREFLE("Trèfle"), CARREAU("Carreau"),
 COEUR("Coeur"), PIQUE("Pique");
 private String couleur;
 Couleur(String couleur) {
 this.couleur = couleur;
 }
 public String toString() {
 return couleur;
 }
}
```

Constructeur  
implicitement  
**private**

R. Grin

Introduction à Java

59

## Exemple

```
public enum ValeurCarte {
 DEUX(2), TROIS(3), QUATRE(4), CINQ(5),
 SIX(6), SEPT(7), HUIT(8), NEUF(9), DIX(10),
 VALET(10), DAME(10), ROI(10), AS(11);
 private int valeur;
 ValeurCarte(int valeur) {
 this.valeur = valeur;
 }
 public int getValeur() {
 return valeur;
 }
}
```

R. Grin

Introduction à Java

60

## Autres compléments

- Une énumération peut implémenter une interface
- Toutes les énumérations implémentent automatiquement l'interface `Serializable` (voir cours sur les entrées-sorties)
- Il n'est pas possible d'hériter de la classe `Enum` ni d'une énumération

R. Grin

Introduction à Java

61

## Collections et énumérations

- 2 types de collections de `java.util` (voir cours sur les collections) sont particulièrement adaptés pour les énumérations :
  - `EnumMap` est une `Map` dont les clés ont leur valeur dans une énumération
  - `EnumSet` est un `Set` dont les éléments appartiennent à une énumération

R. Grin

Introduction à Java

62

## Classe `Enum` (1)

- Classe mère des énumérations qui fournit les fonctionnalités de base des énumérations
- Classe générique `Enum<E extends Enum<E>>` (ne pas chercher à comprendre pour le moment ; voir cours sur la généricité)
- En fait, le compilateur transforme une énumération en une classe invocation de `Enum`
- Par exemple, l'énumération `Couleur` est transformée en `Enum<Couleur>`

R. Grin

Introduction à Java

63

## Classe `Enum` (2)

- `Enum` implémente `Comparable` et `Serializable`
- Elle contient (entre autres) les méthodes publiques
  - `int ordinal()` retourne le numéro de la valeur (en commençant à 0)
  - `String name()`
  - `static valueOf(String)`

R. Grin

Introduction à Java

64

## Énumération et singleton

- La meilleure façon d'implémenter le pattern « singleton » (classe qui ne peut avoir qu'une seule instance) en Java est d'utiliser une énumération (livre « Effective Java, item 3 » :

```
public enum SingletonMachin {
 UNIQUE_INSTANCE;
 // Méthodes de la « classe »
 ...
}
```

R. Grin

Introduction à Java

65

## Tableaux

R. Grin

Introduction à Java

66

## Les tableaux sont des objets

- En Java les tableaux sont considérés comme des objets (dont la classe hérite de `Object`) :
  - les variables de type tableau contiennent des références aux tableaux
  - les tableaux sont créés par l'opérateur `new`
  - ils ont une variable d'instance (`final`) :  
`final int length`
  - ils héritent des méthodes d'instance de `Object`

R. Grin

Introduction à Java

67

## Mais des objets particuliers

- Cependant, Java a une syntaxe particulière pour
  - la déclaration des tableaux
  - leur initialisation

R. Grin

Introduction à Java

68

## Déclaration et création des tableaux

- Déclaration : la taille n'est pas fixée  
`int[] tabEntiers;`  
Déclaration « à la C » possible, mais pas recommandé :  
`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 (0 pour `int`)
- La taille ne pourra plus être modifiée par la suite

R. Grin

Introduction à Java

69

## Initialisation des tableaux

- On peut lier la déclaration, la création et l'initialisation d'un tableau ; sa longueur est alors calculée automatiquement d'après le nombre de valeurs données (attention, cette syntaxe n'est autorisée que 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

70

## 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

71

## Tableaux - utilisation

- Affectation des éléments ; l'indice commence à 0 et se termine à `tabEntiers.length - 1`  
`tabEntiers[0] = 12;`
- Taille du tableau  
`int l = tabEntiers.length;  
int e = tabEntiers[l]; /* Lève une  
ArrayIndexOutOfBoundsException */`
- Déclarations dans la signature d'une méthode  
`int[] m(String[] t)`

R. Grin

Introduction à Java

72

## 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

73

## Afficher les éléments d'un tableau

- La méthode `toString()` héritée de `Object` sans modification, n'affiche pas les éléments du tableau
- Une simple boucle, comme dans le transparent précédent suffit à les afficher
- Pour des besoins de mise au point il est encore plus simple d'utiliser les méthodes `static toString` de la classe `Arrays` (depuis JDK 5)
- `static String toString(double[] t)`, affiche `t` sous la forme `[12.5, 134.76]`

R. Grin

Introduction à Java

74

## Faute à éviter

- Utiliser les objets du tableau avant de les avoir créés

```
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

75

## Copier une partie d'un tableau dans un autre

- On peut copier les éléments un à un mais la classe `System` fournit une méthode `static` plus performante :

```
public static void
arraycopy(Object src, int src_position,
tableau destination, Object dst, int dst_position,
int length)
```

tableau source

indice du 1er élément copié

nombre d'éléments copiés

indice de dst où sera copié le 1er élément

c minuscule !

Introduction à Java

76

## Comparer 2 tableaux (1/2)

- On peut comparer l'égalité de 2 tableaux (au sens où ils contiennent les mêmes valeurs) en comparant les éléments un à un
- On peut aussi utiliser les méthodes `static` à 2 arguments de type tableau de la classe `Arrays` `java.util.Arrays.equals()` par exemple, `equals(double[] a, double[] a2)`
- Ne pas utiliser la méthode `equals` héritée de `Object`

R. Grin

Introduction à Java

77

## Comparer 2 tableaux (2/2)

- Si le tableau a plus d'une dimension, `equals` compare l'identité des tableaux internes ; si on veut comparer les valeurs contenues dans le tableau, quelle que soit la profondeur il faut utiliser `deepEquals(Object[], Object[])`

R. Grin

Introduction à Java

78

## Autres méthodes de **Arrays**

- Méthodes **static** avec des variantes surchargées pour types des éléments du tableau
- **sort** trie un tableau
- **binarySearch** recherche (par dichotomie) une valeur dans un tableau trié
- **fill** remplit un tableau avec une valeur
- **hashCode** retourne une valeur de hashCode pour un tableau (voir section sur **Object** dans le cours sur l'héritage)

R. Grin

Introduction à Java

79

## Autres méthodes de **Arrays**

- **copyOf** copie un tableau en le tronquant ou le complétant avec des valeurs par défaut
- **copyOfRange** renvoie un nouveau tableau qui contient une partie d'un tableau, éventuellement complété avec des valeurs par défaut
- **asList** retourne une liste (voir cours sur les collections) dont les valeurs sont celles d'un tableau
- **hashCode** retourne une valeur de hash code d'un tableau

R. Grin

Introduction à Java

80

## 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][];
```

Il faut donner au moins les premières dimensions

Chacun des 30 étudiants a au plus 3 notes

Chacun des 30 étudiants a un nombre de notes variable

R. Grin

Introduction à Java

81

## 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

82

## 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();
}
```

On peut affecter un tableau entier

R. Grin

Introduction à Java

83

## Méthodes de **Arrays** pour les tableaux à plusieurs dimensions

- **deepEquals** : **equals** adapté aux tableaux à plusieurs dimensions
- **deepToString** : retourne une représentation sous la forme d'une chaîne de caractères d'un tableau à plusieurs dimensions (**toString** ne renvoie pas le contenu des tableaux inclus)
- **deepHashCode** retourne une valeur de hash code d'un tableau à plusieurs dimensions

