

Support de cours Java

Structures de données et Programmation Orientée Objet

H. Mounier

Université Paris sud

2004/2005

Plan

- 1 Bases procédurales de Java
- 2 Notions de programmation orientée objet
- 3 Bases orientées objet de Java
- 4 Exceptions
- 5 Classes utilitaires de base

Plan (suite)

6 java.util : Conteneurs et autres utilitaires

Bases procédurales de Java

Références :

- The Java Language Specification,
J. Gosling, B. Joy et G. Steele
- Java in a Nutshell,
D. Flanagan

Variables et types de données

- Identificateurs
- Constantes
- Types primitifs (entiers, flottants, booléens)
- Tableaux

Identificateurs

- 1 **Identificateur** : suite de
 - lettres
 - minuscules ou majuscules,
 - chiffres,
 - underscore (`_`) et dollar (`$`).
- 2 Un identificateur **ne doit pas** commencer par un chiffre.
- 3 Java distingue minuscules et majuscules (`va`leur diffère de `VA`LEUR).
- 4 Conventions

Identificateurs (suite)

- Toute **méthode publique** et **variable d'instance** commence par une minuscule. Tout changement de mot descriptif se fait *via* une majuscule.
Exs. : `nextItem`, `getTimeOfDay`.
- **Variables locales** et **privées** : lettres minuscules avec des underscores. Exs. : `next_val`, `temp_val`.
- **Variables dites final** représentant des constantes : lettres majuscules avec underscores. Exs. : `DAY_FRIDAY`, `GREEN`.
- Tout **nom de classe** ou d'**interface** commence par une majuscule. Tout changement de mot descriptif se fait *via* une majuscule.
Exs. : `StringTokenizer`, `FileInputStream`.

Représentation littérale

1 Entiers :

- les valeurs octales commencent avec un 0.
- Ainsi 09 génère une erreur : 9 en dehors de la gamme octale 0 à 7.
- Ajouter un 1 ou L pour avoir un entier long.

2 Nombres à virgules : par défaut des double. Ajouter un f ou F pour avoir un float.

3 booléens : 2 valeurs possibles, true et false.

• true (resp. false) n'est pas égal à 1 (resp. 0).

4 Chaînes de caractères : doivent commencer et se terminer sur la même ligne ...

Types entiers et flottants

- ① Toute **assignation**, explicite ou par passage de paramètres, fait l'objet d'une **vérification de type**.
- ② Pas de coercition ou de conversion systématique.
Une différence de type est une erreur de compilation, pas un avertissement (warning).
- ③ Types de données entiers :
 - byte : à n'utiliser que pour des manipulations de bits.
 - short : relativement peu utilisé car sur 16 bits.
 - int : dans toute expression avec des byte, short, int, tous sont **promus** à des int avant calcul.

Plages de variation

Nom	Taille	Plage de variation
long	64 bits	-9 223 372 036 854 775 808... 9 223 372 036 854 775 807
int	32 bits	-2 147 483 648 ... 2 147 483 647
short	16 bits	-32 768 ... 32 767
byte	8 bits	-128 ... 127
double	64 bits	1.7e-308 ... 1.7e308
float	32 bits	3.4e-38 ... 3.4e+38

Transtypage (ou conversions, “cast”)

- 1 Conversions possibles en java.
- 2 Conversion automatique **seulement possible** lorsque le compilateur sait que la variable destination est assez grande.
- 3 Si des bytes, short, int et long font partie d'une expression, tout le monde est promu à long.
- 4 Si une expression contient un float et pas de double, tout le monde est promu à float. S'il y a un double, tout le monde est promu à double.
Tout littéral à virgule est considéré comme double.

Caractères

- 1 Un caractère est codé par un entier allant de 0 à 65536 (selon le standard unicode).
- 2 On peut se servir des caractères comme entiers :

```
int  trois  = 3;  
char un     = '1';  
char quatre = (char) (trois + un);
```

Dans quatre : '4'. un a été promu à int dans l'expression, d'où la conversion en char avant l'assignation.

