



Plan de cours

- Chapitre 1 : Présentation du langage JAVA
- Chapitre 2 : Classes JAVA
- Chapitre 3 : Héritage
- Chapitre 4 : Exceptions
- Chapitre 5 : Interfaces graphiques (Swing)
- Chapitre 6 : JDBC



Point important:

- Prenez des notes, je dis des choses intéressantes et utiles parfois...

3



Plan de cours

- Chapitre 1 : Présentation du langage JAVA
- Chapitre 2 : Classes JAVA
- Chapitre 3 : Héritage
- Chapitre 4 : Exceptions
- Chapitre 5 : Interfaces graphiques (Swing)
- Chapitre 6 : JDBC

4



Chapitre 1 : Présentation du langage JAVA

Le langage Java est un langage de **programmation orienté objet** créé par **James Gosling** et **Patrick Naughton**, employés de **Sun Microsystems**, avec le soutien de **Bill Joy** (cofondateur de **Sun Microsystems** en 1982), présenté officiellement le 23 mai 1995 au SunWorld.

en 2009



5



Chapitre 1 : Présentation du langage JAVA

Quelques chiffres et faits à propos de Java en 2011 :

- ❑ 97% des machines d'entreprises ont une JVM installée
- ❑ Java est téléchargé plus d'un milliards de fois chaque année
- ❑ Il y a plus de 9 millions de développeurs Java dans le monde
- ❑ Java est un des langages les plus utilisé dans le monde
- ❑ Tous les lecteurs de Blue-Ray utilisent Java
- ❑ Plus de 3 milliards d'appareils mobiles peuvent mettre en oeuvre Java
- ❑ Plus de 1,4 milliards de cartes à puce utilisant Java sont produites chaque année

6

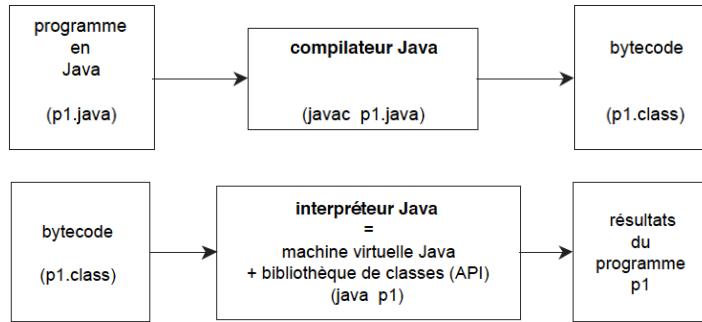


Chapitre 1 : Présentation du langage JAVA

Java possède un certain nombre de caractéristiques qui ont largement contribué à son énorme succès :

Java est interprété

Le fichier source est compilé en pseudo code ou bytecode puis exécuté par un interpréteur Java : la Java Virtual Machine (JVM). Ce concept est à la base du slogan de Sun pour Java : WORA (Write Once, Run Anywhere : écrire une fois, exécuter partout). En effet, le bytecode, s'il ne contient pas de code spécifique à une plate-forme particulière peut être exécuté et obtenir quasiment les mêmes résultats sur toutes les machines disposant d'une JVM.



7



Chapitre 1 : Présentation du langage JAVA

Java possède un certain nombre de caractéristiques qui ont largement contribué à son énorme succès :

Java est portable

il n'y a pas de compilation spécifique pour chaque plate forme. Le code reste indépendant de la machine sur laquelle il s'exécute. Il est possible d'exécuter des programmes Java sur tous les environnements qui possèdent une Java Virtual Machine. Cette indépendance est assurée au niveau du code source grâce à Unicode et au niveau du bytecode.

Java est orienté objet

comme la plupart des langages récents, Java est orienté objet. Chaque fichier source contient la définition d'une ou plusieurs classes qui sont utilisées les unes avec les autres pour former une application. Java n'est pas complètement objet car il définit des types primitifs (entier, caractère, flottant, booléen,...).

8



Chapitre 1 : Présentation du langage JAVA

Java possède un certain nombre de caractéristiques qui ont largement contribué à son énorme succès :

Java est simple

le choix de ses auteurs a été d'abandonner des éléments mal compris ou mal exploités des autres langages tels que la notion de pointeurs (pour éviter les incidents en manipulant directement la mémoire), l'héritage multiple et la surcharge des opérateurs, ...

Java est fortement typé

toutes les variables sont typées et il n'existe pas de conversion automatique qui risquerait une perte de données. Si une telle conversion doit être réalisée, le développeur doit obligatoirement utiliser un cast ou une méthode statique fournie en standard pour la réaliser.

9



Chapitre 1 : Présentation du langage JAVA

Java possède un certain nombre de caractéristiques qui ont largement contribué à son énorme succès :

Java assure la gestion de la mémoire

l'allocation de la mémoire pour un objet est automatique à sa création et Java récupère automatiquement la mémoire inutilisée grâce au garbage collector qui restitue les zones de mémoire laissées libres suite à la destruction des objets.

10



Chapitre 1 : Présentation du langage JAVA

Java possède un certain nombre de caractéristiques qui ont largement contribué à son énorme succès :

Java est sûr

la sécurité fait partie intégrante du système d'exécution et du compilateur. Un programme Java planté ne menace pas le système d'exploitation. Il ne peut pas y avoir d'accès direct à la mémoire. L'accès au disque dur est réglementé dans une applet.

Les applets fonctionnant sur le Web sont soumises aux restrictions suivantes dans la version 1.0 de Java :

- aucun programme ne peut ouvrir, lire, écrire ou effacer un fichier sur le système de l'utilisateur
- aucun programme ne peut lancer un autre programme sur le système de l'utilisateur
- toute fenêtre créée par le programme est clairement identifiée comme étant une fenêtre Java, ce qui interdit par exemple la création d'une fausse fenêtre demandant un mot de passe
- les programmes ne peuvent pas se connecter à d'autres sites Web que celui dont ils proviennent.

11



Chapitre 1 : Présentation du langage JAVA

Java possède un certain nombre de caractéristiques qui ont largement contribué à son énorme succès :

Java est économique

le pseudo code a une taille relativement petite car les bibliothèques de classes requises ne sont liées qu'à l'exécution.

Java est multitâche

il permet l'utilisation de threads qui sont des unités d'exécutions isolées. La JVM, elle-même, utilise plusieurs threads.

12



Chapitre 1 : Présentation du langage JAVA

Il existe 2 types de programmes avec la version standard de Java : les applets et les applications.

- Une application autonome (stand alone program) est une application qui s'exécute sous le contrôle direct du système d'exploitation.
- Une applet est une application qui est chargée par un navigateur et qui est exécutée sous le contrôle d'un plug in de ce dernier.

Les principales différences entre une applet et une application sont :

- les applets n'ont pas de méthode main() : la méthode main() est appelée par la machine virtuelle pour exécuter une application.
- les applets ne peuvent pas être testées avec l'interpréteur. Elles doivent être testées avec l'applet viewer ou doivent être intégrées à une page HTML, elle même visualisée avec un navigateur disposant d'un plug in Java,

13



Chapitre 1 : Présentation du langage JAVA

Un bref historique de Java

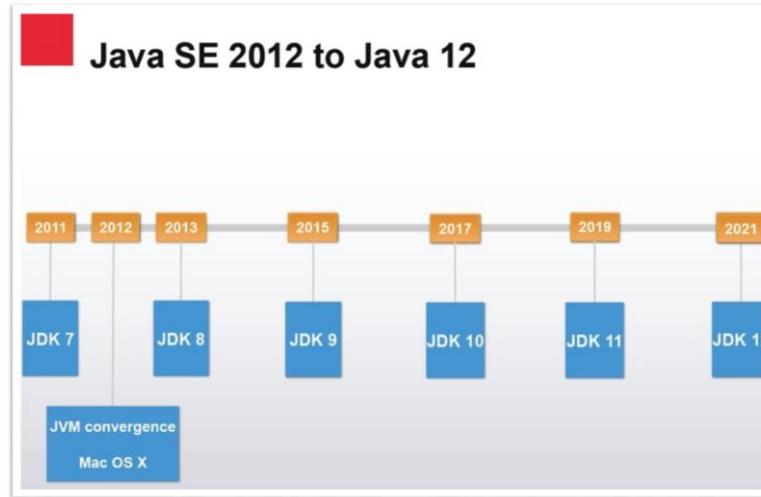
Année	Événements		
1995	mai : premier lancement commercial du JDK 1.0	2011	juillet : Java SE 7 octobre : JavaFX 2.0
1996	janvier : JDK 1.0.1	2012	août : JavaFX 2.2
1996	septembre : lancement du JDC	2013	juin : Java EE 7
1997	février : JDK 1.1	2014	mars : Java SE 8, JavaFX 8
1998	décembre : lancement de J2SE 1.2 et du JCP		
1999	décembre : lancement J2EE		
2000	mai : J2SE 1.3		
2002	février : J2SE 1.4		
2004	septembre : J2SE 5.0		
2006	mai : Java EE 5 décembre : Java SE 6.0		
2008	décembre : Java FX 1.0		
2009	février : JavaFX 1.1 juin : JavaFX 1.2 décembre : Java EE 6		
2010	janvier : rachat de Sun par Oracle avril : JavaFX 1.3		

14



Chapitre 1 : Présentation du langage JAVA

Un bref historique de Java



15



Chapitre 1 : Présentation du langage JAVA

Un bref historique de Java

Quelle est la différence entre les plates-formes JRE et SE ?

	JRE (Java Runtime Environment)	Java SE (plate-forme Java, Standard Edition)
À qui est-elle destinée ?	Aux utilisateurs exécutant des applets et des applications créés à l'aide de la technologie Java.	Aux développeurs de logiciels créant des applets et des applications à l'aide de la technologie Java.
De quoi s'agit-il ?	C'est un environnement nécessaire à l'exécution d'applets et d'applications créés à l'aide du langage de programmation Java.	C'est un kit de développement logiciel utilisé pour créer des applets et des applications à l'aide du langage de programmation Java.
Comment l'obtenir ?	La plate-forme Java 2 est distribuée gratuitement et disponible sur : java.com	La plate-forme Java 2 est distribuée gratuitement et disponible sur : oracle.com/javase

16



Chapitre 1 : Présentation du langage JAVA

Un bref historique de Java

Quelle est la différence entre JRE et JDK ?

JRE (Java Runtime environment)	JDK (Java Development Kit)
Il s'agit d'une implémentation de la machine virtuelle Java* qui exécute des programmes Java.	Il s'agit d'un bundle de logiciels que vous pouvez utiliser pour développer des applications Java.
Java Runtime Environment est un plug-in requis pour l'exécution de programmes Java.	Le kit de développement Java (JDK) est requis pour le développement d'applications Java.
Le JRE est moins volumineux que le JDK et requiert donc moins d'espace disque.	Le JDK requiert davantage d'espace disque car il contient le JRE ainsi que divers outils de développement.
Le JRE peut être téléchargé/pris en charge gratuitement sur java.com	Le JDK peut être téléchargé/pris en charge gratuitement sur oracle.com/technetwork/java/javase/downloads/
Il contient le JVM, les bibliothèques de base et d'autres composants supplémentaires pour l'exécution d'applications et d'applets écrits dans Java.	Il contient le JRE, un jeu de classes API, un compilateur Java, Web Start et d'autres fichiers requis pour l'écriture d'applets et d'applications Java.

17



Chapitre 1 : Présentation du langage JAVA

Un bref historique de Java

Depuis sa version 1.2, Java a été renommé **Java 2**. Les numéros de version 1.2 et 2 désignent donc la même version. Le JDK a été renommé **J2SDK (Java 2 Software Development Kit)** mais la dénomination JDK reste encore largement utilisée, à tel point que la dénomination JDK est reprise dans la version 5.0. Le **JRE** a été renommé **J2RE (Java 2 Runtime Environment)**.

Trois plate-formes d'exécution (ou éditions) Java sont définies pour des cibles distinctes selon les besoins des applications à développer :

- **Java Standard Edition (J2SE / Java SE)** : environnement d'exécution et ensemble complet d'API pour des applications de type desktop. Cette plate-forme sert de base en tout ou partie aux autres plate-formes
- **Java Enterprise Edition (J2EE / Java EE)** : environnement d'exécution reposant intégralement sur Java SE pour le développement d'applications d'entreprises
- **Java Micro Edition (J2ME / Java ME)** : environnement d'exécution et API pour le développement d'applications sur appareils mobiles et embarqués dont les capacités ne permettent pas la mise en oeuvre de Java SE

18



Chapitre 1 : Présentation du langage JAVA

Un bref historique de Java

Avec différentes éditions, les types d'applications qui peuvent être développées en Java sont nombreux et variés :

- Applications desktop
- Applications web : servlets/JSP, portlets, applets
- Applications pour appareil mobile (MIDP) : midlets
- Applications pour appareil embarqué (CDC) : Xlets
- Applications pour carte à puce (Javacard) : applets Javacard
- Applications temps réel

19



Chapitre 1 : Présentation du langage JAVA

LA SYNTAXE DE JAVA

Java est un langage objet qui s'appuie sur la syntaxe du langage C et du C++. Java est un langage objet : on doit utiliser les classes et les objets ; C++ est un langage orienté objet, sorte de langage hybride permettant à la fois la programmation classique sans objets et la programmation avec objets.

La figure ci-dessous présente un programme réalisant la somme des nb éléments d'un tableau tabEnt d'entiers.

```
// SomTab.java somme des nb éléments d'un tableau d'entiers
class SomTab {
    public static void main (String[] args) {
        int[] tabEnt = {1, 2, 3}; // tableau de int nommé tabEnt
        //int tabEnt[] = {1, 2, 3}; identique à l'instruction précédente
        int somme = 0;
        //for (int i=0; i<3; i++) somme += tabEnt[i];
        for (int i=0; i<tabEnt.length; i++) somme += tabEnt[i];
        System.out.println ("Somme : " + somme);
    }
} // class
```

Le résultat de l'exécution est le suivant :

Somme : 6

20

**LA SYNTAXE DE JAVA**

- Java est sensible à la casse.
- Les blocs de code sont encadrés par des accolades. Chaque instruction se termine par un caractère ';' (point virgule).
- Une instruction peut tenir sur plusieurs lignes :

Les commentaires**Il existe trois types de commentaire en Java :**

Type de commentaires	Exemple
commentaire abrégé	// commentaire sur une seule ligne int N=1; // déclaration du compteur
commentaire multiligne	/* commentaires ligne 1 commentaires ligne 2 */
commentaire de documentation automatique	/** * commentaire de la méthode * @param val la valeur à traiter * @since 1.0 * @return la valeur de retour * @deprecated Utiliser la nouvelle méthode XXX */

21

**LA SYNTAXE DE JAVA****La déclaration de variables**

Une variable possède un nom, un type et une valeur.

La déclaration d'une variable permet de réserver la mémoire pour en stocker la valeur.

Le type d'une variable peut être :

- soit un type élémentaire dit aussi type primitif déclaré sous la forme type_elémentaire variable;
- soit une classe déclarée sous la forme classe variable ;

Exemple :

```
long nombre;
int compteur;
String chaine;
```

Rappel : les noms de variables en Java peuvent commencer par une lettre, par le caractère de soulignement ou par le signe dollar. Le reste du nom peut comporter des lettres ou des nombres mais jamais d'espaces.

22

**LA SYNTAXE DE JAVA****Les types élémentaires**

Les types élémentaires ont une taille identique quelque soit la plate-forme d'exécution : c'est un des éléments qui permet à Java d'être indépendant de la plate-forme sur laquelle le code s'exécute

byte	un entier signé sur 8 bits	de -128 à +127
short	un entier signé sur 16 bits	de -32 768 à +32 767
int	un entier signé sur 32 bits	de -2 147 483 648 à 2 147 483 647
long	un entier signé sur 64 bits	de l'ordre de (±) 9 milliards de milliards
float	un réel sur 32 bits	de l'ordre de 1.4E-45 à 3.4E38
double	un réel sur 64 bits	de l'ordre de 4.9E-324 à 1.79E308
boolean	true ou false	
char	un caractère Unicode sur 16 bits	entier positif entre 0 et 65 535

Remarque : Les types élémentaires commencent tous par une minuscule.