R. Grin

Introduction à Java

84

## Classe d'un tableau

- La classe d'un tableau à 2 dimensions contenant des instances de **Object** (déclaré **Object[][]**) est désignée par **[Ljava.lang.Object**
- Pour un tableau à 1 dimension d'instances de **String**, on aurait **[Ljava.lang.String**
- Pour un tableau à 1 dimension d'entiers de type **int**, on aurait **[I** (**ID**, par exemple, pour un tableau de **double**)
- On utilise rarement ces classes

R. Grin

Introduction à Java

85

## Tableau en type retour

- Soit une méthode **m** qui renvoie tableau, par exemple « **int[][] m(...)** »
- Dans le cas où la méthode **m** ne renvoie rien, il est meilleur de renvoyer un tableau de dimension 0 plutôt que **null**
- Le code qui utilisera **m** sera plus facile à écrire :  

```
for (int i=0; i < t.length; i++),
sans avoir besoin d'un test préalable :
if (t == null) ...
```

R. Grin

Introduction à Java

86

## Classes de base

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

R. Grin

Introduction à Java

87

## Chaînes de caractères

R. Grin

Introduction à Java

88

## Chaînes de caractères

- 3 classes du paquetage **java.lang** :
  - **String** pour les chaînes **constantes**
  - **StringBuilder** ou **StringBuffer** pour les chaînes variables
- On utilise le plus souvent **String**, sauf si la chaîne doit être fréquemment modifiée
- Commençons par **String**

R. Grin

Introduction à Java

89

## Affectation d'une valeur littérale

- L'affectation d'une valeur littérale à un **String** s'effectue par :  

```
chaîne = "Bonjour";
```
- La spécification de Java impose que  

```
chaîne1 = "Bonjour";
chaîne2 = "Bonjour";
```

 crée un seul objet **String** (référéncé par les 2 variables)
- ```
chaîne1 = "Bonjour";  
chaîne2 = new String("Bonjour");
```

 provoque la création d'une **String** inutile

new force la création d'une nouvelle chaîne

R. Grin

Introduction à Java

90

Nouvelle affectation avec les **String**

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

Cet objet **String** n'est pas modifié

- La dernière instruction correspond aux étapes suivantes :
 - 1) Une nouvelle valeur (*Hello*) est créée
 - 2) La variable **chaine** référence la nouvelle chaîne *Hello* (et plus l'ancienne chaîne *Bonjour*)
 - 3) La place occupée par la chaîne *Bonjour* pourra être récupérée à un moment ultérieur par le ramasse-miette

R. Grin

Introduction à Java

91

Concaténation de chaînes

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

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

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

- les types primitifs sont traduits par le compilateur
- les instances d'une classe sont traduites en utilisant la méthode **toString()** de la classe

R. Grin

Introduction à Java

92

Concaténation de chaînes

- S'il y a plusieurs **+** dans une expression, les calculs se font de gauche à droite

```
int x = 5;  
s = "Valeur de x + 1 = " + x + 1;  
s contient « Valeur de x + 1 = 51 »
```

```
s = x + 1 + "est la valeur de x + 1";  
s contient « 6 est la valeur de x + 1 »
```

- On peut utiliser des parenthèses pour modifier l'ordre d'évaluation :

```
s = "Valeur de x + 1 = " + (x + 1);
```

R. Grin

Introduction à Java

93

Égalité de **Strings**

- La méthode **equals** teste si 2 instances de **String** contiennent la même valeur :

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

- «**==**» teste si les 2 objets ont la même adresse en mémoire ; **il ne doit pas être utilisé pour comparer 2 chaînes**, même s'il peut convenir dans des cas particuliers
- **equalsIgnoreCase()** ignore la casse des lettres

R. Grin

Introduction à Java

94

Égalité de sous-chaînes

- **regionMatches(int debut, String autre, int debutAutre, int longueur)** compare une sous-chaîne avec une sous-chaîne d'une autre chaîne
- Une variante prend un premier paramètre booléen qui est vrai si on ne tient pas compte des majuscules et minuscules

R. Grin

Introduction à Java

95

Comparaison de **Strings**

- **s.compareTo(t)** renvoie le « signe de **s-t** » :
 - 0 en cas d'égalité de **s** et de **t**,
 - un nombre entier positif si **s** suit **t** dans l'ordre lexicographique
 - un nombre entier négatif sinon
- **compareToIgnoreCase(t)** est une variante qui ne tient pas compte de la casse des lettres

R. Grin

Introduction à Java

96

Quelques méthodes de **String** (1)

- Extraire une sous-chaîne avec `substring(int début, int fin)` et `substring(int début)` : la sous-chaîne commence au caractère d'indice `début`, et se termine juste avant le caractère d'indice `fin` (à la fin de la chaîne pour la version à un seul paramètre) ; on enlève `début` caractères au début et on garde `fin - début` caractères

R. Grin

Introduction à Java

97

Quelques méthodes de **String** (2)

- Rechercher l'emplacement d'une sous-chaîne :
`indexOf(String sousChaîne)`
`indexOf(String sousChaîne, int debutRecherche)`
idem pour `LastIndexOf`
- Autres : `startsWith`, `endsWith`, `trim` (enlève les espaces de début et de fin), `toUpperCase`, `toLowerCase`, `valueOf` (conversions en `String` de types primitifs, tableaux de caractères)

R. Grin

Introduction à Java

98

Exemples de méthodes de **String**

- Récupérer le protocole, la machine, le répertoire et le nom du fichier d'une adresse Web

```
http://truc.unice.fr/rep1/rep2/fichier.html
```

```
int n = adresse.indexOf(":"); // 4
String protocole = adresse.substring(0, n); // http
String reste = adresse.substring(n + 3);
n = reste.indexOf("/"); // 13 | rep1/rep2/fichier.html
String machine = reste.substring(0, n);
String chemin = reste.substring(n + 1);
int m = chemin.lastIndexOf("/"); // 9
String repertoire = chemin.substring(0, m);
String fichier = chemin.substring(m + 1);
```

R. Grin

Introduction à Java

99

String et **char**

- Avant Java 5 une `String` était formée de caractères de type `char`
- Java 5 est adapté à Unicode version 4.0 qui peut coder plus d'un million de caractères ; un caractère ne peut donc plus être représenté par un `char` (2 octets ne peuvent coder que $2^{16} = 65536$ valeurs)
- Voir l'annexe « Codage des caractères » à la fin de cette partie du cours

R. Grin

Introduction à Java

100

Méthodes pour extraire un caractère

- La méthode `char charAt(int i)` pour extraire le $i^{\text{ème}}$ `char` ($i = 0$ pour le 1^{er} `char`) d'une `String`
- Depuis Java 5, `int codePointAt(int i)` renvoie le code d'un caractère Unicode, même si le caractère Unicode est représenté par 2 `char` (si i désigne un *surrogate*, il doit être le premier `char` du couple ; voir annexe à la fin de cette partie et javadoc)

R. Grin

Introduction à Java

101

Méthodes pour extraire un caractère

- Le paramètre i de la méthode `charAt` se réfère au $i^{\text{ème}}$ `char` (et pas au $i^{\text{ème}}$ caractère)
- Si ce `char` est un *surrogate* (voir annexe à la fin de cette partie) on n'obtient qu'une partie d'un caractère Unicode !