Booléens

- 1 Type renvoyé par tous les opérateurs de comparaison, comme $(a < b)$.
- 2 Type **requis** par tous les opérateurs de contrôle de flux, comme `if`, `while` et `do`.

Tableaux

- 1 Création pouvant être faite en deux temps :
 - Déclaration de type, les [] désignant le type d'un tableau
`int tableau_entiers[] ;`
 - Allocation mémoire, *via new*
`tableau_entiers = new int[5] ;`
- 2 Pour les tableaux, la **valeur spéciale** `null` représente un tableau sans aucune valeur.
- 3 Initialisation
`int tableau_initialise[] = { 12, 34, 786 };`
- 4 Vérification par le compilateur de stockage ou de référence **en dehors des bornes** du tableau.

Tableaux multidimensionnels (I)

- 1 Tableaux multidimensionnels créés comme

```
double matrice[] [] = new double[4][4];
```

Ce qui revient à

```
double matrice[] [] = new double[4][];  
matrice[0] = new double[4];  
matrice[1] = new double[4];  
matrice[2] = new double[4];  
matrice[3] = new double[4];
```

Tableaux multidimensionnels (II)

- 1 Initialisation par défaut de tous les éléments à zéro.
- 2 Des expressions sont permises dans les initialisations de tableaux

```
double m[][] = {  
    { 0*0, 1*0, 2*0 },  
    { 0*1, 1*1, 2*1 },  
    { 0*2, 1*2, 2*2 }  
};
```


Opérateurs

- Opérateurs arithmétiques
- Opérateurs relationnels
- Opérateurs logiques

Opérateurs arithmétiques

Op.	Résultat	Op.	Résultat
+	addition	+=	assignation additive
-	soustraction	-=	assignation soustractive
*	multiplication	*=	assignation multiplicative
/	division	/=	assignation divisionnelle
%	modulo	%=	assignation modulo
++	incrémentatation	-	décrémentatation

- 1 Les opérateurs arithmétiques fonctionnent comme en C.
- 2 Une différence : le modulo agit également sur les nombres à virgule.

Opérateurs arithmétiques (suite)

Exemple d'incrémentation

```
class IncDec {  
    public static void main(String args[]) {  
        int a = 1;  
        int b = 2;  
        int c = ++b;  
        int d = a++;  
        c++;  
        System.out.println("a = " + a);  
        System.out.println("b = " + b);  
        System.out.println("c = " + c);  
        System.out.println("d = " + d);  
    }  
}
```

Opérateurs arithmétiques (suite)

La sortie du programme est

```
Prompt > javac IncDec  
Prompt > java IncDec  
a = 2  
b = 3  
c = 4  
d = 1
```

Opérateurs relationnels

Op.	Résultat
==	égal à
!=	différent de
>	strictement supérieur à
<	strictement inférieur à
>=	supérieur ou égal à
<=	inférieur ou égal à

- 1 Tout type java, y compris les types primitifs et les références à des instances d'objets peuvent être comparés avec == et !=

Opérateurs relationnels (suite)

- ② **Seuls les types numériques** peuvent être comparés avec les opérateurs d'ordre.
Les entiers, flottants et caractères peuvent être comparés avec les opérateurs d'ordre strict.
- ③ Chaque opérateur renvoie un type `boolean`.