23

**LA SYNTAXE DE JAVA****Exemples de déclaration de variables avec ou sans valeur initiale**

```
// TypesPrimitifs.java les types primitifs Java
class TypesPrimitifs {
    public static void main (String[] args) {
        // entiers
        byte b; // b : variable locale non initialisée
        short s;
        int i;
        long l;
        byte b1 = 50; // b1 : variable locale initialisée à 50
        short s1 = 5000;
        int i1 = 50000;
        int i2 = Integer.MIN_VALUE; // le plus petit int
        int i3 = Integer.MAX_VALUE; // le plus grand int
        int i4 = 0x1F; // constante entière en hexadécimal
        int i5 = 012; // constante entière en octal
        long l1 = 10000000;
```

24

**LA SYNTAXE DE JAVA**

Exemples de déclaration de variables avec ou sans valeur initiale

```
// réels
float f;
float f1 = 2.5f; // ajout de f pour type float
float f2 = 2.5e3f; // 2.5 x 10 à la puissance 3
float f3 = Float.MIN_VALUE; // le plus petit float
double d;
double d1 = 2.5; // pourrait s'écrire 2.5d (default : double)
double d2 = 2.5e3; // ou 2.5e3d

// caractères
char c;
char c1 = 'é';
char c2 = '\n'; // passage à la ligne suivante
char c3 = '\u00e9'; // caractère Unicode du é
char c4 = '\351'; // caractère é en octal : e916 = 3518

// booléens
boolean bo;
boolean bo1 = true;
boolean bo2 = false;
```

25

**LA SYNTAXE DE JAVA**

Exemples de déclaration de variables avec ou sans valeur initiale

Java vérifie qu'une variable locale (à la fonction main() dans cet exemple) est initialisée avant son utilisation. Si la variable n'est pas initialisée, le compilateur signale un message d'erreur comme sur les exemples ci-dessous.

```
// Les instructions suivantes donnent un message d'erreur :
// variable locale utilisée avant initialisation
//System.out.println ("b : " + b);
//System.out.println ("s : " + s);
//System.out.println ("i : " + i);
//System.out.println ("l : " + l);
//System.out.println ("c : " + c);
//System.out.println ("bo : " + bo);
```

26

**LA SYNTAXE DE JAVA****Exemples de déclaration de variables avec ou sans valeur initiale**

L'instruction suivante écrit en début de ligne (indiquée par \n), pour chaque variable, son nom suivi de sa valeur convertie en caractères. L'opérateur + est ici un opérateur de concaténation de chaînes de caractères.

```
// Écriture d'entiers : byte, short, int, long
System.out.println (
    "\nb1 : " + b1 +
    "\ns1 : " + s1 +
    "\ni1 : " + i1 +
    "\ni2 : " + i2 +
    "\ni3 : " + i3 +
    "\ni4 : " + i4 +
    "\ni5 : " + i5 +
    "\ni11 : " + i11
);
```

27

**LA SYNTAXE DE JAVA****Exemple de résultats d'exécution de la classe TypesPrimitifs :**

```
b1 : 50
s1 : 5 000
i1 : 50 000
i2 : -2 147 483 648
i3 : 2 147 483 647
i4 : 31
i5 : 10
i11 : 10 000 000
f1 : 2.5
f2 : 2500.0
f3 : 1.4E-45
d1 : 2.5
d2 : 2500.0
c1 : é
c3 : é
c4 : é
bo1 : true
bo2 : false
```

le plus petit int
le plus grand int
1F en hexa
12 en octal

le plus petit float

caractère Unicode 00e9 en hexa
351 en octal

Remarque : pour chaque type, des constantes indiquant les maximums et les minimums sont définies : Byte.MIN_VALUE, Byte.MAX_VALUE, Short.MIN_VALUE, etc.

28

**LA SYNTAXE DE JAVA****Exercice 1.1 – Programme d'écriture des valeurs limites des types primitifs**

Écrire le programme complet donnant les limites minimums et maximums de chacun des types entiers et réels.

```
// LimitPrimitif.java  les limites des types primitifs
class LimitPrimitif {
    public static void main (String[] args) {
        System.out.println (
            "\nByte.MIN_VALUE      : " + Byte.MIN_VALUE +
            "\nByte.MAX_VALUE      : " + Byte.MAX_VALUE +
            "\nShort.MIN_VALUE     : " + Short.MIN_VALUE +
            "\nShort.MAX_VALUE     : " + Short.MAX_VALUE +
            "\nInteger.MIN_VALUE   : " + Integer.MIN_VALUE +
            "\nInteger.MAX_VALUE   : " + Integer.MAX_VALUE +
            "\nLong.MIN_VALUE       : " + Long.MIN_VALUE +
            "\nLong.MAX_VALUE       : " + Long.MAX_VALUE +
            "\nFloat.MIN_VALUE      : " + Float.MIN_VALUE +
            "\nFloat.MAX_VALUE      : " + Float.MAX_VALUE +
            "\nDouble.MIN_VALUE     : " + Double.MIN_VALUE +
            "\nDouble.MAX_VALUE     : " + Double.MAX_VALUE
        );
    } // main
} // class LimitPrimitif
```

Byte.MIN_VALUE	:	-128
Byte.MAX_VALUE	:	127
Short.MIN_VALUE	:	-32 768
Short.MAX_VALUE	:	32 767
Integer.MIN_VALUE	:	-2147483648
Integer.MAX_VALUE	:	2 147 483 647
Long.MIN_VALUE	:	-9223372036854775808
Long.MAX_VALUE	:	9223372036854775807
Float.MIN_VALUE	:	1.4E-45
Float.MAX_VALUE	:	3.4028235E38
Double.MIN_VALUE	:	4.9E-324
Double.MAX_VALUE	:	1.7976931348623157E308

**LA SYNTAXE DE JAVA****Les constantes symboliques**

Les constantes symboliques sont déclarées comme des variables précédées de final . Leur contenu ne peut pas être modifié par la suite. L'identificateur de la constante est souvent écrit en majuscules. Exemples de déclaration de constantes symboliques :

```
final int GAUCHE = 1;
final int HAUT   = 2;
final int DROITE = 3;
final int BAS    = 4;

final double PI  = 3.1415926535;
```



Chapitre 1 : Présentation du langage JAVA

LA SYNTAXE DE JAVA

Les opérateurs arithmétiques, relationnels, logiques

Les opérateurs arithmétiques réalisent des opérations sur des variables entières ou réelles. Les opérations possibles sont listées ci-après.

+	addition	$n = a + b;$
-	soustraction	$n = a - b;$
*	multiplication	$n = a * b;$
/	division	$n = a / b;$
+=	addition	$n += b;$ à n, ajouter b
-=	soustraction	$n -= b;$ à n, soustraire b
*=	multiplication	$n *= b;$ multiplier n par b
/=	division	$n /= b;$ diviser n par b

Le modulo fournit le reste de la division entière de deux nombres :

%	modulo	$n = a \% b;$ reste de la division de a par b
%=	modulo	$n \%= b;$ reste de la division de n par b

Les opérateurs d'incrémentation

++	incrémente la variable	$j++;$ ou $++j;$
--	décrémente la variable	$j--;$ ou $--j;$

31



Chapitre 1 : Présentation du langage JAVA

LA SYNTAXE DE JAVA

Les opérateurs (relationnels) de comparaison

>	supérieur	$a > 0$	(a est-il supérieur à 0 ?)
<	inférieur		
<=	inférieur ou égal		
>=	supérieur ou égal	$n >= 1$	(n supérieur ou égal à 1 ?)
==	égal		
!=	non égal (différent)	$a != 0$	(a différent de 0 ?)

Les opérateurs logiques

&&	et logique	$(c >= 0) \&\& (c <= 15)$
 	ou logique	$(n < 0 n >= nbCouleurs)$
!	négation	$(!trouve)$

32

LA SYNTAXE DE JAVALe transtypage (cast)

Le transtypage est nécessaire quand il risque d'y avoir perte d'information, comme lors de l'affectation d'un entier long (64 bits) à un entier int (32 bits), ou d'un réel double vers un réel float. On force alors le type, indiquant ainsi au compilateur qu'on est conscient du problème de risque de perte d'information.

Byte → short → int → long → float → double

```
int i3 = 10;
long l4 = i3 * (long) 2.5f;      // résultat : 20
long l5 = (long) (i3 * 2.5f);   // résultat : 25
```

Remarque : dans les expressions arithmétiques, les entiers de type byte ou short sont considérés comme des int (convertis d'office en int). Si b1 est un byte, b1 + 1 est considéré être un int. L'affectation b1 = b1 + 1 est illégale sans cast (de type int -> byte)

33

LA SYNTAXE DE JAVALes tableaux à une dimension : Les tableaux d'éléments de type primitif

Plusieurs éléments de même type peuvent être regroupés (sous un même nom) en tableau. On peut accéder à chaque élément à l'aide d'un d'indice. Le premier élément porte l'indice 0. Les tableaux sont alloués dynamiquement en Java. Ils sont initialisés par défaut à 0 pour les nombres et à faux pour les tableaux de booléens.

Exemples de déclarations : tableau de types primitifs (type int) avec allocation et initialisation du tableau :

```
int tabEnt[] = {1, 2, 3};           // (1) identique au langage C
int[] tabEnt = {1, 2, 3};          // (2) indique mieux un tableau d'int
```

(1) et (2) sont identiques et corrects en Java. La forme (2) indique mieux un tableau d'entiers nommé tabEnt, et initialisé avec les valeurs 1, 2 et 3.



tabEnt est une référence non initialisée vers un tableau d'entiers.

34

**LA SYNTAXE DE JAVA**

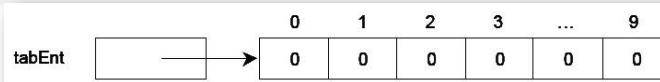
Les tableaux à une dimension : Les tableaux d'éléments de type primitif

Exemples de déclarations avec allocation dynamique du tableau :

```
int[] tabEnt; // référence (non initialisée) vers un tableau de int
```

déclare et définit la variable tabEnt comme une référence sur un tableau d'entiers, mais ne réserve pas de place mémoire pour ce tableau. L'allocation doit se faire avec **new** en indiquant le nombre d'éléments. Ci-après, on réserve 10 éléments de type int, numérotés de 0 à 9.

```
tabEnt = new int [10]; // allocation dynamique de 10 int pour tabEnt
```



35

**LA SYNTAXE DE JAVA**

Les tableaux à une dimension : Les tableaux d'éléments de type primitif

Ou encore en une seule instruction :

```
int[] tabEnt = new int [10]; // tableau d'int de 10 éléments
```

tabEnt.length fournit la longueur déclarée du tableau tabEnt (soit 10 sur l'exemple).

36



Chapitre 1 : Présentation du langage JAVA

LA SYNTAXE DE JAVA

Les tableaux à une dimension : Les tableaux d'objets

On déclare de la même façon un tableau d'objets. tabBalle est un tableau d'éléments de type Balle pouvant référencer six objets de type Balle numérotés de 0 à 5.

```
Balle[] tabBalle = new Balle [6];
```

Les chaînes de caractères étant gérées en Java sous forme de classes, on peut de même déclarer des tableaux d'objets de type String (chaînes de caractères).

```
String[] prenoms = new String [2];      // prenoms : tableau de 2 String
prenoms[0] = "Michel";                 // prenoms[0] référence la chaîne "Michel"
prenoms[1] = "Josette";
```

37

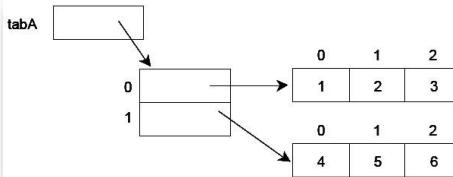


Chapitre 1 : Présentation du langage JAVA

LA SYNTAXE DE JAVA

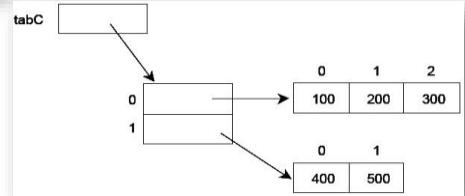
Les tableaux à une dimension : Les tableaux à plusieurs dimensions

On peut généraliser la notion de tableau à une dimension pour créer des tableaux multidimensionnels. Un tableau à deux dimensions est un tableau de tableaux.



Un tableau à deux dimensions de deux lignes et d'un nombre variable de colonnes.

Un tableau d'entiers à deux dimensions de deux lignes et trois colonnes.



38

**LA SYNTAXE DE JAVA****Les instructions de contrôle (alternatives, boucles)**

Les instructions de contrôle du déroulement lors de l'exécution des instructions ont la même syntaxe que pour le langage C. Leur syntaxe est présentée succinctement ci-dessous. L'espace n'étant pas significatif en Java, les instructions peuvent être présentées (indentées) de différentes façons.

Exemple :

```
int lg;                                // numMois de type int
if (numMois == 1) {                      // si numMois vaut 1 alors
    lg = 31;
    System.out.println ("janvier");
}
if (numMois == 1) lg = 31;                // une seule instruction
```

alternative double :

```
if (...) {
    ...
} else {
    ...
}
```

39

**LA SYNTAXE DE JAVA****Les instructions de contrôle (alternatives, boucles)**

choix multiple :

```
switch (exp) {      // exp : expression de type byte, short, int ou char
    case ...:
        ...
        break;
    case ...:
        ...
        break;
    default :           // le cas default est facultatif
        ...
        // break;          // optionnel si c'est le dernier cas
} // switch
```

40

**LA SYNTAXE DE JAVA****Les instructions de contrôle (alternatives, boucles)**

Exemple 1 avec une expression entière :

```
// SwitchJava1.java          test du switch java avec un entier
class SwitchJava1 {
    public static void main (String[] args) {
        int lgMois;
        int numMois = 3;      // changer la valeur pour le test du switch
        //int numMois = 13;      // numéro de mois 13 incorrect
        switch (numMois) {      // numMois est de type int
            case 1 : case 3 : case 5 : case 7 :
            case 8 : case 10 : case 12 :
                lgMois = 31;
                break;
            case 4 : case 6 : case 9 : case 11 :
                lgMois = 30;
                break;
            case 2 :
                lgMois = 28;
                break;
            default :
                lgMois = 0;
                System.out.println ("Numéro de mois " + numMois + " incorrect");
                // break;
        } // switch
        System.out.println ("lgMois : " + lgMois);
    } // main
} // class SwitchJava1
```

41

**LA SYNTAXE DE JAVA****Les instructions de contrôle (alternatives, boucles)****Les boucles for, while et do ... while****boucle for :**

```
for (exp1; exp2; exp3) {
    ...
}
```

- exp1 : évaluée avant de commencer la boucle.
- exp2 : condition évaluée en début d'itération ; la boucle continue si exp2 est vraie.
- exp3 : évaluée en fin de chaque itération.

```
for (int i=0; i < tab.length; i++) somme += tab[i];
```

ou encore surtout s'il y a plusieurs instructions dans la boucle :

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

42

**LA SYNTAXE DE JAVA**

Les instructions de contrôle (alternatives, boucles)

boucle while :

```
// tant que la condition est vraie, on exécute la boucle
while (<condition >) {
    ...
}
```

Exemple :

```
while (!fini) {           // fini est un booléen (tant que non fini faire)
    ...
}                         // à un moment, le corps de la boucle met fini à vrai
```

boucle do ... while :

```
// exécuter la boucle tant que la condition de fin de boucle est vraie
do {
    ...
} while (<condition >);      // condition de poursuite de la boucle
```

43

**Plan de cours**

- Chapitre 1 : Présentation du langage JAVA
- Chapitre 2 : Classes JAVA
- Chapitre 3 : Héritage
- Chapitre 4 : Exceptions
- Chapitre 5 : Interfaces graphiques (Swing)

44



Chapitre 2 : Classes JAVA

La notion de classe

Une classe est une extension de la notion de module. Les données et les fonctions traitant les données sont réunies ensemble dans une classe qui constitue un nouveau type.