R. Grin

Introduction à Java

102

Nombre de **char** et de caractères

- Les **char** d'une **String** sont numérotés de 0 à `length() - 1`
- `int codePointCount(int début, int fin)` renvoie le nombre de caractères Unicode compris entre les caractères de numéro **début** (compris) et **fin** (pas compris) (voir « Codage des caractères » à la fin de ce document)
- Le nombre de caractères Unicode dans une **String s** est donc la valeur retournée par `codePointCount(0, s.length())`

R. Grin

Introduction à Java

103

char[] et **String**

- Constructeur `String(char[])`
- `String chaine = "abc";`
est équivalent à :
`char[] data = { 'a', 'b', 'c' };`
`String chaine = new String(data);`
- `void getChars(int debut, int fin, char[] dst, int debutDst)`
cette méthode de **String** copie les caractères **debut** à **fin - 1** de la **String** dans un tableau de **char**, à partir de l'emplacement numéro **debutDst**

R. Grin

Introduction à Java

104

Expressions régulières

- Des méthodes de la classe **String** qui utilisent des expressions régulières ont été ajoutées dans le SDK 1.4 : **matches**, **replaceFirst**, **replaceAll** et **split**
- Elles sont étudiées dans le cours sur les entrées-sorties avec les classes dédiées aux expressions régulières

R. Grin

Introduction à Java

105

Chaînes modifiables

- **StringBuffer** ou **StringBuilder** possèdent des méthodes qui modifient le receveur du message et évitent la création de nouvelles instances ; par exemple :
 - **append** et **appendCodePoint**
 - **insert**
 - **replace**
 - **delete**
- Attention, **StringBuffer** et **StringBuilder** ne redéfinissent pas **equals()**

R. Grin

Introduction à Java

106

Différences entre **StringBuilder** et **StringBuffer**

- **StringBuilder** a été introduite par le JDK 5.0
- Mêmes fonctionnalités et noms de méthodes que **StringBuffer** mais ne peut être utilisé que par un seul **thread** (pas de protection contre les problèmes liés aux accès multiples)
- **StringBuilder** fournit de meilleures performances que **StringBuffer**

R. Grin

Introduction à Java

107

String et **String{Buffer|Builder}**

- Utiliser plutôt la classe **String** qui possède de nombreuses méthodes
- Si la chaîne de caractères doit être souvent modifiée, passer à **StringBuilder** avec le constructeur `StringBuilder(String s)`
- Repasser de **StringBuilder** à **String** avec `toString()`

R. Grin

Introduction à Java

108

Concaténation - performances

- Attention, la concaténation de **Strings** est une opération coûteuse (elle implique en particulier la création d'un **StringBuilder**)
- Passer explicitement par un **StringBuilder** si la concaténation doit se renouveler

R. Grin

Introduction à Java

109

Exemple

```
String[] t;
...
// Concaténation des éléments de t dans
// chaîne
StringBuilder sb = new StringBuilder(t[0]);
for (int i = 1; i < t.length; i++) {
    sb.append(t[i]);
}
String chaîne = c.toString();
```

R. Grin

Introduction à Java

110

A retenir !

- La création d'instances est coûteuse
- Il faut essayer de l'éviter (mais pas au prix de complications du code, sauf cas exceptionnels)

R. Grin

Introduction à Java

111

Interface **CharSequence**

- Suite de **char** lisibles (introduite par le SDK 1.4)
- Implémentée par **String**, **StringBuffer** et **StringBuilder** (et par **java.nio.CharBuffer** pour travailler sur des fichiers)
- Cette interface est utilisée pour les signatures des méthodes qui travaillent sur des suites de caractères (par exemple dans la classe **java.util.regex.Pattern**)
- Méthodes **charAt**, **length** et **subSequence** (pour extraire une sous-suite)

R. Grin

Introduction à Java

112

BreakIterator

- La classe abstraite **java.text.BreakIterator** permet de récupérer des parties d'un texte écrit en langage naturel
- On peut ainsi récupérer les caractères (pas si facile depuis que le type **char** ne suffit plus pour représenter un caractère), les mots, les phrases, les lignes
- Consultez la javadoc pour en savoir plus

R. Grin

Introduction à Java

113

Récupérer des phrases avec un **BreakIterator**

```
iter = BreakIterator.getSentenceInstance();
s = "Phrase 1. Phrase 2 ; phrase 3. Phrase 4
! Phrase 5.";
iter.setText(s);
int deb = iter.first();
int fin = iter.next();
while (fin != BreakIterator.DONE) {
    System.out.println(s.substring(deb, fin));
    deb = fin;
    fin = iter.next();
}
```

R. Grin

Introduction à Java

114

Classe `Collator`

- La classe `java.text.Collator` permet de comparer des chaînes de caractères « localisées » (adaptées à une langue particulière, par exemple le français)
- Par exemple, elle permet de trier suivant l'ordre du dictionnaire des mots qui contiennent des caractères accentués
- Cette classe n'est pas étudiée en détails dans ce cours

R. Grin

Introduction à Java

115

Classes enveloppes de type primitif - « Auto-boxing/unboxing »

R. Grin

Introduction à Java

116

Classes enveloppes des types primitifs

- En Java certaines manipulations nécessitent de travailler avec des objets (instances de classes) et pas avec des valeurs de types primitifs
- Le paquetage `java.lang` fournit des classes pour envelopper les types primitifs : `Byte`, `Short`, `Integer`, `Long`, `Float`, `Double`, `Boolean`, `Character`
- Attention, les instances de ces classes ne sont pas modifiables (idem `String`)

R. Grin

Introduction à Java

117

Méthodes utilitaires des classes enveloppes

- En plus de permettre la manipulation des valeurs des types primitifs comme des objets, les classes enveloppes offrent des méthodes utilitaires (le plus souvent `static`) pour faire des conversions avec les types primitifs (et avec la classe `String`)
- Elles offrent aussi des constantes, en particulier, `MAX_VALUE` et `MIN_VALUE`

R. Grin

Introduction à Java

118

- Les transparents suivants indiquent comment faire des conversions d'entiers ; il existent des méthodes similaires pour toutes types primitifs (`double` par exemple)

R. Grin

Introduction à Java

119

Conversions des entiers

- `String` → `int` : (méthode de `Integer`)
`static int parseInt(String ch
[,int base])`
- `int` → `String` (méthode de `String`) :
`static String valueOf(int i)`
- Autre moyen pour `int` → `String` :
concaténer avec la chaîne vide (par exemple,
`" " + 3`)

R. Grin

Introduction à Java

120

Conversions des entiers (classe `Integer`)

- `int` → `Integer` :
`new Integer(int i)`
- `Integer` → `int` :
`int intValue()`
- `String` → `Integer` :
`static Integer valueOf(String ch
[,int base])`
- `Integer` → `String` :
`String toString()`

R. Grin

Introduction à Java

121

Exemple de conversion

```
/** Afficher le double du nombre passé en
    paramètre */
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

122

Listes et types primitifs

- Le code est alourdi lorsqu'une manipulation nécessite d'envelopper une valeur d'un type primitif
- Ainsi on verra qu'une liste ne peut contenir de type primitif et on sera obligé d'écrire :
`liste.add(new Integer(89));`
`int i = liste.get(n).intValue();`

R. Grin

Introduction à Java

123

Boxing/unboxing

- Le « *autoboxing* » (mise en boîte) automatise le passage des types primitifs vers les classes qui les enveloppent
- Cette mise en boîte automatique a été introduite par la version 5 du JDK
- L'opération inverse s'appelle « *unboxing* »
- Le code précédent peut maintenant s'écrire :
`liste.add(89);`
`int i = liste.get(n);`

R. Grin

Introduction à Java

124

Autres exemples de *boxing/unboxing*

- `Integer a = 89;`
`a++;`
`int i = a;`
- `Integer b = new Integer(1);`
`// unboxing suivi de boxing`
`b = b + 2;`
- `Double d = 56.9;`
`d = d / 56.9;`

R. Grin

Introduction à Java

125

Unboxing et valeur `null`

- Une tentative de *unboxing* avec la valeur `null` va lancer une `NullPointerException`

R. Grin

Introduction à Java

126

Inconvénients de *boxing/unboxing*

- La conversion est tout de même effectuée entre `int` et `Integer`, mais c'est le compilateur qui la fait
- Le *boxing/unboxing* peut amener une perte de performances significative s'il est fréquent
- Et des comportements curieux car `==` compare
 - l'identité entre `Integer`
 - l'égalité de valeurs entre `int`

R. Grin

Introduction à Java

127

Exemples de bizarreries