Opérateurs booléens logiques

Les opérateurs suivants agissent uniquement sur des opérandes boolean

Op.	Résultat
~	OU exclusif logique
^=	assignation avec OU exclusif
	OU avec court circuit
==	égal à
&&	ET avec court circuit
!=	différent de
!	NON unitaire logique
? :	if-then-else ternaire

Voir exemple des || et && dans les notes de cours

Contrôle de flux

- `if/else`
- `switch/break`
- `return`
- `while/do while/for`

Instruction if-else

- 1 Forme strictement analogue à celle du C

```
if ( expression-booleenne ) expression1;  
[ else expression2; ]
```

- 2 `expression1` peut être une expression composée entourée de `{}`.
- 3 `expression-booleenne` est toute expression renvoyant un boolean.
- 4 🐼 Il est de BONNE PRATIQUE d'entourer d'accolades une **expression** même si elle n'est pas composée.
Ce qui permet, lorsque l'on veut rajouter une expression, de ne rien oublier, comme c'est le cas ci-après

Instruction if-else

```
int octetsDisponibles;  
  
if (octetsDisponibles > 0) {  
    CalculDonnees();  
    octetsDisponibles -= n;  
} else  
    attendreDautresDonnees();  
    octetsDisponibles = n;
```

où la dernière ligne devrait, d'après l'indentation, faire partie du bloc else.

Instruction break


- 1 Utilisation courante strictement analogue à celle du C : pour sortir d'un case à l'intérieur d'un switch.

Instruction switch (I)

- 1 Forme strictement analogue à celle du C

```
switch ( expression ) {  
    case valeur1 :  
        break;  
  
    case valeurN :  
        break;  
    default :  
  
}
```

expression : tout type primitif (valeuri du même type qu'expression)

- 3  Erreur répandue que d'oublier un break \Rightarrow commentaires du type // CONTINUER.
- 4 Exemple d'équivalent de wc (word count)

Instruction switch (II)

```
class WordCount {  
    static String texte =  
        "Trente rayons convergent au moyeu          " +  
        "mais c'est le vide median                 " +  
        "qui fait marcher le char.                  " + "\n"+  
  
        "On faconne l'argile pour en faire des vases, " +  
        "mais c'est du vide interne                  " +  
        "que depend leur usage.                     " + "\n"+  
  
        "Une maison est percee de portes et de fenetres, " +  
        "c'est encore le vide                         " +  
        "qui permet l'habitat.                       " + "\n"+  
  
        "L'Etre donne des possibilites,              " +  
        "c'est par le non-etre qu'on les utilise.     " + "\n"+  
  
        "Tao-to king, Lao-tseu, XI                   \n";  
  
    static int long = text.length();
```

Instruction switch (III)

```
public static void main(String args[]) {  
    boolean dansMot    = false;  
    int      nbreCars   = 0, nbreMots   = 0, nbreLignes = 0;  
  
    for (int i = 0; i < long; i++) {  
        char c = texte.charAt(i);  
        nbreCars++;  
        switch (c) {  
            case '\n' : nbreLignes++; // CONTINUER  
            case '\t' :           // CONTINUER  
            case ' ' : if (dansMot) {  
                        nbreMots++;  
                        dansMot = false;  
                    }  
                    break;  
            default  : dansMot = true;  
        }  
    }  
    System.out.println("\t" + nbreLignes + "\t" + nbreMots +  
                        "\t" + nbreCars);  
} // main()
```

Instruction return

- 1 Même usage qu'en C.
- 2  C'est une **erreur de compilation** que d'avoir du code inatteignable en java.

Instructions while/do-while/for

❶ Mêmes usages et syntaxe qu'en C. while

```
[ initialisation; ]  
while ( terminaison ) {  
    corps;  
    [ iteration; ]  
}
```

❷ do-while

```
[ initialisation; ]  
do {  
    corps;  
    [ iteration; ]  
} while ( terminaison );
```

❸ for

Instructions while/do-while/for (suite)

```
for (initialisation; terminaison; iteration)
    corps;
```

Notions de programmation orientée objet

Références :

- Object-Oriented Analysis and Design with Applications,
G. Booch,
- Data Structures,
M.A. Weiss.