Les attributs (les données, les champs)	Pile
	<ul style="list-style-type: none"> - int sommet - int[] element
Les méthodes (les fonctions)	<ul style="list-style-type: none"> + Pile (int max) + boolean pileVide () + void empiler (int v) + int depiler () + void viderPile () + void listerPile () - void erreur (String mes)

La classe Pile (-indique un attribut ou une méthode privé).

Exemple :

```
Pile p1 = new Pile (5); // p1 et p2 sont des instances de la classe Pile
Pile p2 = new Pile (100); // p1 et p2 sont des objets de type Pile
```

45



Chapitre 2 : Classes JAVA

La notion de classe

Pile p1	Pile p2
<ul style="list-style-type: none"> - int sommet - int[] element <ul style="list-style-type: none"> + Pile (int max) + boolean pileVide () + void empiler (int v) + int depiler () + void viderPile () + void listerPile () - void erreur (String mes) 	<ul style="list-style-type: none"> - int sommet - int[] element <ul style="list-style-type: none"> + Pile (int max) + boolean pileVide () + void empiler (int v) + int depiler () + void viderPile () + void listerPile () - void erreur (String mes)

Deux objets (deux instances) de la classe Pile.

- Chaque objet a ses propres attributs (en grisé) qui sont privés (encapsulés dans l'objet).
- Les méthodes par contre ne sont pas dupliquées

46

**La notion de classe**

```
// Pile.java gestion d'une pile d'entiers
public class Pile {
    // sommet et element sont des attributs privés
    private int sommet;          // repère le dernier occupé (le sommet)
    private int[] element;       // tableau d'entiers alloué dynamiquement
    // erreur est une méthode privée utilisable seulement
    // dans la classe Pile
    private void erreur (String mes) {
        System.out.println ("***erreur : " + mes);
    }
    public Pile (int max) {      // voir 2.2.1 Le constructeur d'un objet
        sommet = -1;
        element = new int [max];           // allocation de max entiers
    }
}
```

Le codage en Java de la classe Pile est indiqué ci-après. Le mot clé "private" indique que les attributs sommet et element, et la méthode erreur() sont privés.

47

**La notion de classe et Objet****Qu'est ce qu'un objet ?**

- Toute entité identifiable, concrète ou abstraite, peut être considérée comme un objet
- Un objet réagit à certains messages qu'on lui envoie de l'extérieur
 - la façon dont il réagit détermine le comportement de l'objet
- Il ne réagit pas toujours de la même façon à un même événement
 - sa réaction dépend de l'état dans lequel il se trouve

48

La notion de classe et Objet

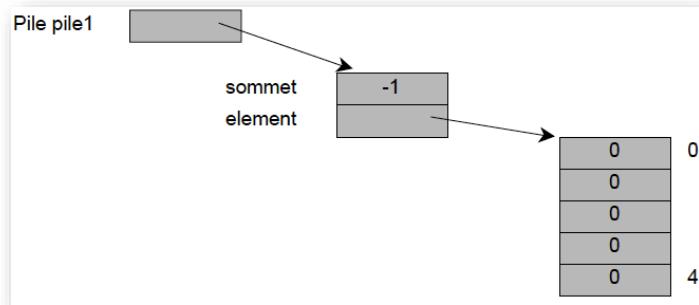
- Un objet a :
 - une adresse en mémoire (identifie l'objet)
 - un comportement (ou interface)
 - un état interne
- Le comportement est donné par des fonctions ou procédures, appelées méthodes
- L'état interne est donné par des valeurs de variables d'instances

49

La notion de classe et Objet

Un objet est une instance, un exemplaire construit dynamiquement sur le modèle que décrit la classe.

```
Pile pile1 = new Pile (5) ;
```



50

**La notion de constructeur d'un objet**

Constructeur : méthode particulière, portant le même nom que la classe et automatiquement invoquée juste après la création d'une instance.

- Un constructeur peut avoir un nombre quelconque de paramètres.
- Un constructeur sans paramètre est conseillé sur toute classe.
- Un paramètre peut avoir le même nom qu'une variable d'instance.

L'exécution d'un constructeur débute par une initialisation éventuelle et automatique des attributs de l'objet si des valeurs ont été données à la déclaration. Elle se poursuit avec l'exécution des instructions du constructeur

51

**La notion de constructeur d'un objet**Exemple :

```
public class Point {
    double x = 0.0;
    double y;

    public Point() {
        x = 1.0;
        y = 2.0;
    }

    public Point(double i, double j){
        x = i;
        y = j;
    }
}
```

```
class Rectangle{
    Point origin = new Point(0,0);
    Point corner;

    Rectangle(Point p1, p2) {
        origin = p1;
        corner = p2;
    }
}
```

```
new Rectangle();
new Rectangle(new Point(), new Point(3,4));
```

52

**La notion de constructeur d'un objet****Quelques règles concernant les constructeurs**

- 1) un constructeur ne fournit aucune valeur. Dans son en-tête, aucun type ne doit figurer devant son nom. Même la présence (logique) de void est une erreur :

```
class Truc
{
    ....
    public void Truc () // erreur de compilation : void interdit ici
    {
        ....
    }
}
```

- 2) Une classe peut ne disposer d'aucun constructeur. On peut alors instancier des objets comme s'il existait un constructeur par défaut sans arguments.

- 3) Un constructeur ne peut pas être appelé directement depuis une autre méthode. :

```
Point a = new Point (3, 5) ;
.....
a.Point(8, 3) ; // interdit
```

53

**La notion de constructeur d'un objet****Quelques règles concernant les constructeurs**

- 4) Un constructeur peut appeler un autre constructeur de la même classe. Cette possibilité utilise la sur-définition des méthodes et nécessite l'utilisation du mot clé super

- 5) Un constructeur peut être déclaré privé (private). Dans ce cas, il ne pourra plus être appelé de l'extérieur, c'est-à-dire qu'il ne pourra pas être utilisé pour instancier des objets :

```
class A
{ private A() { ..... } // constructeur privé sans arguments
    .....
}
.....
A a() ; // erreur : le constructeur correspondant A() est privé
```

54



Chapitre 2 : Classes JAVA

La notion de destruction d'un objet

Ramasse-miette

Java dispose d'un ramasse-miette (garbage collector). L'espace mémoire occupé par tout objet qui n'est plus référencé est automatiquement récupéré au bout d'un certain temps.

La méthode finalize

Il est possible, pour toute classe, d'écrire une méthode finalize, automatiquement invoquée lorsqu'un objet va être récupéré par le GC (Équivalent des destructeurs C++).

```
protected void finalize() throws Throwable {
    ...
    super.finalize();
}
```

55



Chapitre 2 : Classes JAVA

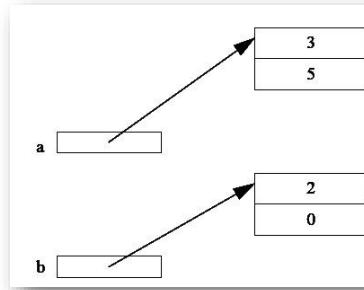
Affection et comparaison d'objets

Exemple 1

Supposons que nous disposions d'une classe Point possédant un constructeur à deux arguments entiers et considérons ces instructions :

```
Point a, b ;
.....
a = new Point (3, 5) ;
b = new Point (2, 0) ;
```

Après leur exécution, on aboutit à cette situation :

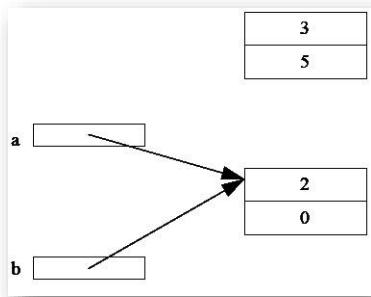


56

**Affectation et comparaison d'objets**

Exécutons maintenant l'affectation : a = b

?



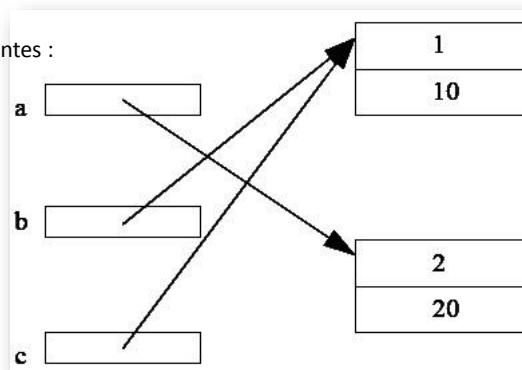
Dorénavant, a et b désignent le même objet, et non pas deux objets de même valeur

57

**Affectation et comparaison d'objets****Exemple 2**

Considérons les instructions suivantes :

```
Point a, b, c ;
.....
a = new Point (1, 10) ;
b = new Point (2, 20) ;
c = a ;
a = b ;
b = c ;
```



Notez bien qu'il n'existe ici que deux objets de type Point et trois variables de type Point (trois références, dont deux de même valeur).

58

**Initialisation de référence et référence nulle**

En Java, il n'est pas possible de définir une variable locale d'un type primitif sans l'initialiser.
La règle se généralise aux variables locales de type classe .

Considérez cet exemple utilisant une classe Point disposant d'une méthode affiche :

```
public static void main (String args[])
{ Point p ;           // p est locale à main
  p.affiche() ;      // erreur de compilation : p n'a pas encore reçu de valeur
  ....
}
```

59

**La notion de clone**

L'affectation de variables de type objet se limite à la recopie de références. Elle ne provoque pas la recopie de la valeur des objets.

Si on le souhaite, on peut bien entendu effectuer explicitement la recopie de tous les champs d'un objet dans un autre objet de même type.

```
class Point
{ public Point(int abs, int ord) { x = abs ; y = ord ; }
  public Point copie ()          // renvoie une référence à un Point
  { Point p = new Point(x, y) ;
    p.x = x ; p.y = y ;
    return p ;
  }
  private int x, y ;
}
.....
Point a = new Point(1, 2) ;
Point b = a.copie() ;      // b est une copie conforme de a
```

Cette démarche est utilisable tant que la classe concernée ne comporte pas de champs de type classe.

60



Chapitre 2 : Classes JAVA

La notion de clone

la copie profonde d'un objet : on se contente de recopier la valeur de tous ses champs, y compris ceux de type classe,

la copie superficielle d'un objet : comme précédemment, on recopie la valeur des champs d'un type primitif mais pour les champs de type classe, on crée une nouvelle référence à un autre objet du même type de même valeur.

61



Chapitre 2 : Classes JAVA

Comparaison d'objet

Les opérateurs == et != s'appliquent théoriquement à des objets. Mais comme ils portent sur les références elles-mêmes, leur intérêt est très limité.

Ainsi, avec :

Point a, b ;

L'expression a == b est vraie uniquement si a et b font référence à un seul et même objet, et non pas seulement si les valeurs des champs de a et b sont les mêmes.

62

**Règles d'écriture des méthodes****Méthodes fonction**

Une méthode peut ne fournir aucun résultat. Le mot clé **void** figure alors dans son en-tête à la place du type de la valeur de retour.

Mais une méthode peut aussi fournir un résultat. Nous parlerons alors de méthode fonction. Voici par exemple une méthode `distance` qu'on pourrait ajouter à une classe `Point` pour obtenir la distance d'un point à l'origine :

```
public class Point
{
    ....
    double distance ()
    { double d ;
        d = Math.sqrt (x*x* + y*y) ;
        return d ;
    }
    private int x, y ;
    ....
}
```

```
int getX { return x ; }
int getY { return y ; }
```

63

**Règles d'écriture des méthodes****Méthodes fonction**

La valeur fournie par une méthode fonction peut apparaître dans une expression, comme dans ces exemples utilisant les méthodes `distance`, `getX` et `getY` précédentes :

```
Point a = new Point(...) ; double u, r ;
..... u = 2. * a.distance() ; r = Math.sqrt(a.getX() * a.getX() +
a.getY() * a.getY() ) ;
```

64

**Règles d'écriture des méthodes****Arguments muets ou effectifs**

Comme dans tous les langages, les arguments figurant dans l'en-tête de la définition d'une méthode

Il est possible de déclarer un argument muet avec l'attribut final . Dans ce cas, le compilateur s'assure que sa valeur n'est pas modifiée par la méthode :

Les arguments fournis lors de l'appel de la méthode portent quant à eux le nom d'arguments effectifs (ou encore paramètres effectifs).

```
void f (final int n, double x)
{ ..... n = 12 ; // erreur de compilation x = 2.5 ; // OK .....
}
```

65

**Règles d'écriture des méthodes****Conversion des arguments effectifs**

Jusqu'ici, nous avions appelé nos différentes méthodes en utilisant des arguments effectifs d'un type identique à celui de l'argument muet correspondant.

Il faut simplement que la conversion dans le type attendu soit une conversion implicite légale

66

**Règles d'écriture des méthodes****Conversion des arguments effectifs**

Voici quelques exemples usuels :

```
class Point
{ ..... void deplace (int dx, int dy) { ..... } .....
}
.....
Point p = new Point(...);
int n1, n2 ; byte b ; long q ;
.....
p.deplace (n1, n2);                                // OK : appel normal
p.deplace (b+3, n1);                               // OK : b+3 est déjà de type int
p.deplace (b, n1);                                 // OK : b de type byte sera converti en int
p.deplace (n1, q);                                 // erreur : q de type long ne peut être converti en int
p.deplace (n1, (int)q);                            // OK
```

67

**Règles d'écriture des méthodes****Conversion des arguments effectifs**

Voici quelques autres exemples plus insidieux :

```
class Point
{ ..... void deplace (byte dx, byte dy) { ..... } .....
}
.....Point p = new Point(...); byte b1, b2 ;
```

```
p.deplace (b1, b2);                                // OK : appel normal
p.deplace (b1+1,b2);                             // erreur : b1+1 de type int ne peut être converti en byte p.
deplace (b1++, b2);                            // OK : b1++ est de type byte
```

68



Chapitre 2 : Classes JAVA

Méthodes & Attribut statiques :

Un attribut statique est un attribut qui est commun à tous les objets que vous pourrez créer. On peut par exemple citer un compteur du nombres d'instances de classe que vous aurez lancées.

```
public class Test{
    public static int nombre;
    public static final int nb = 5;
    public Test(){
        nombre++;
        System.out.println("Nombre d'instances créées : " + nombre);
    }
}
```

System.out.println("Nombre d'instances créées : " + Test.nb);

Ce code affichera 5.

69



Chapitre 2 : Classes JAVA

Méthodes & Attribut statiques :

Supposons que vous voulez écrire une méthode int abs(int x) dans une classe Math qui calcule la valeur absolue d'un nombre x. Le comportement de cette méthode ne dépend pas de la valeur des variables d'instance de la class Math.

Pourquoi devrait-on créer un objet de la classe Math pour utiliser cette méthode ? Ce serait perdre de la place sur la mémoire.

La classe Math existe en Java, ainsi que la méthode abs(). Et effectivement il n'est pas possible d'instancier cette classe.

```
Math objetMath = new Math ();
```

Cette instruction renvoie l'erreur suivante à la compilation :

```
% javac TestMath
TestMath . java :3: Math () has private
access in java . lang . Math
...
1 error
```

70

**Méthodes & Attribut statiques :**

- En général, une classe possédant des méthodes statiques n'est pas conçue pour être instanciée.
- Mais cela ne signifie pas qu'une classe possédant une méthode statique ne doit jamais être instanciée : si une méthode de la classe n'est pas statique, alors l'instanciation doit être possible.

71

**Méthodes & Attribut statiques :**

Une méthode statique est une méthode qui peut être appelée même sans avoir instancié la classe. Une méthode statique ne peut accéder qu'à des attributs et méthodes statiques.