```
Integer a = new Integer(1);
int b = 1;
Integer c = new Integer(1);
if (a == b)
    System.out.println("a = b");
else
    System.out.println("a != b");
if (b == c)
    System.out.println("b = c");
else
    System.out.println("b != c");
if (a == c)
    System.out.println("a = c");
else
    System.out.println("a != c");
```

`==` ne semble plus transitif car il s'affiche :
`a = b`
`b = c`
`a != c`

R. Grin

Introduction à Java

128

Bloc d'instructions

R. Grin

Introduction à Java

129

Programmation structurée

- Les méthodes sont structurées en blocs par les structures de la programmation structurée
 - suites de blocs
 - alternatives
 - répétitions
- Un bloc est un ensemble d'instructions délimité par `{` et `}`
- Les blocs peuvent être emboîtés les uns dans les autres

R. Grin

Introduction à Java

130

Portée des identificateurs

- Les blocs définissent la portée des identificateurs
- La portée d'un identificateur commence à l'endroit où il est déclaré et va jusqu'à la fin du bloc dans lequel il est défini, y compris dans les blocs emboîtés

R. Grin

Introduction à Java

131

Déclaration des variables locales - compléments

- Les variables locales peuvent être déclarées n'importe où dans un bloc (pas seulement au début)
- On peut aussi déclarer la variable qui contrôle une boucle « `for` » dans l'instruction « `for` » (la portée est la boucle) :

```
for (int i = 0; i < 8; i++) {
    s += valeur[i];
}
```

R. Grin

Introduction à Java

132

Interdit de cacher une variable locale

- Attention ! Java n'autorise pas la déclaration d'une variable dans un bloc avec le même nom qu'une variable d'un bloc emboîtant, ou qu'un paramètre de la méthode

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

Interdit !

R. Grin

Introduction à Java

133

Instructions de contrôle

R. Grin

Introduction à Java

134

Alternative « if » ou « if ... else »

if (*expressionBooléenne*)
bloc-instructions ou instruction

[else
bloc-instructions ou instruction] « else » facultatif

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

Un bloc serait préférable, même s'il n'y a qu'une seule instruction

R. Grin

Introduction à Java

135

if emboîtés

- Lorsque plusieurs **if** sont emboîtés les uns dans les autres, un bloc **else** se rattache au dernier bloc **if** qui n'a pas de **else** (utiliser les accolades pour ne pas se tromper)

R. Grin

Introduction à Java

136

Exemple

```
x = 3;  
y = 8;  
if (x == y)  
    if (x > 10)  
        x = x + 1;  
else  
    x = x + 2;
```

Quelle valeur pour x à la fin de ce code ?

Facile de se tromper si on ne met pas d'accolades, surtout si on indente mal son code !

R. Grin

Introduction à Java

137

Exemple

```
if (x == y) {  
    if (x > 10) {  
        x = x + 1;  
    }  
}  
else {  
    x = x + 2;  
}
```

Plus clair en mettant des accolades !

R. Grin

Introduction à Java

138

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;
```

Parenthèses pas indispensables

R. Grin

Introduction à Java

139

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** (depuis le JDK 7)

■ S'il n'y a pas de clause **default**, rien n'est exécuté si *expression* ne correspond à aucun **case**

R. Grin

Introduction à Java

140

Exemple 1 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++;  
}
```

R. Grin

Introduction à Java

141

Exemple 2 de switch

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

R. Grin

Introduction à Java

142

Répétitions « tant que »

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

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

Le bloc d'instructions est exécuté au moins une fois

R. Grin

Introduction à Java

143

Exemple : diviseur d'un nombre

```
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

144

Lecture d'une ligne entrée au clavier, caractère par caractère

```
public static String lireLigneClavier()
    throws IOException {
    String ligne = "";
    char c;
    do {
        c = (char)System.in.read();
        ligne += c;
    } while (c != '\n' && c != '\r' );
    return
        ligne.substring(0, ligne.length() - 1);
}
```

Juste un exemple de do while
Il y a d'autres solutions
pour faire ça

R. Grin

Introduction à Java

145

Répétition **for**

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

est équivalent à

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

R. Grin

Introduction à Java

146

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

147

« **for each** »

- Une nouvelle syntaxe introduite par la version 5 du JDK simplifie le parcours d'un tableau
- La syntaxe est plus simple/lisible qu'une boucle *for* ordinaire
- Attention, on ne dispose pas de la position dans le tableau (pas de « variable de boucle »)
- On verra par la suite que cette syntaxe est encore plus utile pour le parcours d'une « collection »

R. Grin

Introduction à Java

148

Parcours d'un tableau

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

R. Grin

Introduction à Java

149

Instructions liées aux boucles

- **break** sort de la boucle et continue après la boucle
- **continue** passe à l'itération suivante
- **break** et **continue** peuvent être suivis d'un nom d'étiquette qui désigne une boucle englobant la boucle où elles se trouvent (une étiquette ne peut se trouver que devant une boucle)

R. Grin

Introduction à Java

150

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

151

Étiquette de boucles

```
boucleWhile: while (pasFini) {
    ...
    for (int i = 0; i < t.length; i++) {
        ...
        if (t[i] < 0)
            continue boucleWhile;
        ...
    }
    ...
}
```

R. Grin

Introduction à Java

152

Compléments sur les méthodes

R. Grin

Introduction à Java

153

Contexte d'exécution

- Le contexte d'exécution d'une méthode d'instance est l'objet à qui est envoyé le message (le « *this* ») ; il comprend
 - les valeurs des variables d'instance
 - les valeurs des variables de la classe de l'objet (variables **static**)
- Le contexte d'exécution d'une méthode de classe comprend seulement les variables de classes

R. Grin

Introduction à Java

154

En-tête d'une méthode

[accessibilité] [static] type-ret nom([liste-param])

{
public
protected
private
}

{
void
int
Cercle
...
}

La méthode ne renvoie aucune valeur

```
public double salaire()
static int nbEmployes()
public void setSalaire(double unSalaire)
private int calculPrime(int typePrime, double salaire)
public int m(int i, int j, int k)
public Point getCentre()
```

Le type retourné peut être le nom d'une classe

R. Grin

Introduction à Java

155

Passage des arguments des méthodes

- Le passage se fait par valeur : les valeurs des arguments sont copiées dans l'espace mémoire de la méthode
- Attention, pour les objets, la valeur passée est une référence ; donc,
 - si la méthode modifie l'objet référencé par un paramètre, l'objet passé en argument sera modifié en dehors de la méthode
 - si la méthode change la valeur d'un paramètre (type primitif ou référence), ça n'a pas d'incidence en dehors de la méthode

R. Grin

Introduction à Java

156

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 ?