POO, Objets, Classes

- Notions générales de Programmation Orientée Objet (POO)
- Notion d'objet et de classe

Conception orientée objets

- 1 Dans une **décomposition orientée objets**, on partitionne le système selon les entités fondamentales du domaine du problème.
- 2 **Sujet : objet** — **verbe : opération**.
- 3 On utilise la conception orientée objets parce qu'elle nous aide à mieux organiser la complexité inhérente aux logiciels et celle caractéristique du monde réel à modéliser.

Notion d'objet, de classe

- 1 Nous verrons qu'un **objet** a un **état**, un **comportement** et une **identité** ;
- 2 Les termes **instance** et objet sont **interchangeables**.
- 3 Une **classe** est un **squelette** pour un ensemble d'objets qui partagent une **structure commune** et un **comportement commun**.

Type ou classe ; objet

1 Classe :

- Notion de type, ou classe
- Attribut, ou champ
- Déclaration d'opération
- Description d'opération

2 Objet :

- État
- Comportement
- Identité
- Objet Java

Notion de type ou de classe

- 1 Un **type** (ou une **classe**) est constituée
 - d'**attributs** (ou champs),
 - de **déclarations d'opérations** (signatures de méthodes)
 - de **descriptions extensives d'opérations** (corps de méthodes)
- 2 Ce que l'on peut résumer par la formule suivante

$$\text{TYPE} \equiv (\text{Champ}_1, \dots, \text{Champ}_n, \\ \text{sig_meth}_1, \text{corps_meth}_1, \\ \dots, \\ \text{sig_meth}_n, \text{corps_meth}_n)$$

où sig_meth_i désigne la signature de méthode n° i et corps_meth_i désigne le corps de méthode n° i .

Notion de type ou de classe (suite)

- ③ Un type est par essence une entité **statique**, par opposition à un objet, de nature **dynamique**.
- ④ D'une certaine manière, le type est un squelette, l'objet son incarnation.

Exemple de classe

Classe (type) décrivant un cercle

```
class Cercle {  
    // champs : rayon du cercle  
    double r;  
    // Constructeur : initialisation des champs  
    Cercle(double nouvRayon) {  
        r = nouvRayon;  
    }  
    // methode de calcul d'une surface  
    double calculeSurface() {  
        return(3.1416*r*r);  
    }  
} // Fin de class Cercle
```

Notion d'état d'un objet

- 1 L'état d'un objet englobe toutes les propriétés (habituellement statiques) de l'objet plus les valeurs courantes (généralement dynamiques) de chacune de ces propriétés.
- 2 Une propriété est une caractéristique naturelle ou discriminante, un trait, une qualité ou une particularité qui contribue à rendre un objet unique.
- 3 Par exemple, dans un distributeur, un numéro de série est une propriété statique et la quantité de pièces qu'il contient est une valeur dynamique.

Notion de comportement d'un objet

- 1 Le **comportement** est la façon dont un objet agit et réagit, en termes de changement d'état et de transmission de messages.
- 2 **Message**, **opération** et **méthode** sont interchangeables.
- 3 Par exemple, dans le cas d'un distributeur de boissons, nous pouvons déclencher une action (appuyer sur un bouton) pour réaliser notre sélection.
 - Si nous n'avons pas introduit suffisamment d'argent, il ne se passera probablement rien.
 - Si nous avons mis assez d'argent, la machine l'encaissera et nous servira une boisson (modifiant ainsi son état).

Comportement d'un objet : les opérations

- 1 Une **opération** désigne un service qu'une classe offre à ses clients. En pratique, nous avons constaté qu'un client effectuait typiquement 5 sortes d'opérations sur un objet. Les 3 les plus courantes sont les suivantes :

Modificateur une opération qui altère l'état d'un objet

Sélecteur une opération qui accède à l'état d'un objet, mais qui n'altère pas celui-ci.

Itérateur une opération qui permet d'accéder à toutes les parties d'un objet dans un ordre bien défini.

Comportement d'un objet : les opérations (II)

Deux autres types d'opération sont courants :

Constructeur une opération qui crée un objet et/ou initialise son état.