Exemple d'application avec une méthode statique :

```
public class Test{
    public int test;
    public static String chaine = "bonjour";
    public Test(){
        MaMethodeStatique();
    }
    public static void MaMethodeStatique(){
        int nombre = 10;
        System.out.println("Appel de la méthode statique : " + nombre + chaine);
    }
}
```

Vous pouvez sans avoir instancié la classe accéder à la méthode statique en tapant ceci :
Test.MaMethodeStatique();

n'importe où dans votre code.

72

**Méthodes & Attribut statiques :**

Bien que Java soit un langage objet, il existe des cas où une instance de classe est inutile. Le mot clé **static** permet alors à une méthode de s'exécuter sans avoir à instancier la classe qui la contient. L'appel à une méthode statique se fait alors en utilisant le nom de la classe, plutôt que le nom de l'objet.

```
public class MaClassMath {
    ...
    public static int min( int a, int b) {
        // retourne la plus petite valeur
    }
}

public TestMaClassMath {
    public static void main( String []args){
        int x = MaClassMath.min(21,4);
    }
}
```

73

**La surcharge de méthodes :**

La surcharge survient lorsque l'on a deux méthodes du même nom mais qui ne prennent pas les mêmes paramètres. :

Exemple de surcharge de méthode

```
1- public class Test{
2-     public Test(){
3-         MaMethode();
4-         MaMethode(50);
5-     }
6-     public void MaMethode(int variable){
7-         System.out.println("Le nombre que vous avez passé en paramètre vaut : " + variable);
8-     }
9-     public void MaMethode(){
10-         System.out.println("Vous avez appelé la méthode sans paramètre ");
11-     }
12 }
```

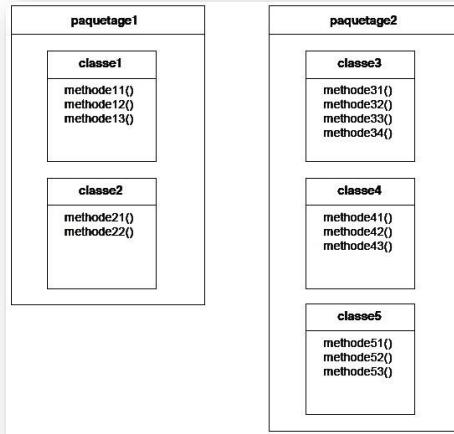
74



Chapitre 2 : Classes JAVA

Les packages en Java

Java permet de regrouper les classes en ensembles appelés packages afin de faciliter la modularité.



75



Chapitre 2 : Classes JAVA

Les packages en Java

Parmi les paquetages de Java vous avez :

<code>java.applet</code>	Classes de base pour les applets
<code>java.awt</code>	Classes d'interface graphique AWT
<code>java.io</code>	Classes d'entrées/sorties (flux, fichiers)
<code>java.lang</code>	Classes de support du langage
<code>java.math</code>	Classes permettant la gestion de grands nombres.
<code>java.net</code>	Classes de support réseau (URL, sockets)
<code>java.rmi</code>	Classes pour les méthodes invoquées à partir de machines virtuelles non locales.
<code>java.security</code>	Classes et interfaces pour la gestion de la sécurité.
<code>java.sql</code>	Classes pour l'utilisation de JDBC.
<code>java.text</code>	Classes pour la manipulation de textes, de dates et de nombres dans plusieurs langages.
<code>java.util</code>	Classes d'utilitaires (vecteurs, hashtable)
<code>javax.swing</code>	Classes d'interface graphique

76

**Les packages en Java**

Nom : chaque paquetage porte un nom.

Ce nom est soit un simple identificateur ou une suite d'identificateurs séparés par des points.
L'instruction permettant de nommer un paquetage doit figurer au début du fichier source (.java) comme suit:

```
----- Fichier Exemple.java -----
package nomtest; // 1ere instruction

class MTest {}

class Uneautre {}

public class Exemple {// blabla suite du code ...}

----- Fin du Fichier -----
```

Dans cette opération nous venons d'assigner toutes les classes du fichier **Exemple.java** au paquetage **nomtest**.

77

**Les packages en Java**

Si vous avez besoin d'utiliser une classe se trouve dans le paquetage **nomtest**

import nomtest.;*

Par cette instruction vous avez demandé d'importer toutes les classes se trouvant dans le paquetage nomtest

Remarques :

- 1) Le paquetage java.lang est importé automatiquement par le compilateur.
- 2) *import nomtest.*;*
 - Cette instruction ne va pas importer de manière récursive les classes se trouvant dans nomtest et dans ses sous paquetages.
 - Elle va uniquement se contenter de faire un balayage d'un seul niveau.
 - Elle va importer donc que les classes du paquetage nomtest.

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

78

**Les packages en Java**

Si deux paquetages contiennent le même nom de classe il y a problème!

Par exemple

- la classe List des paquetages java.awt et java.util ou bien la classe Date des paquetages java.util et java.sql etc.
- Pour corriger ce problème vous devez impérativement spécifier le nom exact du paquetage à importer (ou à utiliser).
- java.awt.List (ou) java.util.List ; java.util.Date (ou) java.sql.Date
 - *import java.awt.*; // ici on importe toutes les classes dans awt*
 - *import java.util.*; // ici aussi*
 - *import java.util.List; // ici on précise la classe List à utiliser*

79

**Classe String**

En Java, les chaînes de caractères sont gérées à l'aide de la classe String fournie en standard dans le paquetage ***java.lang***. Cette gestion diffère du langage C où les chaînes de caractères sont mémorisées comme une suite de caractères terminée par 0.

- Le principe d'encapsulation et l'accès direct aux données d'un objet de la classe String n'est pas possible et les structures de données utilisées sont inconnues.
- La classe String met à la disposition de l'utilisateur un large éventail de méthodes gérant les chaînes de caractères.

Remarque :

- *la chaîne de caractères d'un objet de type String ne peut pas être modifiée .*
- *En cas de modification, les méthodes fournissent un nouvel objet à partir de la chaîne de l'objet courant et répondant aux caractéristiques de la méthode. Une même chaîne de caractères (String) peut être référencée plusieurs fois puisqu'on est sûr qu'elle ne peut pas être modifiée.*
- *La classe StringBuffer par contre permet la création et la modification d'une chaîne de caractères.*

80

**Classe String**

Les premières méthodes sont des constructeurs d'un objet de type String.

- **String ()** : crée une chaîne vide
- **String (String)** : crée une chaîne à partir d'une autre chaîne
- **String (StringBuffer)** : crée une chaîne à partir d'une autre chaîne de type String-Buffer

Remarque :

String s = "Monsieur" ; est accepté par le compilateur et est équivalent à

String s = new String ("Monsieur") ;

- **char charAt (int n)** : fournit le n ième caractère de la chaîne de l'objet courant.
- **int compareTo (String s)** : compare la chaîne de l'objet et la chaîne s ; fournit 0 si =, <0 si inférieur, >0 sinon.
- **String concat (String s)** : concatène la chaîne de l'objet et la chaîne s (crée un nouvel objet).
- **boolean equals (Object s)** : compare la chaîne de l'objet et la chaîne s.
- **boolean equalsIgnoreCase (String s)** : compare la chaîne de l'objet et la chaîne s (sans tenir compte de la casse).

81

**Classe String**

- **int length ()** : longueur de la chaîne.
- **String toLowerCase ()** : fournit une nouvelle chaîne (nouvel objet) convertie en minuscules.
- **String toUpperCase ()** : fournit une nouvelle chaîne (nouvel objet) convertie en majuscules.
- **char charAt (int n)** : fournit le n ième caractère de la chaîne de l'objet courant.
- **int compareTo (String s)** : compare la chaîne de l'objet et la chaîne s ; fournit 0 si =, <0 si inférieur, >0 sinon.
- **String concat (String s)** : concatène la chaîne de l'objet et la chaîne s (crée un nouvel objet).
- **boolean equals (Object s)** : compare la chaîne de l'objet et la chaîne s.
- **boolean equalsIgnoreCase (String s)** : compare la chaîne de l'objet et la chaîne s (sans tenir compte de la casse).
- **int length ()** : longueur de la chaîne.
- **String toLowerCase ()** : fournit une nouvelle chaîne (nouvel objet) convertie en minuscules.
- **String toUpperCase ()** : fournit une nouvelle chaîne (nouvel objet) convertie en majuscules.

82

**Classe String**

- **int indexOf (char c)** : indice du caractère c dans la chaîne ; -1 s'il n'existe pas.
- **int indexOf (char, int)** : indice d'un caractère de la chaîne en partant d'un indice donné.
- **int indexOf (String s)** : indice de la sous-chaîne s.
- **int indexOf (String, int)** : indice de la sous-chaîne en partant d'un indice.
- **int lastIndexOf (int c)** : indice du dernier caractère c.
- **int lastIndexOf (char c, int)** : indice du dernier caractère en partant de la fin et d'un indice.
- **int lastIndexOf (String)** : indice de la dernière sous-chaîne.
- **int lastIndexOf (String, int)** : indice de la dernière sous-chaîne en partant d'un indice.
- **String replace (char c1, char c2)** : crée une nouvelle chaîne en remplaçant le caractère c1 par c2.
- **boolean startsWith (String s)** : teste si la chaîne de l'objet commence par s.
- **boolean startsWith (String s, int n)** : teste si la chaîne de l'objet commence par s en partant de l'indice n..

83

**Classe String**

- **boolean endsWith (String s)** : teste si la chaîne de l'objet se termine par s.
- **String substring (int d)** : fournit la sous-chaîne de l'objet commençant à l'indice d.
- **String substring (int d, int f)** : fournit la sous-chaîne de l'objet entre les indices d (inclus) et f (exclu).
- **String trim ()** : enlève les espaces en début et fin de chaîne.
- L'opérateur + est un opérateur de concaténation de chaînes de caractères souvent utilisé pour la présentation des résultats à écrire.
- Il peut aussi s'utiliser pour former une chaîne de caractères.
- Les objets de type String ne sont pas modifiables ; il faut donc créer un nouvel objet de type String si on concatène deux objets de type String.
- Si les deux opérandes de l'opérateur + sont de type String, le résultat est de type String.
- Si un des deux opérandes est de type String et l'autre de type primitif (entier, réel, caractère, booléen), le type primitif est converti en un String.
- Si un des deux opérandes est de type String et l'autre est un objet, l'objet est remplacé par une chaîne de caractères fournie par la méthode `toString()` de l'objet (ou celle de la classe Object, ancêtre de toutes les classes)

84

**Classe String**

```
class TestString1 {
    public static void main (String[] args) {
        String ch1 = "Louis"; int i1 = 13;
        String ch2 = ch1 + i1; System.out.println ("ch2 : " + ch2);
        // .....
        String ch3 = i1 + ch1; System.out.println ("ch3 : " + ch3);
        // .....
        String ch4 = ch1 + i1 + 2; System.out.println ("ch4 : " + ch4);
        // .....
        String ch5 = ch1 + (i1 + 2); System.out.println ("ch5 : " + ch5);
        // .....
        //String ch6 = i1 + 2;
        // .....
        String ch7 = "" + i1 + 2;
        // .....
        System.out.println ("ch7 : " + ch7); String ch8 = i1 + 2 + "";
        // .....
        System.out.println ("ch8 : " + ch8); String ch9 = Integer.toString (i1 + 2);
        // .....
    }
}
```

85

**Classe String**

```
class TestString1 {
    public static void main (String[] args) {
        String ch1 = "Louis"; int i1 = 13;
        String ch2 = ch1 + i1; System.out.println ("ch2 : " + ch2);
        // Louis13
        String ch3 = i1 + ch1; System.out.println ("ch3 : " + ch3);
        // 13Louis
        String ch4 = ch1 + i1 + 2; System.out.println ("ch4 : " + ch4);
        // Louis132
        String ch5 = ch1 + (i1 + 2); System.out.println ("ch5 : " + ch5);
        // Louis15
        //String ch6 = i1 + 2;
        // impossible de convertir int en String
        String ch7 = "" + i1 + 2;
        // 132
        System.out.println ("ch7 : " + ch7); String ch8 = i1 + 2 + "";
        // 15
        System.out.println ("ch8 : " + ch8); String ch9 = Integer.toString (i1 + 2);
        // 15 } }
```

86

**Classe String****Résultats d'exécution**

```
ch2 : Louis13
ch3 : 13Louis
ch4 : Louis132
ch5 : Louis15
ch7 : 132
ch8 : 15
ch9 : 15
```

87

**Classe String****Exemple de la méthode compareTo**

```
public class CompareToExample {
    public static void main(String args[]){
        String s1="hello";
        String s2="hello";
        String s3="hemlo";
        String s4="flag";
        System.out.println(s1.compareTo(s2)); //0
        System.out.println(s1.compareTo(s3)); //1
        System.out.println(s1.compareTo(s4)); //2
    }
}
```

0
-1
2

88

**Classe String****Exemple de la méthode charAt**

```

public class Exercise1 {
    public static void main(String[] args)
    {
        String str = "Java Exercises!";
        System.out.println("Original String = " + str);
        // Get the character at positions 0 and 10.
        int index1 = str.charAt(0);
        int index2 = str.charAt(10);

        // Print out the results.
        System.out.println("The character at position 0 is " +
                           (char)index1);
        System.out.println("The character at position 10 is " +
                           (char)index2);
    }
}

```

89

**Classe String****Exemple de la méthode trim**

```

1 | public class StringTrimExample{
2 | public static void main(String args[]){
3 |     String s1=" hello ";
4 |     System.out.println(s1+"how are you");           // without trim()
5 |     System.out.println(s1.trim()+"how are you"); // with trim()
6 | }

```

```

hello how are you
hellohow are you

```

90

**Classe String****Exemple de la méthode toString**

```

3 class Student{
4     int rollno;
5     String name;
6     String city;
7
8@ Student(int rollno, String name, String city){
9     this.rollno=rollno;
0     this.name=name;
1     this.city=city;
2 }
3
4@ public static void main(String args[]){
5     Student s1=new Student(101,"Raj","lucknow");
6     Student s2=new Student(102,"Vijay","ghaziabad");
7
8     System.out.println(s1);//compiler writes here s1.toString()
9     System.out.println(s2);//compiler writes here s2.toString()
0 }
1 }
```

Student@70dea4e
Student@5c647e05

La classe *Object*, dont dérive toute classe, dispose d'une méthode *toString* qui, par défaut, affiche le nom de la classe et l'adresse de l'objet concerné.

91

**Classe String****Exemple de la méthode toString**

```

class Student{
    int rollno;
    String name;
    String city;

    Student(int rollno, String name, String city){
        this.rollno=rollno;
        this.name=name;
        this.city=city;
    }

    public String toString(){//overriding the toString() method
        return rollno+" "+name+" "+city;
    }

    public static void main(String args[]){
        Student s1=new Student(101,"Raj","lucknow");
        Student s2=new Student(102,"Vijay","ghaziabad");

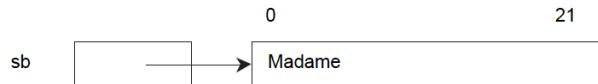
        System.out.println(s1);//compiler writes here s1.toString()
        System.out.println(s2);//compiler writes here s2.toString()
    }
}
```

101 Raj lucknow
102 Vijay ghaziabad

92

**Classe StringBuffer**

Un objet de type **StringBuffer** représente une chaîne de caractères modifiable. On peut changer un caractère, ajouter des caractères au tampon interne de l'objet défini avec une capacité qui peut être changée.



Un objet sb de la classe **StringBuffer**.

Il contient six caractères et a une capacité de 22 caractères, numérotés de 0 à 21.

- **StringBuffer ()** : construit un tampon ayant 0 caractère mais d'une capacité de 16 caractères.
- **StringBuffer (int n)** : construit un tampon de capacité n caractères.
- **StringBuffer (String)** : construit un objet de type StringBuffer à partir d'un objet de type String.