Il sera affiché
10
800.0
Bernard

R. Grin

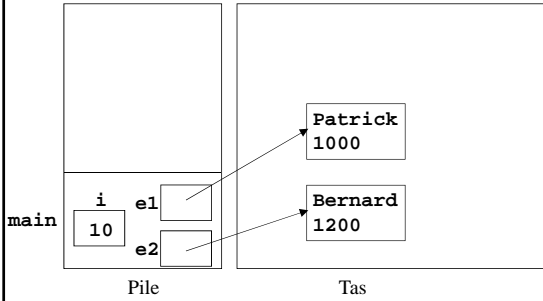
Introduction à Java

157

main() :

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

Passage de paramètres



R. Grin

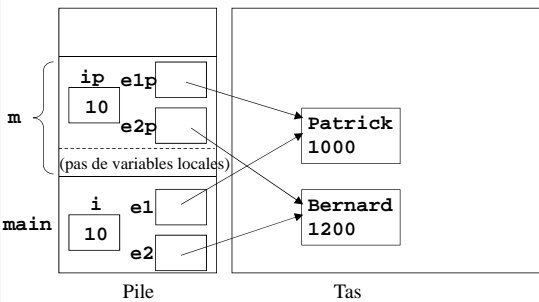
Introduction à Java

158

main() :

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

Passage de paramètres



R. Grin

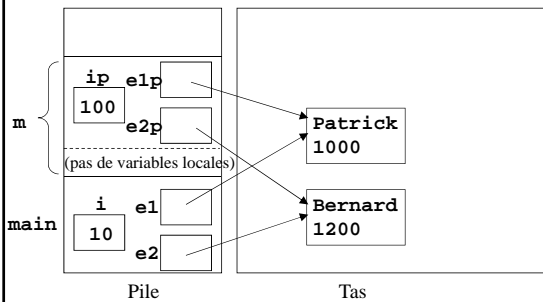
Introduction à Java

159

m() :

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

Passage de paramètres



R. Grin

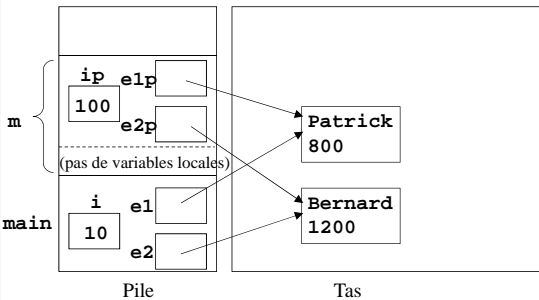
Introduction à Java

160

m() :

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

Passage de paramètres



R. Grin

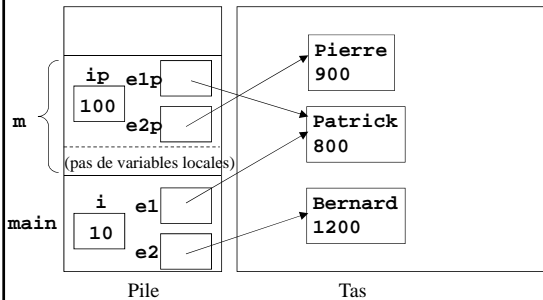
Introduction à Java

161

m() :

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

Passage de paramètres



R. Grin

Introduction à Java

162

Passage de paramètres

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

R. Grin Introduction à Java 163

Paramètre **final**

- **final** indique que le paramètre ne pourra être modifié dans la méthode
- Si le paramètre est d'un type primitif, la valeur du paramètre ne pourra être modifiée :
`int m(final int x)`
- Attention ! si le paramètre n'est pas d'un type primitif, la référence à l'objet ne pourra être modifiée mais l'objet lui-même pourra l'être
`int m(final Employe e1)` Le salaire de l'employé e1 pourra être modifié

R. Grin Introduction à Java 164

Nombre variable d'arguments

- Quelquefois il peut être commode d'écrire une méthode avec un nombre variable d'arguments
- L'exemple typique est la méthode *printf* du langage C qui affiche des arguments selon un format d'affichage donné en premier argument
- Depuis le JDK 5.0, c'est possible en Java

R. Grin Introduction à Java 165

Syntaxe pour arguments variables

- A la suite du type du dernier paramètre on peut mettre « ... » :
`String...`
`Object...`
`int...`

R. Grin Introduction à Java 166

Traduction du compilateur

- Le compilateur traduit ce type spécial par un type tableau :
`m(int p1, String... params)`
est traduit par
`m(int p1, String[] params)`
- Le code de la méthode peu utiliser `params` comme si c'était un tableau (boucle `for`, affectation, etc.)

R. Grin Introduction à Java 167

Exemple 1

```
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 168

Exemple 2

```
private int p1;
private String[] options;
Classe(int p1, String... params) {
    this.p1 = p1;
    this.options = params;
}
```

R. Grin

Introduction à Java

169

Remarque

- On peut passer un tableau en dernier argument ; les éléments du tableau seront considérés comme la liste d'arguments de taille variable attendue par la méthode

R. Grin

Introduction à Java

170

Retour de la valeur d'une méthode

- **return** sert à sortir d'une méthode en renvoyant une valeur (du type déclaré pour le type retour dans la définition de la méthode) :
`return i * j;`
`return new Cercle(p, x+y);`
- **return** sert aussi à sortir d'une méthode sans renvoyer de valeur (méthode ayant **void** comme type retour) :
`if (x == 0)`
`return;`

R. Grin

Introduction à Java

171

Contrôle du compilateur pour la valeur de retour

- Le compilateur détecte quand une méthode risque de ne pas retourner une valeur
- Il permet ainsi de découvrir des erreurs de programmation telles que la suivante :

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

R. Grin

Introduction à Java

172

Réversivité des méthodes

- Les méthodes sont récursives ; elles peuvent s'appeler elles-mêmes :

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

R. Grin

Introduction à Java

173

Paquetages

R. Grin

Introduction à Java

174

Définition d'un paquetage

- Les classes Java sont regroupées en paquetages (*packages* en anglais)
- Ils correspondent aux « bibliothèques » des autres langages comme le langage C, Fortran, Ada, etc...
- Les paquetages offrent un niveau de modularité supplémentaire pour
 - réunir des classes suivant un centre d'intérêt commun
 - la protection des attributs et des méthodes
- Le langage Java est fourni avec un grand nombre de paquetages

R. Grin

Introduction à Java

175

Quelques paquetages du SDK

- **java.lang** : classes de base de Java
- **java.util** : utilitaires, collections
- **java.io** : entrées-sorties
- **java.awt** : interface graphique
- **javax.swing** : interface graphique avancée
- **java.applet** : applets
- **java.net** : réseau
- **java.rmi** : distribution des objets

R. Grin

Introduction à Java

176

Nommer une classe

- Le nom complet d'une classe (*qualified name* dans la spécification du langage Java) 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 » (les classes du paquetage `java.util` peuvent désigner la classe ci-dessus par « `ArrayList` »)
- Une classe d'un autre paquetage doit être désignée par son nom complet

R. Grin

Introduction à Java

177

Importer une classe d'un paquetage

- Pour pouvoir désigner une classe d'un autre paquetage par son nom terminal, il faut l'importer

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

- On peut utiliser une classe sans l'importer ; l'importation permet seulement de raccourcir le nom d'une classe dans le code

R. Grin

Introduction à Java

178

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

179

Lever une ambiguïté (1/2)

- On aura une erreur à la compilation si
 - 2 paquetages ont une classe qui a le même nom
 - ces 2 paquetages sont importés en entier
 - la classe commune aux 2 paquetages est désignée sans son nom de paquetage
- Exemple (2 classes `List`) :

```
import java.awt.*;
import java.util.*;
...
List l = getListe();
```

R. Grin

Introduction à Java

180

Lever une ambiguïté (2/2)

- Pour lever l'ambiguïté, on devra donner le nom complet de la classe. Par exemple,

```
java.util.List l = getListe();
```

Importer des constantes **static**

- Depuis le JDK 5.0 on peut importer des variables ou méthodes statiques d'une classe ou d'une interface avec « **import static** »
- On allège ainsi le code, par exemple pour l'utilisation des fonctions mathématiques de la classe **java.lang.Math**
- A utiliser avec précaution pour ne pas nuire à la lisibilité du code (il peut être plus difficile de savoir d'où vient une constante ou méthode)

Exemple d'**import static**

A placer avec les autres import

```
import static java.lang.Math.*;
public class Machin {
    . . .
    x = max(sqrt(abs(y)), sin(y));
}
```

- On peut importer une seule variable ou méthode :

```
import static java.lang.Math.PI;
. . .
x = 2 * PI;
```

Ajout d'une classe dans un paquetage

- `package nom-paquetage;` doit être la première instruction du fichier source définissant la classe (avant même les instructions **import**)
- Par défaut, une classe appartient au « paquetage par défaut » qui n'a pas de nom, et auquel appartiennent toutes les classes situées dans le même répertoire (et qui ne sont pas dans un paquetage particulier)

Fichier contenant plusieurs classes

- L'instruction **package** doit être mise avant la première classe du fichier
- Toutes les classes du fichier sont alors mises dans le paquetage
- De même, les instructions **import** doivent être mises avant la première classe et importent pour toutes les classes du fichier
- Rappel : il n'est pas conseillé de mettre plusieurs classes dans un même fichier

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 ; on devra écrire par exemple :

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

Nom d'un paquetage

- Le nom d'un paquetage est hiérarchique :
`java.awt.event`
- Il est conseillé de préfixer ses propres paquetages par son adresse Internet :
`fr.unice.toto.liste`

(inutile pour les petites applications non diffusées)

R. Grin

Introduction à Java

187

Placement d'un paquetage

- Les fichiers `.class` doivent se situer dans l'arborescence d'un des répertoires du *classpath*
- Le nom relatif du répertoire par rapport au répertoire du *classpath* doit correspondre au nom du paquetage
- Par exemple, les classes du paquetage `fr.unice.toto.liste` doivent se trouver dans un répertoire `fr/unice/toto/liste` relativement à un des répertoires du *classpath*

R. Grin

Introduction à Java

188

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

189

Compiler les classes d'un paquetage

```
javac -d repertoire-racine Classe.java
```

- L'option « `-d` » permet d'indiquer le répertoire *racine* où sera rangé le fichier compilé
- Si on compile avec l'option « `-d` » un fichier qui comporte l'instruction « `package nom1.nom2` », le fichier `.class` est rangé dans le répertoire *repertoire-racine/nom1/nom2*

R. Grin

Introduction à Java

190

Compiler les classes d'un paquetage

- Si on compile sans l'option « `-d` », le fichier `.class` est rangé dans le même répertoire que le fichier `.java` (quel que soit le paquetage auquel appartient la classe) ; à éviter !

R. Grin

Introduction à Java

191

Compiler les classes d'un paquetage

- Quand on compile les classes d'un paquetage avec l'option `-d`, on doit le plus souvent indiquer où sont les classes déjà compilées du paquetage avec l'option `-classpath` :

```
javac -classpath repertoire-racine  
-d repertoire-racine Classe.java
```

R. Grin

Introduction à Java

192

Option `-sourcepath`

- javac peut ne pas savoir où sont les fichiers source de certaines classes dont il a besoin (en particulier si on compile plusieurs paquetages en même temps)
- L'option `-sourcepath` indique des emplacements pour des fichiers sources
- Comme avec `classpath`, cette option peut être suivie de fichiers jar, zip ou des répertoires racines pour les fichiers source (l'endroit exact où se trouvent les fichiers `.java` doit refléter le nom du paquetage)

R. Grin

Introduction à Java

193

Utilité de `-sourcepath`

- javac pourra ainsi retrouver les fichiers sources qui ne sont pas explicitement donnés dans la commande
- Ce qui permettra, par exemple, à javac de recompiler un fichier `.java` si sa date de dernière modification est plus récente que celle du fichier `.class` correspondant

R. Grin

Introduction à Java

194

Exemple avec `sourcepath`

- Situation : compiler une classe `C`, et toutes les classes dont cette classe dépend (certaines dans des paquetages pas encore compilés)
- « `javac C.java` » ne retrouvera pas les fichiers class des paquetages qui ne sont pas déjà compilés
- On doit indiquer où se trouvent les fichiers source de ces classes par l'option `-sourcepath` ; par exemple,

```
javac -sourcepath src -classpath classes  
-d classes C.java
```

R. Grin

Introduction à Java

195

Problème avec `sourcepath`

- Si `C.java` a besoin d'un paquetage pas encore compilé et dont les sources sont indiquées par l'option `sourcepath`
- Si ce paquetage comporte une erreur qui empêche sa compilation javac envoie un message d'erreur indiquant que le paquetage n'existe pas, sans afficher de message d'erreur de compilation pour le paquetage
- L'utilisateur peut alors penser à tort que l'option `sourcepath` ne marche pas

R. Grin

Introduction à Java

196

Exécuter une classe d'un paquetage

- On lance l'exécution de la méthode `main` d'une classe en donnant le nom complet de la classe (préfixé par le nom de son paquetage)
- Par exemple, si la classe `C` appartient au paquetage `p1.p2` :

```
java p1.p2.C
```
- Le fichier `C.class` devra se situer dans un sous-répertoire `p1/p2` d'un des répertoires du `classpath` (option `-classpath` ou variable `CLASSPATH`)

R. Grin

Introduction à Java

197

Utilisation pratique des paquetages

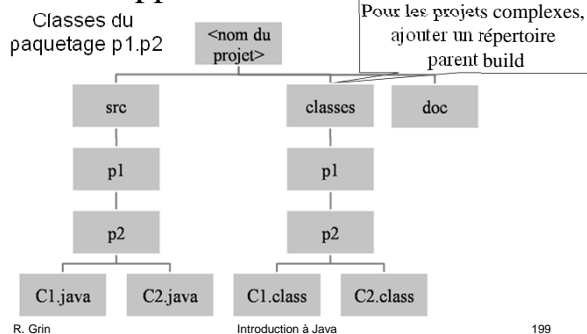
- Les premières tentatives de développement avec paquetages conduisent à de grosses difficultés pratiques pour compiler les classes
- Ces difficultés peuvent être évitées
 - en respectant quelques principes simples pour le placement des fichiers sources et classes (exemple dans les transparents suivants)
 - en utilisant correctement les options `-classpath`, `-d` et `-sourcepath`

R. Grin

Introduction à Java

198

Placements préconisés pour le développement d'une application



Commandes à lancer

- Si on se place dans le répertoire racine,
 - pour compiler (*en une seule ligne*) :

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

```
java -classpath classes p1.p2.C1
```
 - pour générer la documentation :

```
javadoc -d doc -sourcepath src p1.p2
```
- On peut ajouter d'autres répertoires ou fichier .jar dans le *classpath*

R. Grin Introduction à Java 200

Classes inter-dépendantes

- Si 2 classes sont inter-dépendantes, il faut les indiquer toutes les deux dans la ligne de commande java de compilation :