93

**Classe StringBuffer**

Les méthodes de type **StringBuffer append ()** concatènent au tampon de l'objet courant la représentation sous forme de caractères de leur argument (boolean, int, etc.).

- **StringBuffer append (boolean);**
- **StringBuffer append (char);**
- **StringBuffer append (char[]);**
- **StringBuffer append (char[], int, int);**
- **StringBuffer append (double);**
- **StringBuffer append (float);**
- **StringBuffer append (int);**
- **StringBuffer append (long);**
- **StringBuffer append (Object) :** ajoute **toString()** de l'objet.
- **StringBuffer append (String);**
- **StringBuffer insert (int, boolean) ;**
- **StringBuffer insert (int, char) ;**
- **StringBuffer insert (int, char[]) ;**
- **StringBuffer insert (int, double) ;**
- **StringBuffer insert (int, float) ;**
- **StringBuffer insert (int, int) ;**

94



Chapitre 2 : Classes JAVA

Exercice

Il s'agit d'écrire un programme qui, étant donnée une chaîne de caractères (une instance de la classe String).

Indiquer s'il s'agit ou non d'un palindrome ??

95



Chapitre 2 : Classes JAVA

Correction

```

1. import java.util.*;
2.
3. class Palindrome
4. {
5.     public static void main(String args[])
6.     {
7.         String original, reverse = ""; // Objects of String class
8.         Scanner in = new Scanner(System.in);
9.
10.        System.out.println("Enter a string to check if it is a palindrome");
11.        original = in.nextLine();
12.
13.        int length = original.length();
14.
15.        for (int i = length - 1; i >= 0; i--)
16.            reverse = reverse + original.charAt(i);
17.
18.        if (original.equals(reverse))
19.            System.out.println("The string is a palindrome.");
20.        else
21.            System.out.println("The string isn't a palindrome.");
22.
23.    }
24. }
```

96



Chapitre 2 : Classes JAVA

Exercice

Écrire un programme Java qui permet de supprimer les caractères doublons d'une chaîne donnée.

The given string is : w3resource
 After removing duplicates characters the new string is : w3resouc

97



Chapitre 2 : Classes JAVA

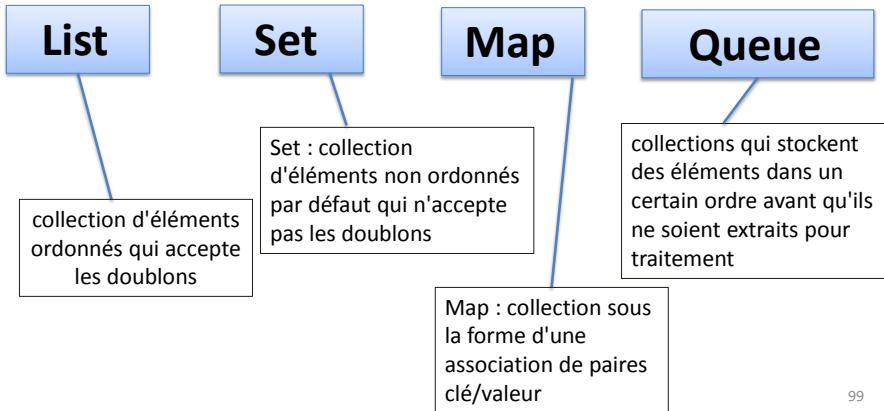
Exercice

```
import java.util.*;
public class Main {
    public static void main(String[] args) {
        String str1 = "w3resource";
        System.out.println("The given string is: " + str1);
        System.out.println("After removing duplicates characters the new string is: " + removeDuplicateChars(str1));
    }
    private static String removeDuplicateChars(String sourceStr) {
        char[] arr1 = sourceStr.toCharArray();
        String targetStr = "";
        for (char value: arr1) {
            if (targetStr.indexOf(value) == -1) {
                targetStr += value;
            }
        }
        return targetStr;
    }
}
```

98



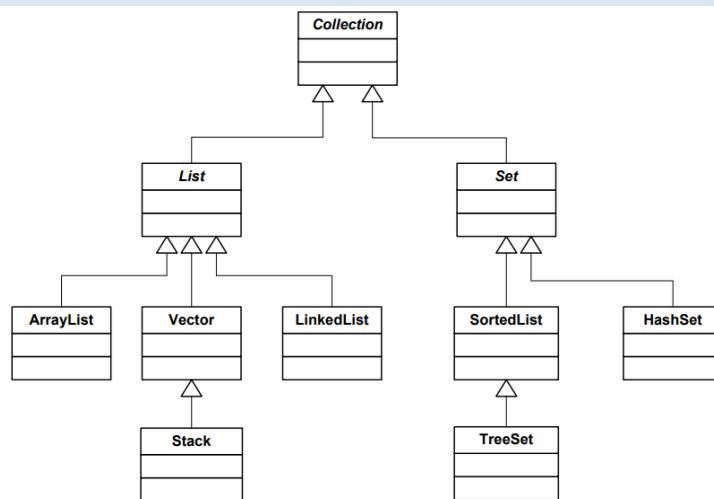
Types Abstraits de Données Ou Collections java



99



Types Abstraits de Données Ou Collections java



100

**Classe Vector**

Lorsque l'on crée un tableau, il faut spécifier sa taille qui est fixe. Java fournit un objet très utilisé: le vecteur (classe **java.util.Vector**). L'utilisation d'un vecteur plutôt qu'un tableau est souvent avantageux lorsque la taille finale du tableau n'est pas connue à l'avance. Un vecteur est un tableau dynamique. La première cellule d'un vecteur est à l'index zéro.

Principales méthodes de la classe Vector**constructeurs :**

- **Vector()** crée un vecteur vide
- **Vector(int nombre)** crée un vecteur vide de capacité précisée.
- **elementCount** nombre d'éléments du vecteur

méthodes :

- **isEmpty()** retourne true si le vecteur est vide
- **size()** retourne le nombre d'éléments du vecteur
- **addElement(Objet)** ajoute un élément à la fin du vecteur
- **insertElementAt(Objet, int position)** ajoute un élément à la position spécifiée
- **contains(Objet)** retourne true s'il l'Objet se trouve dans le vecteur : la comparaison se fait par la méthode equals
- **ElementAt(int position)** renvoie un objet qu'il faut caster.

```
Integer var=(Integer)MonVecteur.elementAt(0);
```

101

**Classe Vector : Exemple**

```
import java.util.Vector;

class Main {
    public static void main(String[] args) {
        Vector<String> mammals= new Vector<>();

        // Using the add() method
        mammals.add("Dog");
        mammals.add("Horse");

        // Using index number
        mammals.add(2, "Cat");
        System.out.println("Vector: " + mammals);

        // Using addAll()
        Vector<String> animals = new Vector<>();
        animals.add("Crocodile");

        animals.addAll(mammals);
        System.out.println("New Vector: " + animals);
    }
}
```

102

Classe Vector : Exemple

```

import java.util.Iterator;
import java.util.Vector;

class Main {
    public static void main(String[] args) {
        Vector<String> animals= new Vector<>();
        animals.add("Dog");
        animals.add("Horse");
        animals.add("Cat");

        // Using get()
        String element = animals.get(2);
        System.out.println("Element at index 2: " + element);

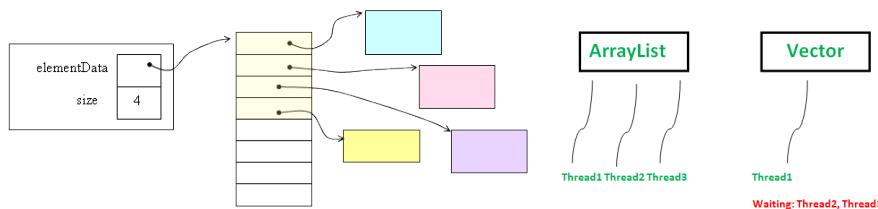
        // Using iterator()
        Iterator<String> iterate = animals.iterator();
        System.out.print("Vector: ");
        while(iterate.hasNext()) {
            System.out.print(iterate.next());
            System.out.print(", ");
        }
    }
}

```

103

Classe ArrayList

La classe **ArrayList** implémente un tableau d'objets qui peut grandir ou rétrécir à la demande, ce qui débarrasse le programmeur de la gestion de la taille du tableau. Comme pour un tableau on peut accéder à un élément du ArrayList, par un indice.



```
1 | ArrayList< String> names = new ArrayList< String>();
```

104



Chapitre 2 : Classes JAVA

Classe ArrayList : constructeur

```
ArrayList< String> noms = new ArrayList< String>(10);
```

Le tableau suivant résume quelques méthodes utiles ArrayList.

Méthode	Description
<code>public void add(Object)</code>	Ajouter un élément à une ArrayList; la version par défaut ajoute un élément au prochain emplacement disponible; une version surchargée vous permet de spécifier une position à laquelle nous voulons ajouter l'élément
<code>public void add(int, Object)</code>	
<code>public void remove(int)</code>	Supprimer un élément d'une ArrayList à un emplacement spécifié
<code>public void set(int, Object)</code>	Modifier un élément à un emplacement spécifié dans une ArrayList
<code>Object get(int)</code>	Récupérer un élément d'un emplacement spécifié dans une ArrayList
<code>public int size()</code>	Renvoyer la taille actuelle de ArrayList

105



Chapitre 2 : Classes JAVA

Classe ArrayList

```
import java.util.ArrayList;

class Main {
    public static void main(String[] args) {
        ArrayList<String> animals= new ArrayList<>();
        // Add elements in the array list
        animals.add("Dog");
        animals.add("Cat");
        animals.add("Horse");
        System.out.println("ArrayList: " + animals);

        // Change the element of the array list
        animals.set(2, "Zebra");
        System.out.println("Modified ArrayList: " + animals);
    }
}
```

```
ArrayList: [Dog, Cat, Horse]
Modified ArrayList: [Dog, Cat, Zebra]
```

106



Chapitre 2 : Classes JAVA

Classe ArrayList

```
import java.util.ArrayList;

class Main {
    public static void main(String[] args) {
        ArrayList<String> animals = new ArrayList<>();

        // Add elements in the array list
        animals.add("Dog");
        animals.add("Cat");
        animals.add("Horse");
        System.out.println("Initial ArrayList: " + animals);

        // Remove element from index 2
        String str = animals.remove(2);
        System.out.println("Final ArrayList: " + animals);
        System.out.println("Removed Element: " + str);
    }
}
```

```
Initial ArrayList: [Dog, Cat, Horse]
Final ArrayList: [Dog, Cat]
Removed Element: Horse
```

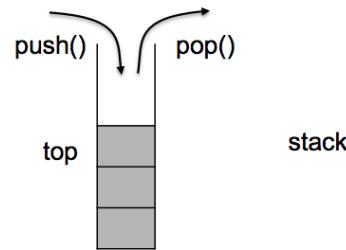
107



Chapitre 2 : Classes JAVA

Classe Stack

Une pile peut aussi contenir des objets de différents types.



La classe stack hérite directement de la classe Vector et ajoute entre autres les méthodes

- **push (objet)** : ajoute un objet en tête;
- **pop ()** : retire le premier objet;
- **peek ()** : renvoie une référence au premier objet sans le retirer;
- **empty ()** : teste si la pile est vide.
- **int search(Object o)** Retourne l'index de l'objet depuis le sommet de la pile (1 = sommet de la pile, -1 = non trouvé)

108



Chapitre 2 : Classes JAVA

Classe Stack

```
import java.util.Stack;

class Main {
    public static void main(String[] args) {
        Stack<String> animals= new Stack<>();

        // Add elements to Stack
        animals.push("Dog");
        animals.push("Horse");
        animals.push("Cat");
        System.out.println("Initial Stack: " + animals);

        // Remove element stacks
        String element = animals.pop();
        System.out.println("Removed Element: " + element);
    }
}
```

Initial Stack: [Dog, Horse, Cat]
Removed Element: Cat

109



Chapitre 2 : Classes JAVA

Classe Stack

```
import java.util.Stack;

class Main {
    public static void main(String[] args) {
        Stack<String> animals= new Stack<>();

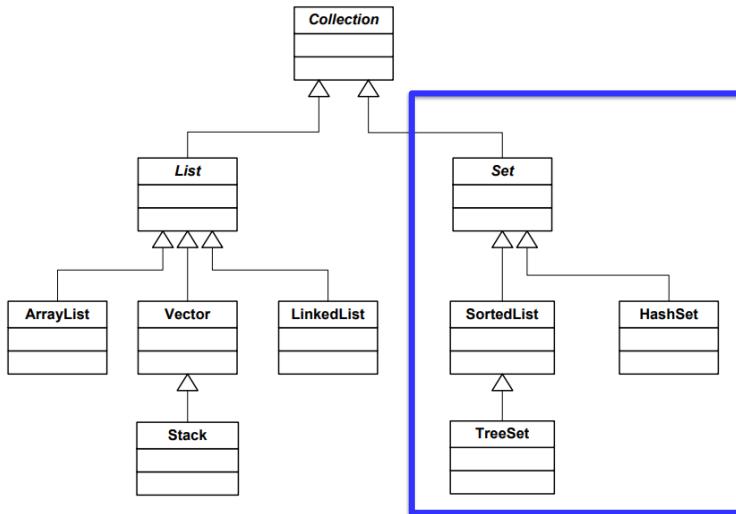
        // Add elements to Stack
        animals.push("Dog");
        animals.push("Horse");
        animals.push("Cat");
        System.out.println("Stack: " + animals);

        // Access element from the top
        String element = animals.peek();
        System.out.println("Element at top: " + element);
    }
}
```

Output

Stack: [Dog, Horse, Cat]
Element at top: Cat

110

Set

111

Set

L'interface Set définit les fonctionnalités d'une collection qui ne peut pas contenir de doublons dans ses éléments.

La valeur renournée par **hashCode()** est recherchée dans la collection :

- si aucun objet de la collection n'a la même valeur de hachage alors l'objet n'est pas encore dans la collection.
- si un ou plusieurs objets de la collection ont la même valeur de hachage alors la méthode **equals()** est invoquée,

112

**Set****Les différentes implémentations*****HashSet******TreeSet******LinkedHashSet******SortedSet***

113

**Set****L'interface définit plusieurs méthodes :**

boolean add() : Ajouter l'élément fourni en paramètre à la collection si celle-ci ne le contient pas déjà et renvoyer un booléen qui précise si la collection a été modifiée (l'implémentation de cette opération est optionnelle)

void clear() : Retirer tous les éléments de la collection (l'implémentation de cette opération est optionnelle)

boolean contains(Object o) : Renvoyer un booléen qui précise si la collection contient l'élément fourni en paramètre

boolean equals(Object o) : Comparer l'égalité de la collection avec l'objet fourni en paramètre. L'égalité est vérifiée si l'objet est de type Set, que les deux collections ont le même nombre d'éléments et que chaque élément d'une collection est contenu dans l'autre

int hashCode() Retourner la valeur de hachage de la collection

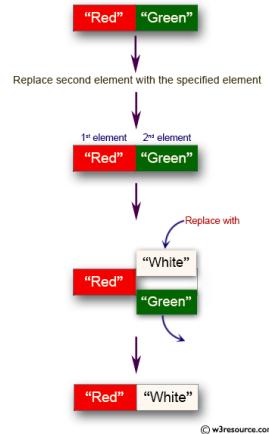
boolean isEmpty() : Renvoyer un booléen qui précise si la collection est vide

boolean remove(Object o) : Retirer l'élément fourni en paramètre de la collection si celle-ci le contient et renvoyer un booléen qui précise si la collection a été modifiée (l'implémentation de cette opération est optionnelle)

114

Exercices

Ecrivez un programme Java pour remplacer le deuxième élément d'un **ArrayList** par l'élément spécifié.