```
javac ... A.java B.java
```

ou

```
javac ... *.java
```

R. Grin Introduction à Java 201

Utilisation des fichiers source

- javac recherche les classes dont il a besoin dans le *classpath*
- S'il ne trouve pas le fichier *.class*, mais s'il trouve le fichier *.java* correspondant, il le compilera pour obtenir le *.class* cherché
- S'il trouve les 2 (*.class* et *.java*), il recompilera la classe si le *.java* est plus récent que le *.class*

R. Grin Introduction à Java 202

Ant et Maven

- Pour le développement d'applications complexes, il vaut mieux s'appuyer sur un utilitaire de type *make*
- L'utilitaire Ant, très évolué, et très utilisé par les développeurs Java est spécialement adapté à Java (un autre cours étudie Ant)
- Maven est très utilisé pour les projets importants ; il permet de gérer les dépendances entre projets et fournit des archétypes pour les types de projet les plus courants

R. Grin Introduction à Java 203

Recherche des classes par la JVM

R. Grin Introduction à Java 204

Chemin de recherche des classes

- Les outils *java* et *javac* recherchent toujours d'abord dans les fichiers système qui sont placés dans le répertoire dit `<java-home>`, le répertoire dont le nom commence par `jre`
- Ces fichiers système sont les suivants :
 - fichiers `rt.jar` et `i18n.jar` dans le répertoire `jre/lib` où *java* a été installé,
 - fichiers `.jar` ou `.zip` dans le sous-répertoire `jre/lib/ext`
- Ils regardent ensuite dans le *classpath*

R. Grin

Introduction à Java

205

Classpath (1)

- Le *classpath* contient par défaut le seul répertoire courant (il est égal à `<< . >>`)
- Si on donne une valeur au *classpath*, on doit indiquer explicitement le répertoire courant si on veut qu'il y soit

R. Grin

Introduction à Java

206

Classpath (2)

- Le *classpath* indique les endroits où trouver les classes et interfaces :
 - des répertoires (les classes sont recherchées sous ces répertoires selon les noms des paquetages)
 - des fichiers `.jar` ou `.zip`
 - depuis le JDK 6, `<< * >>` indique que tous les fichiers `.jar` placés directement sous le répertoire sont inclus ; par exemple, `lib/*`

R. Grin

Introduction à Java

207

Classpath (3)

- L'ordre dans lequel les entrées sont données dans le *classpath* est important
- La recherche d'une classe se termine dès qu'elle a été trouvée dans une des entrées, en commençant par les entrées les plus à gauche

R. Grin

Introduction à Java

208

Exemples de Classpath

- Sous Unix, le séparateur est `<< : >>` :
`.:~/java/mespackages:~/mesutil/cl.jar`
- Sous Windows, le séparateur est `<< ; >>` :
`.;c:\java\mespackages;c:\mesutil\cl.jar`

R. Grin

Introduction à Java

209

Bibliothèques « *endorsed* »

- Certaines bibliothèques de programmes bien définies sont utilisées en interne par *java*, sans être gérées par le mécanisme JCP qui régit le développement de *Java*
- Par exemple, les bibliothèques liées à XML
- L'exécutable *Java* (JRE) contient une version de ces bibliothèques, pas toujours la plus récente
- Un mécanisme permet l'utilisation des dernières versions de ces bibliothèques

R. Grin

Introduction à Java

210

Mécanisme « *endorsed* »

- Il suffit de déposer des fichiers jar de ces bibliothèques dans le répertoire `<java-home>/lib/endorsed` pour que ces fichiers jar soient utilisés par java à la place des versions embarquées dans le JDK
- La propriété système `java.endorsed.dirs` permet de donner d'autres répertoires à la place de ce répertoire
- Placer ces fichiers jar ailleurs, par exemple dans le `classpath`, ne fonctionne pas

R. Grin

Introduction à Java

211

Compléments sur javac et java

R. Grin

Introduction à Java

212

Quelques autres options de javac

- `-help` affiche les options standard
- `-x` affiche les options non standard
- `-verbose` donne des informations sur la compilation
- `-nowarn` n'affiche pas les messages d'avertissement (tels que les messages sur les problèmes de codage de caractères)
- `-deprecation` donne plus d'information sur l'utilisation de membres ou classes déconseillés

R. Grin

Introduction à Java

213

Quelques autres options de javac

- `-target` permet d'indiquer la JVM avec laquelle sera exécuté le programme
- `-source` spécifie la version de java des codes source ; par exemple, `-source 1.4` permet l'utilisation des assertions du SDK 1.4
- `-g` ajoute des informations dans les fichiers classes pour faciliter la mise au point
- `-encoding` spécifie le codage des caractères du code source ; par exemple `-encoding ISO-8859-1`

R. Grin

Introduction à Java

214

Option `-Xlint`

- Ajoutée par le JDK 5, cette option affiche des avertissements qui sont peut-être (mais pas nécessairement) des erreurs de programmation
- Un avertissement n'empêche pas la génération des fichiers `.class`
- `-Xlint` peut être suivi d'un type d'avertissement (les autres ne sont pas rapportés) ; par exemple, l'option `-Xlint:fallthrough` avertit si une entrée d'un switch ne se termine pas par un break

R. Grin

Introduction à Java

215

D'autres options `-Xlint`

- `-Xlint:unchecked` donne plus d'informations sur les problèmes de conversions (liés à la généricité ; voir cours sur la généricité)
- `-Xlint:path` avertit si un chemin est manquant (`classpath`, `sourcepath`,...)
- `-Xlint:serial` avertit si une classe sérialisable ne contient pas de `serialVersionUID` (peut nuire à la maintenance)

R. Grin

Introduction à Java

216

Option « verbeuse » de javac

- L'option **-verbose** affiche les noms
 - des classes chargées en mémoire
 - des fichiers source compilés

R. Grin

Introduction à Java

217

Informations pour la mise au point

- L'option **-g** de javac permet d'inclure dans les fichiers classe des informations utiles pour la mise au point des programmes ; « **-g** » seul inclut les informations sur les fichiers source (**source**) et sur les numéros de ligne (**lines**)
- On peut aussi ajouter les informations sur les variables locales (**vars**)
- Exemple : `javac -g:source,lines,vars ...`
- On peut enlever ces informations avec l'option « **-g:none** »

R. Grin

Introduction à Java

218

JVM cible

- L'option **-target** permet d'indiquer la JVM cible pour les fichiers classes générés
- Au moment où ces lignes sont écrites, c'est la JVM du SDK 1.2 qui est la cible par défaut (et donc aussi toutes celles qui sont plus récentes)
- On peut en choisir une autre ; par exemple :
`javac -target 1.4 . . .`

R. Grin

Introduction à Java

219

Options de java

- **-Dpropriété=valeur** permet de donner la valeur d'une propriété utilisée par le programme (voir cours sur les propriétés)
- **-version** affiche la version de java utilisée et s'arrête
- **-showversion** affiche la version de java utilisée et continue
- **-jar** permet d'exécuter un fichier jar (voir cours sur les fichiers jar)

R. Grin

Introduction à Java

220

Option « verbeuse » de java

- L'option **-verbose** affiche divers informations qui peuvent être utiles pour mettre au point les programmes
- Elle affiche
 - les noms des classes chargées en mémoire
 - des événements liés au ramasse-miettes
 - des informations sur l'utilisation des bibliothèques natives (pas écrites en Java)

R. Grin

Introduction à Java

221

Unknown source

- Parfois un message d'erreur n'affiche pas la ligne qui a provoqué l'erreur mais le message « *Unknown source* »
- Pour faire apparaître le numéro de ligne on a vu qu'il faut compiler la classe où a lieu l'erreur avec l'option de java « **-g** »
- Dans les anciennes versions de java, il fallait lancer java avec l'option **-Djava.compiler=none**

R. Grin

Introduction à Java

222

Annexe : codage des caractères

R. Grin

Introduction à Java

223

- Voici les modifications apportées à Java pour prendre en compte les dernières versions d'Unicode (à partir de Java 5)

R. Grin

Introduction à Java

224

Le codage des caractères

- En interne Java utilise le codage Unicode pour coder ses caractères (il n'utilise pas le code ASCII)
- Avant la version 5, les caractères étaient codés sur 2 octets avec le type primitif `char`
- Unicode (à partir de la version 4.0) code maintenant plus de 65.536 caractères, le type `char` ne suffit plus et tout se complique un peu

R. Grin

Introduction à Java

225

Unicode

- Java 7 code les caractères en utilisant Unicode, version 6.0.0
- Unicode permet de coder plus d'un million de caractères
- Les nombres associés à chacun des caractères Unicode sont nommés « *code points* » ; ils vont de 0 à 10FFFF en hexadécimal

R. Grin

Introduction à Java

226

Les « plans » Unicode

- Les caractères de code 0 à FFFF sont appelés les caractères du BMP (*Basic Multilingual Plane*)
- Les caractères européens appartiennent tous à ce plan
- Les autres caractères (de code allant de 10000 à 10FFFF) sont des caractères dits supplémentaires
- Ce sont essentiellement des caractères asiatiques

R. Grin

Introduction à Java

227

Représentation des caractères en Java 5

- A partir de la version 5,
 - au bas niveau de l'API, un caractère est représenté par un `int` qui représente le *code point* du caractère Unicode
 - les types `String`, `StringBuffer`, `StringBuilder`, `char[]` utilisent le codage UTF-16 pour les interpréter comme contenant des `char`

R. Grin

Introduction à Java

228

Codage UTF-16

- Les caractères du BMP sont codés par une unité de codage (*code unit* en anglais) de 16 bits
- Les caractères supplémentaires sont codés à l'aide de 2 unités de codage, appelés *surrogates* (« de remplacement » en français)
- Une unité de codage est codé par 1 `char` (2 octets) en Java

R. Grin

Introduction à Java

229

Surrogates de UTF-16

- Une astuce de codage permet de savoir immédiatement si une unité de codage correspond à un caractère entier ou à un *surrogate* qui doit être complété par une autre unité de codage pour obtenir un caractère Unicode

R. Grin

Introduction à Java

230

Surrogates de UTF-16

- Le premier caractère *surrogate* a un code compris entre `\uD800` et `\uDBFF` (si Java rencontre un `char` avec cette valeur, il sait qu'il doit compléter par un autre `char`) ; on dit qu'il est dans l'intervalle « haut » des *surrogates*
- Le deuxième doit avoir un code dans l'intervalle `\uDC00` - `\uDFFF` (intervalle « bas » des *surrogates*)

R. Grin

Introduction à Java

231

`char` depuis la version 5

- Si on extrait un caractère d'une `String` on peut donc tomber sur un caractère « *surrogate* » et pas sur un caractère Unicode complet
- Si on veut écrire un code qui fonctionne dans tous les cas, **il faut éviter l'utilisation directe de données représentant un seul `char`** et lui préférer l'utilisation des séquences de caractères (`String`, `StringBuffer`, `StringBuilder`, `char[]`, `CharSequence`,...)

R. Grin

Introduction à Java

232

Exemple : compter les caractères