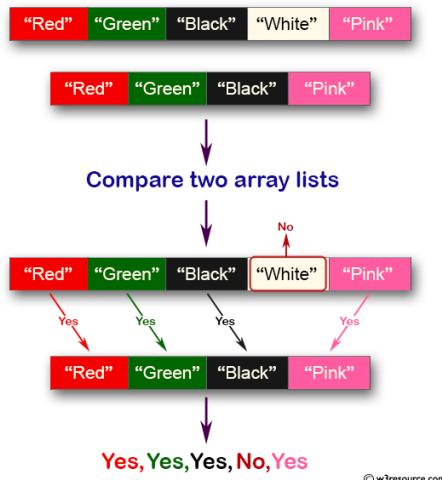


115

© w3resource.com

Exercices

Ecrivez un programme Java pour comparer deux ArrayList.



116

© w3resource.com



Plan de cours

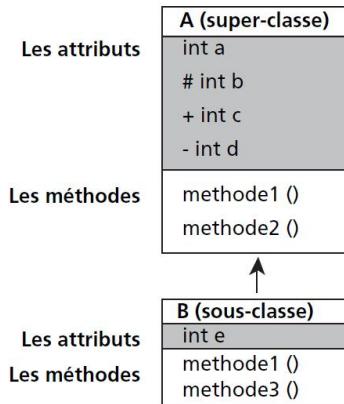
- Chapitre 1 : Présentation du langage JAVA
- Chapitre 2 : Classes JAVA
- **Chapitre 3 : Héritage**
- Chapitre 4 : Exceptions
- Chapitre 5 : Interfaces graphiques (Swing)

117



Chapitre 3 : Héritage JAVA

Héritage



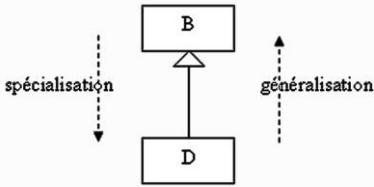
118

**Héritage**

Une classe Java dérive toujours d'une autre classe, Object quand rien n'est spécifié. Pour spécifier de quelle classe hérite une classe on utilise le mot-clé **extends** :

```
class D extends B {
    . . .
}
```

La classe D dérive de la classe B. On dit que la classe B est la super classe.

**modificateurs de visibilité :**

- **private**
- **protected**
- **Public**

La visibilité **protected** rend l'accès possible :

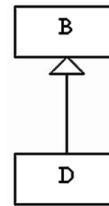
- depuis les classes dérivée.
- depuis les classes du paquetage.

119

**Héritage**

Une référence à une classe de base peut être affectée d'une référence à une classe dérivée, l'inverse doit faire l'objet d'une opération de conversion de type ou cast :

- B b = new B();
- D d = new D();
- b = d; // OK
- d = b; // interdit
- d = (D) b ; // OK avec cast



Remarque : Cette dernière instruction peut lever une exception si le cast est impossible.

120

Constructeurs et héritage

```
class X{
    int x;
    public X (int x){
        this.x = x;
    }
}
```

```
class Y extends X{
}
```

provoque l'erreur de compilation : le super constructeur implicite X() n'est pas défini pour le constructeur par défaut.

Chaque instance est munie de deux références particulières :

- **this** réfère l'instance elle-même.
- **super** réfère la partie héritée de l'instance.

121

Opérateur instanceof

L'opérateur *instanceof* permet de savoir à quelle classe appartient une instance :

```
class B{ ...}
class D extends B{...}
class C {...}
    B b = new B();
    D d = new D();
    C c = new C();
```

```
b instanceof B
// true
b instanceof D
// false
d instanceof B
// true
d instanceof D
// true
c instanceof B
// erreur de compilation Erreur No. 365 : // impossible de comparer C avec B
```

122

Opérateur instanceof

la méthode equals(Object o), pour une classe X, est définie, en général, de la façon suivante :

```
public boolean equals(Object o){  
    if(this==o) return true;  
    if(!(o instanceof X)) return false;  
    X x = (X)o;  
    ...  
}
```

123

Redéfinition des méthodes

```
class B {  
    void m(){  
        System.out.println( "méthode m de B");  
    }  
}  
class D extends B{  
}  
  
B b=new B();  
D d=new D();  
b.m();  
b = d;  
b.m();
```

Affiche deux fois
m de B

```
class B {  
    void m(){  
        System.out.println( "méthode m de B");  
    }  
}  
class D extends B{  
    void m(){  
        System.out.println( "méthode m de D");  
    }  
}
```

```
B b=new B();  
D d=new D();  
b.m();  
b=d; b.m();
```

Affiche m de
B, puis m de D

124

**Redéfinition vs surcharge****V.S**

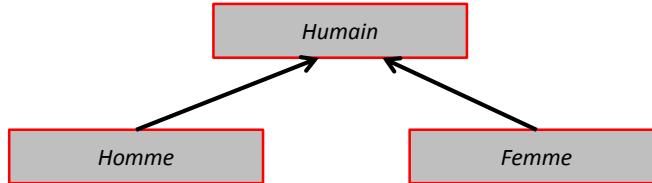
```
class vehicule {
    public void start(){
        System.out.println("démarrez le véhicule");
    }
}
class voiture extends vehicule {
    public void stop(){
        System.out.println("arrêtez la voiture");
    }
    public void start(){
        System.out.println("démarrez la voiture");
    }
}
```

```
class voiture {
    public void start(){
        System.out.println("démarrez la voiture");
    }
    public void start(int num){
        for (int i=0;i<num;i++)
            System.out.println("démarrez la voiture"+i);
    }
}
```

- La surcharge survient lorsque deux méthodes ou plus dans une classe ont le même nom de méthode mais des paramètres différents.
- La redéfinition signifie avoir deux méthodes avec le même nom et les mêmes paramètres, l'une des méthodes est dans la classe mère et l'autre dans la classe fille.

125

Humain

**Héritage : Classe abstraite**

Une classe définie avec le modificateur abstract est une classe abstraite qui ne peut pas produire d'instance.

- Sa définition peut contenir des méthodes abstraites.
- Une classe qui contient une méthode abstraite doit être abstraite.
- Une méthode abstraite est une méthode définie uniquement par son prototype.
- Le rôle des classes abstraites est la factorisation de fonctionnalités communes à plusieurs classes dérivées.

126

Héritage : Exemple

Supposons que nous définissions la classe suivante :

```
public class Employe{
    protected String nom ;
    protected double salaire ;
    public Employe ( String nom, double salaire){
        this.nom = nom ; this.salaire = salaire ;
    }
    publique Chaîne getNom () { retour nom;}
    publique à double getSalaire () { retour salaire;}
    publique String toString () { retourner getNom () + "" + getSalaire () ; }
}
```

127

Héritage : Exemple

Un chef «est un» employé qui a une prime en plus du salaire :

```
public class Chef extends Employe{
    private double prime;
    public Chef ( String nom, double salaire, double prime){
        super( nom, salaire) ;
        this.prime = prime;
    }
    public double getSalaire() {return salaire + prime;}
}
```

La méthode **getSalaire** pourrait être (mieux ?) programmée de la façon suivante : cela permettrait de changer les accès de nom et salaire en private.

```
publique à double getSalaire () { retour de super .getSalaire () + prime;}
```

128

Héritage : Exemple

Une entreprise est un tableau d'employés :

```
Employe entreprise[] = new Employe[5] ;
entreprise[0]= new Chef("Cronos", 1000, 500) ;
entreprise[1]= new Chef("Zeus ", 1000, 600) ;
entreprise[2]= new Employe("Ares", 620) ;
entreprise[3]= new Employe ("Apollon", 700) ;
entreprise[4]= new Employe ("Aphrodite ", 100) ;
```

L'affichage des noms et salaires des différents employés se fera de la façon suivante :

```
pour ( int i = 0; i <5; i ++ ) {
    si (entreprise [i] instanceof Chef) System.out.print ("Chef:");
    System.out.println (entreprise [i] .toString ());
}
```

129

Héritage : Exemple

Le calcul de la somme des salaires des employés qui ne sont pas des chefs ?

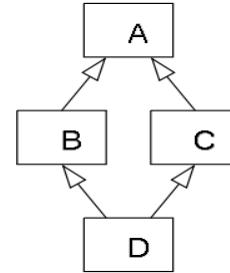
```
doubles sommeSalaires = 0;
pour ( int i = 0; I <5; ++ i)
    si (! (entreprise [i] instanceof Chef)) sommeSalaires += entreprise [i] .getSalaire ();
```

130

**Héritage multiple**

Un diagramme d'héritage en diamant. En informatique, le **problème du diamant** (ou **problème du losange** dans certains articles scientifiques) arrive principalement en **programmation orientée objet**, lorsque le langage permet l'héritage multiple

Si une classe D hérite de deux classes B et C, elles-mêmes filles d'une même classe A, se pose un problème de conflit lorsque des fonctions ou des champs des classes B et C portent le même nom. Le nom de ce problème provient de la forme du schéma d'héritage des classes A, B, C et D dans ce cas.



131

**Héritage multiple : exemple**

Par exemple, dans le cas d'une interface graphique, une classe **Button** pourrait hériter de deux classes :

- **Rectangle** (gérant son apparence)
- **Clickable** (gérant les clics de souris)

et ces deux classes hériter d'une classe **Object**.

Si la classe **Object** définit la fonction **equals** (gérant la comparaison entre objets), que les deux sous-classes **Rectangle** et **Clickable** étendent cette fonction pour l'adapter à leurs particularités, laquelle des fonctions **equals** de **Rectangle** ou de **Clickable** la classe **Button** doit-elle utiliser ?

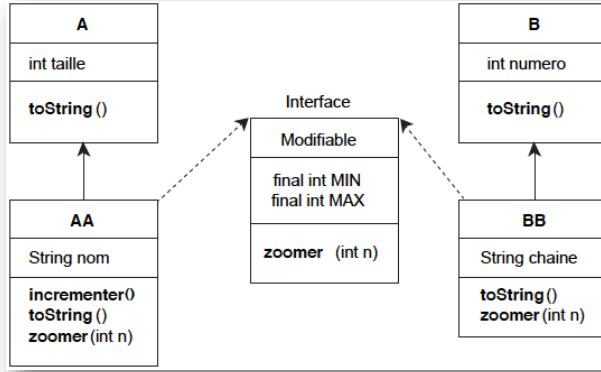
Choisir arbitrairement une seule des deux fonctions ferait perdre l'intérêt de l'héritage

132

**Héritage : Interface**

Java n'autorise que l'héritage simple. Une classe ne peut avoir qu'une seule superclasse. Une certaine forme d'héritage multiple est obtenue grâce à la notion d'interface.

Une interface définit uniquement des constantes et des prototypes de méthodes.



133

**Héritage : Votre première interface**

Pour définir une interface, au lieu d'écrire :

```
public class A{ }
```

... il vous suffit de faire :

```
public interface I{ }
```

vous venez d'apprendre à déclarer une interface. Vu qu'une interface est une classe 100 % abstraite, il ne vous reste qu'à y ajouter des méthodes abstraites, mais sans le mot clé abstract

134

Héritage : Votre première interface

```
public interface I{
    public void A();
    public String B();
}
```

```
public interface I2{
    public void C();
    public String D();
}
```

Et pour faire en sorte qu'une classe utilise une interface, il suffit d'utiliser le mot clé *implements*. Ce qui nous donnerait :

```
public class X implements I{
    public void A(){
        ...
    }
    public String B(){
        ...
    }
}
```

**la classe X implémente
l'interface I.**

135

Héritage : Votre première interface

vous pouvez implémenter plusieurs interfaces; ci-dessous un exemple:

```
public class X implements I, I2{
    public void A(){
        ...
    }
    public String B(){
        ...
    }
    public void C(){
        ...
    }
    public String D(){
        ...
    }
}
```

136

**Héritage : Votre première interface**

Par contre, lorsque vous implémentez une interface, vous devez *obligatoirement* redéfinir les méthodes de l'interface ! Ainsi, le polymorphisme vous permet de faire ceci :

```
public static void main(String[] args){
    // Avec cette référence, vous pouvez utiliser les méthodes de l'interface I
    I var = new X();
    //Avec cette référence, vous pouvez utiliser les méthodes de l'interface I2
    I2 var2 = new X();
    var.A();
    var2.C();
}
```

137

**Héritage : Interface**

Java 8 : du neuf dans les interfaces !



138

**Héritage : Interface****Java 8 : du neuf dans les interfaces !**

En Java 7 et antérieur, une méthode déclarée dans une interface ne fournit pas d'implémentation. Ce n'est qu'une signature, un contrat auquel chaque classe dérivée doit se conformer en fournissant une implémentation propre.

Mais il arrive que plusieurs classes similaires souhaitent partager une même implémentation de l'interface.

Java 8 propose d'ajouter deux éléments supplémentaires dans une interfaces : des méthodes statiques et des méthodes par défaut.

139

**Héritage : Interface**

La syntaxe est simple et sans surprises : il suffit de fournir un corps à la méthode, et de la qualifier avec le mot-clé **default**.

```
public interface Foo {
    public default void foo() {
        System.out.println("Default implementation of foo()");
    }
}
```

Les classes filles sont alors libérées de l'obligation de fournir elles-mêmes une implémentation à cette méthode - en cas d'absence d'implémentation spécifique, c'est celle par défaut qui est utilisée.

140

Héritage : Interface**Exercice**

Certain animaux peuvent crier, d'autres sont muets. On représentera le fait de crier au moyen d'une méthode affichant à l'écran le cri de l'animal.

- écrire une interface contenant la méthode permettant de crier.
- écrire les classes des chats, des chiens et des lapins (qui sont muets)
- écrire un programme avec un tableau pour les animaux qui savent crier, le remplir avec des chiens et des chats, puis faire crier tous ces animaux. Décrire ce qui s'affiche à l'écran à l'exécution de ce programme.

141

Héritage : Interface

```

interface Criant{
    void crier();
}
class Chat implements Criant{
    public void crier(){
        Terminal.ecrireStringln("maou");
    }
}
class Chien implements Criant{
    public void crier(){
        Terminal.ecrireStringln("wouf");
    }
}
class Lapin{
    public void froncerDuNez(){
    }
}

```

```

public class Animaux{
    public static void main(String[] a){
        Criant[] tab = new Criant[4];
        tab[0] = new Chat();
        tab[1] = new Chien();
        tab[2] = new Chat();
        tab[3] = new Chien();
        for (int i=0; i<4; i++){
            tab[i].crier();
        }
    }
}

```

142



Plan de cours

- ❑ Chapitre 1 : Présentation du langage JAVA
- ❑ Chapitre 2 : Classes JAVA
- ❑ Chapitre 3 : Héritage
- ❑ Chapitre 4 : Exceptions
- ❑ Chapitre 5 : Interfaces graphiques (Swing)
- ❑ Chapitre 6 : JDBC

143



Chapitre 4 : Exceptions JAVA

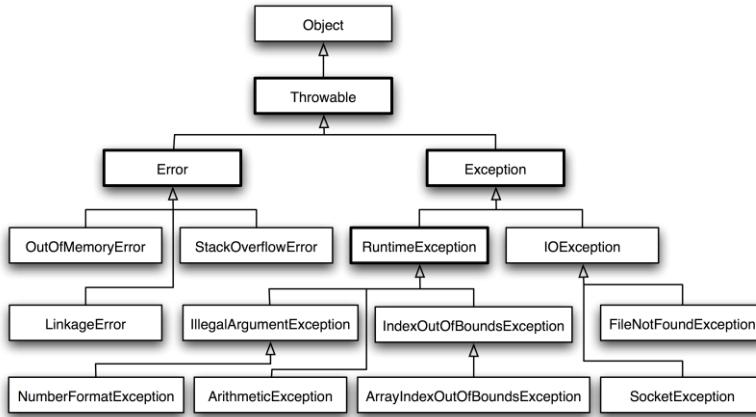
Exceptions

- Mécanisme permettant de traiter les comportements exceptionnels.
- Constituent un moyen fiable et pratique d'effectuer un contrôle des erreurs rigoureux sans alourdir le code.
- La notion d'erreur fait partie intégrante de la fonction

En Java, la fonction peut lancer une exception en indiquant un type d'exception.