```
// Les 2 derniers char ne forment
// qu'un seul caractère Unicode
String s = "\u0041\u00DF\u6771\uD801\uDC00";
int nb = 0;
for (int i = 0; i < s.length(); i++) {
    char c = s.charAt(i);
    if (! Character.isHighSurrogate(c)) {
        nb++;
    }
}
System.out.println("Nbre de caract. " + nb);
```

R. Grin

Introduction à Java

233

Compter les caractères

- En fait `String` fournit une méthode pour compter les caractères :
`int codePointCount(int début, int fin)`
- Pour compter tous les caractères d'une `String` :
`s.codePointCount(0, s.length())`

R. Grin

Introduction à Java

234

Conversion entre `char` et `int`

- La classe `Character` qui enveloppe `char` a 2 méthodes `static` de conversion :
- `int toCodePoint(char high, char low)` convertit 2 `char` en `codePoint` sans validation ; si on veut valider il faut utiliser la méthode `boolean isSurrogatePair(char high, char low)`
- `char[] toChars(int codePoint)` convertit un `codePoint` en 1 ou 2 `char` (il existe une variante où `char[]` est passé en paramètre)

R. Grin

Introduction à Java

235

UTF-8

- C'est un codage des caractères Unicode sur 1, 2, 3 ou 4 octets
- Les caractères ASCII (étendus avec les caractères les plus utilisés des langues occidentales ; ISO-8859-1) sont codés sur 1 seul octet
- Ce codage peut donc être pratique dans les pays occidentaux comme la France

R. Grin

Introduction à Java

236

Autres codages

- Si Java code les caractères avec UTF-16 en interne (`char` et `String`), l'utilisateur peut souhaiter utiliser un autre codage
- Les codages disponibles dépendent de l'implémentation de Java (ils sont nombreux) ; voir méthode `Charset.availableCharsets()`
- Les codages UTF-8, UTF-16, US-ASCII et ISO-8859-1 sont requis sur toutes les implémentations

R. Grin

Introduction à Java

237

Codages standards

- La classe `StandardCharsets` (paquetage `java.nio.charset`) fournit des constantes pour les charsets qui doivent être fournis par toute implémentation du JDK
- Les constantes : `UTF_8`, `US_ASCII`, `ISO_8859_1`, `UTF_16` (ordre des octets identifié par un « BOM »), `UTF_16BE` (*Big-Endian*), `UTF_16LE` (*Little-Endian*)

R. Grin

Introduction à Java

238

Noms d'un codage

- Pas toujours évident de trouver le bon nom
- Chaque codage a un nom canonique (qui a pu changer suivant les versions) mais il peut avoir plusieurs alias ; le plus souvent un de ces noms peut désigner le codage dans le code Java
- Exemple : le nom canonique de UTF-8 est « UTF-8 » mais il existe aussi l'alias « UTF8 »

R. Grin

Introduction à Java

239

Charset

- La classe `java.nio.charset.Charset` représente un codage d'une chaîne de caractères en une suite d'octets
- La JVM a un charset par défaut qui dépend typiquement de la locale et du charset du système d'exploitation sous-jacent ; il est retourné par la méthode `static Charset defaultCharset()` de la classe `Charset`

R. Grin

Introduction à Java

240

Désigner un **Charset**

- Selon les méthodes du JDK, on peut désigner un charset en donnant simplement son nom canonique ou un alias (type **String**) ou en créant une instance de **Charset**
- On peut obtenir une instance de **Charset** par la méthode **static Charset.forName(String nomOuAlias)** ou (pour les charsets standards) par une constante de **StandardCharsets**
- Exemple : **StandardCharsets.UTF_8** ou **Charset.forName("UTF-8")**

R. Grin

Introduction à Java

241

Charset par défaut

- Sous Windows c'est le codage windows-1252 (CP1252, variante de ISO-8859-1) mais, si on lance l'exécution sous Eclipse, le codage par défaut dépend des propriétés du projet ou même du répertoire qui contient les fichiers et il peut être différent (UTF-8 par exemple)
- Il ne faut donc pas présumer de ce codage par défaut et il faut toujours donner explicitement le codage que l'on utilise (voir, en particulier, le cours sur les entrées-sorties)

R. Grin

Introduction à Java

242

Changer le codage par défaut

- La propriété système **file.encoding** fixe le codage par défaut utilisé par la JVM
- Il faut donner sa valeur au moment du lancement de la JVM (c'est trop tard quand l'application a déjà démarré)
- Par exemple

```
java -Dfile.encoding=utf-8 ...
```

met le codage par défaut à UTF-8

R. Grin

Introduction à Java

243

Conversion entre **String** et **byte[]**

- La conversion doit indiquer le codage à utiliser
- Exemple de conversion dans les 2 sens :

```
String s = "abc";
byte [] ba1 = s.getBytes("8859_1");
String t = new String(ba2, "Cp1252");
```
- Exemple d'utilisation : vérifier qu'une chaîne de caractères conservée dans une base de données ne dépasse pas la taille maximum en octet

R. Grin

Introduction à Java

244

Conversion entre **String** et **char[]**

- **String** vers **char[]** : **toCharArray()**
- **char[]** vers **String** : **new String(char[])**
Attention, comme pour tous les tableaux, **toString()** renvoie l'adresse du tableau en mémoire et pas la traduction de **char[]** vers **String**
- Exemples :

```
String s = "abc";
char[] ca = s.toCharArray();
String s = new String(ca);
```

R. Grin

Introduction à Java

245