- L'exception se "propage" de retour de méthode en retour de méthode jusqu'à ce qu'une méthode la traite.
- Si aucune méthode ne la prend en compte, il y a un message d'erreur d'exécution.
- Suite à une exception, toutes les instructions sont ignorées, sauf les instructions chargées de capter cette exception.

144



145

ExceptionsError Vs ExceptionDifférences clés entre Exception et Error :

- 1) En cas **d'exception**, nous pouvons le gérer en utilisant le bloc `try/catch`. Si une **erreur** se produit, nous ne pouvons pas la gérer, l'exécution du programme sera terminée.
- 2) Une **erreur** se produit uniquement lorsque les ressources du système sont déficientes alors qu'une **exception** est provoquée si un code présente un problème.
- 3) Une **erreur** ne peut jamais être récupérée alors qu'une **exception** peut être récupérée en préparant le code pour gérer **l'exception**.
- 4) Si une **erreur** est survenue, le programme sera terminé de manière anormale. En revanche, si **l'exception** se produit, le programme lancera une **exception** et le traitera à l'aide du bloc `try/catch`.
- 5) Les **erreurs** sont de type non contrôlé, c'est-à-dire que les compilateurs ne connaissent pas **l'erreur**, tandis qu'une **exception** est classée comme vérifiée et non contrôlée.
- 6) Les erreurs sont définies dans `java.lang.Error` alors que les **exceptions** sont définies dans `java.lang.Exception`.
- 7) Les **exceptions** sont liées à l'application, tandis que les **erreurs** sont liées à l'environnement dans lequel l'application est exécutée.

146

Les exceptions de type RunTimeException

L'exécution du programme suivant provoque un message d'erreur indiquant une exception du type *ArrayIndexOutOfBoundsException*, et l'arrêt du programme. Le tableau de 4 entiers tabEnt a des indices de 0 à 3. L'accès à la valeur d'indice 5 provoque une exception non captée.

```
public class TestException1 {
    public static void main (String[] args) {
        int tabEnt[] = { 1, 2, 3, 4 };
        // Exception de type java.lang.ArrayIndexOutOfBoundsException
        // non contrôlée par le programme; erreur d'exécution
        tabEnt [4] = 0; // erreur et arrêt
        System.out.println ("Fin du main"); // le message n'apparaît pas
    } // main
} // class TestException1
```

147

Les exceptions de type RunTimeException

Dans le programme suivant, la même erreur d'accès à un élément est captée par le programme. Lors de la rencontre de l'erreur, les instructions en cours sont abandonnées et un bloc de traitement (*catch*) de l'exception est recherché et exécuté. Le traitement se poursuit après ce bloc sauf indication contraire du catch (return ou arrêt par exemple). Le message "Fin du main" est écrit après interception de l'exception.

```
public class TestException2 {
    public static void main (String[] args) {
        try {
            int tabEnt[] = { 1, 2, 3, 4 };
            // Exception de type ArrayIndexOutOfBoundsException
            // contrôlée par le programme
            tabEnt [4] = 0; // Exception et suite au catch
            System.out.println ("Fin du try"); // non exécutée
        } catch (Exception e) {
            System.out.println ("Exception : " + e); // écrit le message
        }
        System.out.println ("Fin du main"); // exécutée
    } // class TestException2
```

Exemples de résultats d'exécution :

```
Exception : java.lang.ArrayIndexOutOfBoundsException
Fin du main
```

148

**Exemple2 : les débordements de tableaux (dans une méthode)**

```

class TestException3 {
    int[] tabEnt = new int[4];
    void initTableau () {
        tabEnt[0] = 1; tabEnt[1] = 2;
        tabEnt[2] = 3; tabEnt[3] = 4;
        tabEnt[4] = 5; // l'exception se produit ici; non captée
        System.out.println("Fin de initTableau"); // non exécutée
    }
    void ecrireTableau () {
        for (int i=0; i < tabEnt.length; i++) {
            System.out.print (" " + i);
        }
        System.out.println();
    }
} // class TestException3

public class PPTTestException3 {
    public static void main (String[] args) {
        try {
            TestException3 te = new TestException3();
            te.initTableau(); // l'exception se produit dans initTableau
            te.ecrireTableau(); // non exécutée
        } catch (Exception e) { // exception captée ici
            System.out.println ("Exception : " + e); // exécutée
        }
        System.out.println ("Fin de main"); // exécutée
    } // main
} // class PPTTestException3

```

l'instruction **tabEnt[4]** provoque une exception dans la méthode **initTableau()**. L'instruction qui suit n'est pas prise en compte ; l'exception n'est pas traitée dans **initTableau()**. Les instructions qui suivent l'appel de **initTableau()** dans **main()** (le programme appelant) sont ignorées également. Par contre, il existe un bloc **catch** qui traite l'exception. Le message "Fin de main" est lui pris en compte après le traitement de l'exception.

Exception :
java.lang.ArrayIndexOutOfBoundsException
Fin de main

149

**Exemple2 : les débordements de tableaux (dans une méthode)**

L'exception peut aussi être traitée dans **initTableau()** comme indiqué ci-dessous. Le programme se poursuit normalement une fois le bloc **catch** exécuté (retour de **initTableau()**, appel de **ecrireTableau()** et message de fin).

```

class TestException4 {
    int[] tabEnt = new int[4];
    void initTableau () {
        try {
            tabEnt[0] = 1; tabEnt[1] = 2; tabEnt[2] = 3; tabEnt [3] = 4; tabEnt[4] = 5;
            // l'exception se produit ici
        }
        catch (Exception e) {
            // captée ici
            System.out.println ("Exception : " + e); // écrit le message
        }
        System.out.println("Fin de initTableau"); // exécutée
    }
    void ecrireTableau () { for (int i=0; i < tabEnt.length; i++) {
        System.out.print (" " + tabEnt[i]); } System.out.println();
    }
} // class TestException4

public class PPTTestException4 {
    public static void main (String[] args) { TestException4 te = new
        TestException4(); te.initTableau(); //Exception : java.lang.ArrayIndexOutOfBoundsException
        ecrireTableau(); // exécutée System.out.println("Fin de initTableau")
    } // main
} // class PPTTestException4

```

ecrireTableau 150

Fin de main

**Autres exemples d'exceptions de type RunTimeException**

L'instructions suivante provoque une exception de type **NullPointerException**. s1 est une référence *null* sur un (futur) objet de type String. On ne peut pas accéder au troisième caractère de l'objet String référencé par *null*, d'où l'exception.

```
public class TestException{
    public static void main (String[] args) {
        String s1 = null;
        //exception NullPointerException ci-dessous
        System.out.println (s1.charAt(3));
    }
}

Exception in thread "main" java.lang.NullPointerException
at TestException.main(TestException.java:5)
```

151

**Autres exemples d'exceptions de type RunTimeException**

L'instructions suivante provoque une exception de type **NumberFormatException** . Le programme suivant donne des exemples où une exception est lancée : chaîne à convertir en entier ne contenant pas un entier (lecture erronée d'un entier par exemple), .

```
public class TestException{
    public static void main (String[] args) {
        int n2 = Integer.parseInt ("231x");
        // exception NumberFormatException
        System.out.println ("n2 : " + n2);
    }
}

Exception in thread "main" java.lang.NumberFormatException: For input string: "231x"
at java.lang.NumberFormatException.forInputString(NumberFormatException.java:65)
at java.lang.Integer.parseInt(Integer.java:580)
at java.lang.Integer.parseInt(Integer.java:615)
at TestException.main(TestException.java:3)
```

152

**Autres exemples d'exceptions de type RunTimeException**

L'instructions suivante provoque une exception de type **ArithmetiException**. Le programme suivant donne des exemples où une exception est lancée : division d'un entier par la valeur zéro,

```
public class TestException{
    public static void main (String[] args) {
        int d = 0;
        int quotient = 17 / d;
        // exception de type ArithmetiException
        System.out.println ("quotient : " + quotient);
    }
}

Exception in thread "main" java.lang.ArithmetiException: / by zero
at TestException.main(TestException.java:4)
```

153

**Autres exemples d'exceptions de type RunTimeException**

L'instructions suivante provoque une exception de type **ClassCastException**. Le programme suivant donne des exemples où une exception est lancée : transtypage non cohérent de deux objets de classes différentes.

```
public class TestException{
    public static void main (String[] args) {
        Object p = new Point();
        // exception ClassCastException
        Color coul = (Color) p;
    }
}
```

```
Exception in thread "main" java.lang.ClassCastException: Point cannot be cast to
Color
at TestException.main(TestException.java:6)
```

154

**La syntaxe des exceptions****Les mots clés try, catch et finally**

```

try {
    opération_risquée1;
    opération_risquée2;
} catch (ExceptionInteressante e) {
    traitements
} catch (ExceptionParticulière e) {
    traitements
} catch (Exception e) {
    traitements
} finally {
    traitement_pour_terminer_proprement;
}

public class TestException{
    public static void main(java.lang.String[] args) {
        // Insert code to start the application here.
        int i = 3;
        int j = 0;
        try {
            System.out.println("résultat = " + (i / j));
        }
        catch (Exception e) {
        }
        catch (ArithmeticsException e) {
        }
    }
}

```

155

**La syntaxe des exceptions****Lève une exception**

Une méthode qui lève une exception doit donner la liste des nouvelles classes d'exception qu'elle peut déclencher. Seules les exceptions des classes Error, RuntimeException et de leurs dérivées n'ont pas à être citées.

```

void méthode(...) throws E1, E2{
    // du code qui peut lever
    // des exceptions de la classe E1 ou E2
    // sans les capturer
}

```

156

**La syntaxe des exceptions****Exceptions déclenchées par l'application**

Il est également possible de déclencher nos propres exceptions, qu'elles soient instances de classes fournies par l'API Java, ou de nos propres classes. Voyons cela sur l'exemple suivant (déclenchement manuel d'une exception)

```
if (temperature > 40) {
    throw (new Exception("Il fait trop chaud !"));
}
```

```
public FileReader(String fileName) throws FileNotFoundException
```

Nb: On notera la différence entre la génération manuelle d'une exception qui utilise le mot-clé **throw** (sans s), et la déclaration au niveau de la signature d'une méthode, qui utilise le mot-clé **throws** (avec un s).

157

**La définition et le lancement d'exception**

Il existe de nombreuses classes d'exceptions prédéfinies en Java, que l'on peut classer en trois Catégories :

- Celles définies en étendant la classe **Error** : elles représentent des erreurs critiques qui ne sont pas censée être gérées en temps normal. Par exemple, une exception de type **OutOfMemoryError** est lancée lorsqu'il n'y a plus de mémoire disponible dans le système.
- Celles définies en étendant la classe **Exception** : elles représentent les erreurs qui doivent normalement être gérées par le programme. Par exemple, une exception de type **IOException** est lancée en cas d'erreur lors de la lecture d'un fichier
- Celles définies en étendant la classe **RuntimeException** : elles représente des erreurs pouvant éventuellement être gérée par le programme. L'exemple typique de ce genre d'exception est **NullPointerException**, qui est lancée si l'on tente d'accéder au contenu d'un tableau ou d'un objet qui vaut null.

158



Plan de cours

- ❑ Chapitre 1 : Présentation du langage JAVA
- ❑ Chapitre 2 : Classes JAVA
- ❑ Chapitre 3 : Héritage
- ❑ Chapitre 4 : Exceptions
- ❑ Chapitre 5 : Interfaces graphiques (Swing)
- ❑ Chapitre 6 : JDBC

159



Chapitre 5 : Interfaces graphiques (Swing)

Les packages IHM de Java

- ❑ **java.awt** : (awt : abstract window toolkit, java1.1) bas niveau. Anciens composants, dits composants lourds (ils sont opaques et affichés en dernier).
- ❑ **javax.swing** : java 1.2. Composants légers = pas de code natif (écrits entièrement en java).
- ❑ **SWT**: la version IBM utilisée dans Eclipse
- ❑ **JavaFX** est une bibliothèque d'interface utilisateur issue du projet OpenJFX, qui permet aux développeurs Java de créer une interface graphique
- ❑ La librairie Swing est totalement gérée par la machine virtuelle ce qui permet de ne plus se limiter au sous-ensemble de composants graphiques communs à toutes les plates-formes.
- ❑ Swing définit de nouveaux composants. Cependant, la bibliothèque Swing nécessite plus de temps CPU, car la machine virtuelle doit dessiner les moindres détails des composants

160



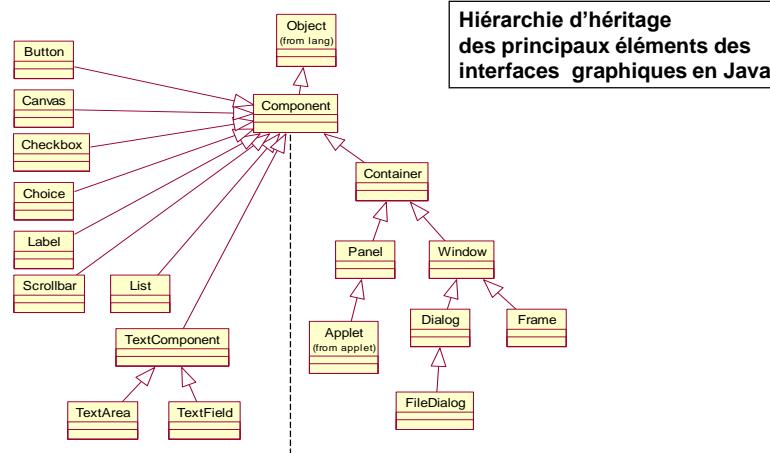
Conteneurs et composants (1)

- Une interface graphique en Java est un assemblage conteneurs (*Container*) et de composants (*Component*).
- **Un composant** est une partie "visible" de l'interface utilisateur Java.
 - ◆ C'est une sous-classes de la classe abstraite **java.awt.Component**.
 - ◆ Exemple : les boutons, les zones de textes ou de dessin, etc.
- **Un conteneur** est un espace dans lequel on peut positionner plusieurs composants.
 - ◆ Sous-classe de la classe **java.awt.Container**
 - ◆ La classe Container est elle-même une sous-classe de la classe Component
 - ◆ Par exemple les fenêtres, les applets, etc.

161



Conteneurs et composants (1)



162



Conteneurs et composants (1)

- ❑ Les deux conteneurs les plus courants sont le **Frame** et le **Panel**.
- ❑ Un Frame représente une fenêtre de haut niveau avec un titre, une bordure et des angles de redimensionnement.
 - ◆ La plupart des applications utilisent au moins un Frame comme point de départ de leur interface graphique.
- ❑ Un Panel n'a pas une apparence propre et ne peut pas être utilisé comme fenêtre autonome.
 - ◆ Les Panels sont créés et ajoutés aux autres conteneurs de la même façon que les composants tels que les boutons
 - ◆ Les Panels peuvent ensuite redéfinir une présentation qui leur soit propre pour contenir eux-mêmes d'autres composants.

163



Conteneurs et composants (1)

- ❑ On ajoute un composant dans un conteneur, avec la méthode add() :


```
Panel p = new Panel();
Button b = new Button();
p.add(b);
```
- ❑ De manière similaire, un composant est retirer de son conteneur par la méthode remove() :


```
p.remove(b);
```
- ❑ Un composant a (notamment) :
 - ◆ une taille préférée que l'on obtient avec **getPreferredSize()**
 - ◆ une taille minimum que l'on obtient avec **getMinimunSize()**
 - ◆ une taille maximum que l'on obtient avec **getMaximunSize()**

164



Conteneurs et composants (1)

```
import java.awt.*;
public class EssaiFenetre1
{
    public static void main(String[] args)
    {
        Frame f = new Frame("Ma première fenêtre");
        Button b = new Button("coucou");
        f.add(b);
        f.pack();
        f.show();
    }
}
```

Création d'une fenêtre (un objet de la classe Frame) avec un titre

Création du bouton ayant pour label « coucou »

Ajout du bouton dans la fenêtre

On demande à la fenêtre de choisir la taille minimum avec pack() et de se rendre visible avec show()

165



Swing Composants graphiques

- ❑ Un composant graphique léger (en anglais, *lightweight GUI component*) est un composant graphique indépendant du gestionnaire de fenêtre local.
 - ◆ Un composant léger ressemble à un composant du gestionnaire de fenêtre local.
 - ◆ Un bouton léger est un rectangle dessiné sur une zone de dessin qui contient une étiquette et réagit aux événements souris.
 - ◆ Tous les composants de Swing, exceptés **JApplet**, **JDialog**, **JFrame** et **JWindow** sont des composants légers.

166

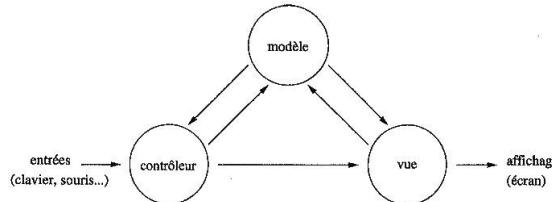
**Architecture MVC**

La conception des classes Swing s'appuie assez librement sur l'architecture MVC (Modèle/Vue/Contrôleur).

Le patron modèle-vue-contrôleur (en abrégé MVC, de l'anglais model-view-controller), tout comme les patrons modèle-vue-présentation ou Présentation, abstraction, contrôle, est un modèle destiné à répondre aux besoins des applications interactives en séparant les problématiques liées aux différents composants au sein de leur architecture respective.

Ce paradigme regroupe les fonctions nécessaires en trois catégories :

- un modèle (modèle de données) ;
- une vue (présentation, interface utilisateur) ;
- un contrôleur (logique de contrôle, gestion des événements, synchronisation).



167

**Modèle**

- ❑ Le modèle représente le cœur (algorithmique) de l'application : traitements des données, interactions avec la base de données, etc.
- ❑ La base de données sera l'un de ses composants. Le modèle comporte des méthodes standards pour mettre à jour ces données (insertion, suppression, changement de valeur). Il offre aussi des méthodes pour récupérer ces données.
- ❑ Le modèle peut autoriser plusieurs vues partielles des données. Si par exemple le programme manipule une base de données pour les emplois du temps, le modèle peut avoir des méthodes pour avoir tous les cours d'une salle, tous les cours d'une personne ou tous les cours d'un groupe de TD.

168

Vue

Ce avec quoi l'utilisateur interagit se nomme précisément la vue.

- Sa première tâche est de présenter les résultats renvoyés par le modèle.
- Sa seconde tâche est de recevoir toute action de l'utilisateur (clic de souris, sélection d'un bouton radio, entrée de texte, de mouvements, de voix, etc.). Ces différents événements sont envoyés au contrôleur.
- La vue n'effectue pas de traitement, elle se contente d'afficher les résultats des traitements effectués par le modèle et d'interagir avec l'utilisateur.

169

Contrôleur

- Le contrôleur prend en charge la gestion des événements de synchronisation pour mettre à jour la vue ou le modèle et les synchroniser.
- Il reçoit tous les événements de la vue et déclenche les actions à effectuer.
- Si une action nécessite un changement des données, le contrôleur demande la modification des données au modèle afin que les données affichées se mettent à jour.
- Le contrôleur n'effectue aucun traitement, ne modifie aucune donnée. Il analyse la requête du client et se contente d'appeler le modèle adéquat et de renvoyer la vue correspondant à la demande.

170



Chapitre 5 : Interfaces graphiques (Swing)

Swing

Une interface graphique Swing est un arbre qui part d'un objet du système (heavyweight):

- Une fenêtre en java est représentée par la classe Window (package java.awt). Mais on utilise rarement Window directement car cette classe ne définit ni bords ni titre pour la fenêtre.
 - Dans une application Swing standard, on instanciera un cadre JFrame qui permet d'avoir une fenêtre principale avec un titre et une barre de menu
-
- Les noms de composants Swing commencent (souvent) par J
 - tous héritent du composant graphique Jcomponent
 - JComponent hérite du composant graphique AWT Container qui permet à un composant d'en contenir d'autres

171



Chapitre 5 : Interfaces graphiques (Swing)

Swing JFrame

JFrame est la classe permettant de faire une «application». Une instance de JFrame est composée d'un JRootPane lui même composé de :

exemple d'application

- LayeredPane
 - MenuPane : le menu
 - ContentPane : les composants
- GlassPane : peut servir à intercepter des événements souris
- décorations
 - Bordure
 - Titre
 - boutons d'iconification et fermeture
 - icône



Titre de l'application
Barre de menu
Une zone de saisie de texte

172

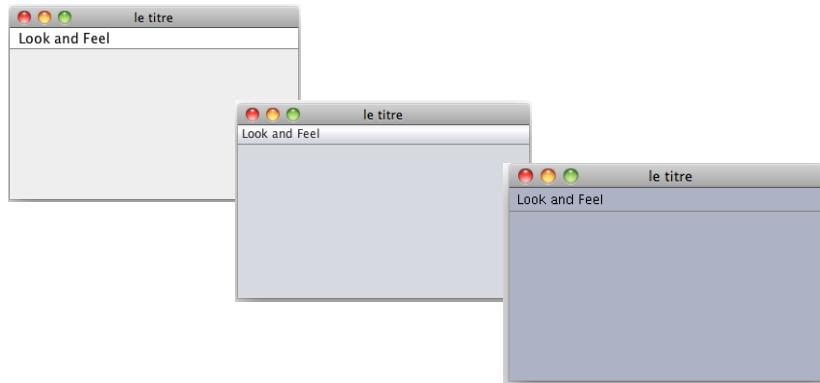


Chapitre 5 : Interfaces graphiques (Swing)

Swing : JFrame

Visualisation des apparences

Une application a une apparence (metal, motif, Windows, Windows classic, Nimbus, etc ...), l'exemple suivant permet de visualiser les différents «look and feel» d'une application (exécuter)



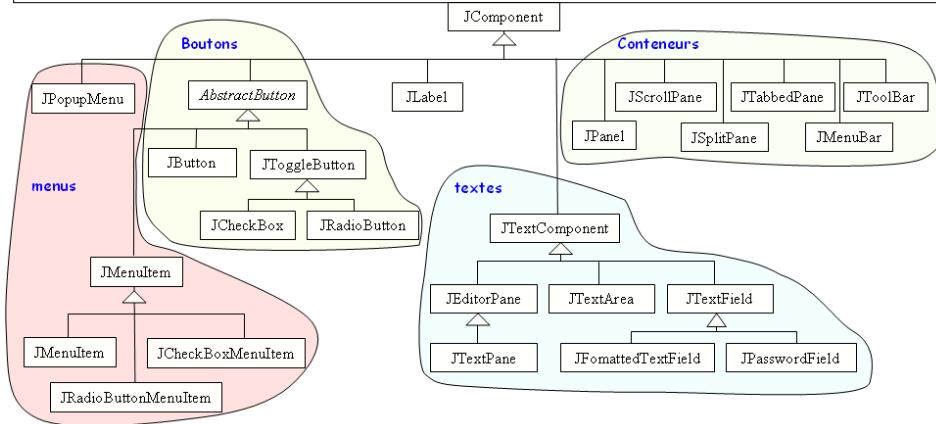
173



Chapitre 5 : Interfaces graphiques (Swing)

Swing : JComponent

JComponent est la classe de base de tous les composants Swing, à part JFrame, JDialog et Japplet.



174



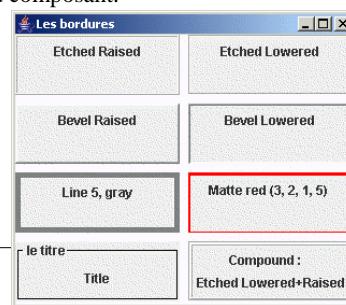
Swing : JComponent

Un JComponent a les caractéristiques suivantes :

- ❑ un ToolTipText : un texte de conseil au survol du composant par la souris. Les deux méthodes getToolTipText et setToolTipText permettent de manipuler ce texte, qui peut être du texte brut, ou du texte HTML.
- ❑ une bordure, et une façon de se dessiner : la bordure est manipulée par les méthodes setBorder(Border b) et getBorder() et la façon de se dessiner est définie par la méthode void paintComponent(Graphics g) du composant.

les bordures prédefinies sont :

- Etched
- Bevel
- Line
- Matte
- Title
- Compound



175



Swing : JComponent

- ❑ Un composant est soit opaque soit transparent : void setOpaque(boolean b)
 - lorsqu'il est opaque l'affichage se fait en appelant d'abord paintComponent, puis paintBorder et enfin paintChildren.
 - lorsqu'il est transparent, il faut d'abord afficher les composants qui sont dessous.
- ❑ une taille minimale, une taille maximale un alignement en X et en Y : toutes ces propriétés sont utilisées par le LayoutManager qui place le composant dans l'interface.
- ❑ un nom : setName(String n) peut être utile pour des composants n'affichant pas de texte.
- ❑ des propriétés : une propriété est un couple (nom, valeur). Les propriétés sont utilisées par exemple pour spécifier les contraintes pour certains gestionnaires de placement.
- ❑ Un composant peut être actif : setEnabled(true) ou inactif.
- ❑ ...

176

[Principaux paquetages Swing](#)

- javax.swing
 - ◆ le paquetage général
- javax.swing.border
 - ◆ pour dessiner des bordures autour des composants
- javax.swing.colorchooser
 - ◆ classes et interfaces utilisées par le composant JColorChooser
- javax.swing.event
 - ◆ les événements générés par les composants Swing
- javax.swing.filechooser
 - ◆ classes et interfaces utilisées par le composant JFileChooser

177

[Principaux paquetages Swing](#)

- javax.swing.table
 - ◆ classes et interfaces pour gérer les JTable
- javax.swing.text
 - ◆ classes et interfaces pour la gestion des composants « texte »
- javax.swing.tree
 - ◆ classes et interfaces pour gérer les JTree
- javax.swing.undo
 - ◆ pour la gestion de undo/redo dans une application

178

[Swing : événements](#)

Les composants Swing créent des événements, soit directement, soit par une action de l'utilisateur sur le composant. Ces événements peuvent déclencher une action exécutée par d'autre(s) composant(s).

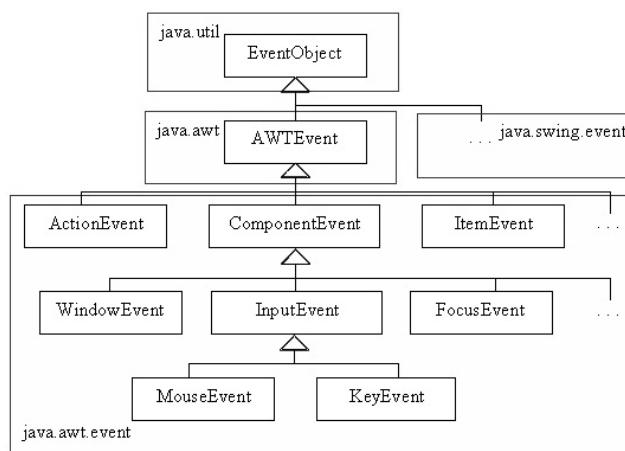
- Un composant qui crée des événements est appelé source. Le composant source délègue le traitement de l'événement au composant auditeur.
- Un composant qui traite un événement est appelé auditeur (listener)

Un composant auditeur doit s'inscrire auprès du composant source des événements qu'il veut traiter.

179

[Swing : événements](#)

Hiérarchie : Schéma de la hiérarchie simplifiée des événements java



180

**Swing : événements****ActionEvent**

<code>Object getSource()</code>	Retourne l'objet source de l'événement.
<code>int getID()</code>	Retourne le type d'événement.
<code>String getActionCommand()</code>	Retourne le texte associé au composant source de l'événement (bouton ou menu)
<code>int getModifiers</code>	Retourne un entier indiquant si les touches maj, alt ou ctrl étaient appuyées au moment de la génération de l'événement.
<code>long getWhen()</code>	Retourne la date et l'heure de la génération de l'événement.
<code>String paramString()</code>	Retourne une chaîne de caractères contenant toutes les informations précédentes.

```
interface ActionListener{
    void actionPerformed(ActionEvent e);
}
```

Les composants sources de ActionEvent sont :

- Boutons : JButton, JRadioButton, JCheckBox, JToggleButton
- Menus : JMenuItem, JMenu, JRadioButtonMenuItem, JCheckBoxMenuItem
- Texte : JTextField

181

**Swing : événements****MouseEvent**

<code>Object getSource()</code>	Retourne l'objet source de l'événement.
<code>int getID()</code>	Retourne le type d'événement.
<code>Point getPoint()</code>	Retourne les coordonnées de la souris lors de la génération de l'événement.
<code>int getX()</code>	Retourne la coordonnée en X de la souris lors de la génération de l'événement.
<code>int getY()</code>	Retourne la coordonnée en Y de la souris lors de la génération de l'événement.
<code>int getModifiers</code>	Retourne un entier indiquant si les touches maj, alt ou ctrl étaient appuyées au moment de la génération de l'événement.
<code>long getWhen()</code>	Retourne la date et l'heure de la génération de l'événement.
<code>int getButton()</code>	Retourne quel bouton a été cliqué. les trois valeurs possibles sont : <code>MouseEvent.BUTTON1, MouseEvent.BUTTON2, MouseEvent.BUTTON3</code>
<code>int getClickCount()</code>	Retourne le nombre de clics associés à cet événement.

```
interface MouseListener{
    void mouseClicked(MouseEvent e);
    void mouseEntered(MouseEvent e);
    void mouseExited(MouseEvent e);
    void mousePressed(MouseEvent e);
    void mouseReleased(MouseEvent e);
}
```

Tous les composants peuvent être sources de MouseEvent . Lors d'un clic les méthodes sont appelées dans l'ordre suivant :

- mousePressed
- mouseReleased
- mouseClicked

182



Chapitre 5 : Interfaces graphiques (Swing)

Swing : événements

KeyEvent

<code>Object getSource()</code>	Retourne l'objet source de l'événement.
<code>int getID()</code>	Retourne le type d'événement.
<code>char getKeyChar()</code>	Retourne le caractère correspondant à la touche du clavier, ou <code>java.awt.event.KeyEvent.CHAR_UNDEFINED</code> s'il n'y a pas de caractère associé à la touche (touche de fonction par exemple).
<code>int getKeyCode()</code>	Retourne le code du caractère correspondant à la touche du clavier.
<code>int getModifiers</code>	Retourne un entier indiquant si les touches maj, alt ou ctrl étaient appuyées au moment de la génération de l'événement.
<code>long getWhen()</code>	Retourne la date et l'heure de la génération de l'événement.
<code>String getKeyModifiersText(int > mod)</code>	Retourne le modifieur sous forme de chaîne de caractères : Maj+Crtl+Alt.

```
interface MouseListener{
    void keyTyped(KeyEvent e);
    void keyPressed(KeyEvent e);
    void keyReleased(KeyEvent e);
}
```

Tous les composants peuvent être sources de KeyEvent

183



Chapitre 5 : Interfaces graphiques (Swing)

Swing : événements

Adaptateur

Un adaptateur XxxAdapter est une classe abstraite qui fourni une implémentation de la classe XxxListener. Les méthodes implémentées sont toutes vides, et ainsi permettent à un utilisateur de l'adaptateur de ne redéfinir que le traitement qui l'intéresse.

Exemple : Un adaptateur qui ne redéfinit que KeyPressed et keyTyped :

```
composantSource.addKeyListener(new java.awt.event.KeyAdapter() {
    public void keyPressed(java.awt.event.KeyEvent e){
        ...
    }
    public void keyTyped(java.awt.event.KeyEvent e){
        ...
    }
});
```

184