Destructeur Une opération qui libère l'état d'un objet et/ou détruit l'objet lui-même.

Notion d'identité d'un objet

- 1 L'identité est cette propriété d'un objet qui le distingue de tous les autres objets.
- 2 Deux objets peuvent être déclarés égaux en 2 sens différents.
- 3 Ils peuvent être égaux au sens de leur références (les pointeurs internes qui référencent les données de l'objet en mémoire) ou
- 4 au sens de leur contenu (égalité de leur état), bien qu'ils soient situés à des emplacements mémoire différents.

Nature d'un objet en Java

- ① Un objet Java peut être décrit par la formule suivante :

$$\text{OBJET} \equiv (\text{état}, \text{op}_1, \dots, \text{op}_n, \text{ref})$$

où

etat	ensemble des variables d'instance
op_i	(pointeur sur) la méthode d'instance n° i
ref	(pointeur sur) un emplacement mémoire contenant l'état et des références internes vers les opérations (pointeurs sur les méthodes)

Nature d'un objet en Java (II)

- 1 Exemple d'objet, de type tasse à café.
- 2 Des **attributs** d'une tasse à café pourront être :
 - sa couleur,
 - la quantité de café qu'elle contient,
 - sa position dans le café (la brasserie ou le bar)
- 3 **Tasse à café** est un type et "la tasse à café rouge qui contient actuellement 38 millilitres de café et qui se trouve sur la dernière table du fond" est un objet.
- 4 **Rouge, 38 millilitres et sur la dernière table du fond** constituent l'état de cet objet.
- 5 Un type, ou une classe sert de modèle à partir duquel on peut **instancier** (créer) des objets contenant des variables d'instance et des méthodes définies dans la classe.

Relations

- Séparation interface/implantation
- Relations d'héritage

Séparation de l'interface et de l'implantation

- 1 Une idée clé est de **séparer l'interface externe d'un objet de son implantation**.
- 2 L'**interface** d'un objet est constituée des messages qu'il peut accepter d'autres objets. Autrement dit, c'est la déclaration des opérations associées à l'objet.
- 3 L'**implantation d'un objet** se traduit par la valeur de ses attributs et son comportement en réponse aux messages reçus.

Séparation interface/implantation (II)

- 1 Dans un monde orienté-objets, un objet expose son interface aux autres objets, mais garde son implantation privée.
- 2 L'implantation doit donc être séparée de l'interface.
- 3 De l'extérieur, **le seul moyen pour interagir avec un objet est de lui envoyer un message** (d'exécuter l'une de ses opérations).
- 4 La séparation de l'interface et de l'implantation permet aux objets d'avoir la **responsabilité** de gérer leur propre état.

Séparation interface/implantation (III)

- 1 Les autres objets ne peuvent manipuler cet état directement et doivent passer par des messages (ou opérations).
- 2 L'objet qui reçoit un message peut décider de changer ou non son état. Par contre, il ne contrôle pas à quel instant il va recevoir des messages.
- 3 Un aspect fondamental de la programmation orientée objet est que **chaque objet d'une classe particulière peut recevoir les mêmes messages.**
- 4 L'interface externe d'un objet ne dépend donc que de sa classe.

Relations entre classes

- 1 Il existe trois types fondamentaux de relations entre classes :
 - La **généralisation/spécialisation**, désignant une relation “est un”.
Par exemple, **une rose est une sorte de fleur** : une rose est une sous-classe plus spécialisée de la classe plus générale de fleur.
 - L'**ensemble/composant**, dénotant une relation “partie de”.
Par exemple, **un pétale est une partie d'une fleur**.
 - L'**association**, traduisant une dépendance sémantique entre des classes qui ne sont pas reliées autrement. Par exemple, **une fleur et une bougie** peuvent **ensemble** servir de décoration sur une table.

Relations d'héritage entre classes

- 1 L'**héritage** est une relation entre les classes dont l'une partage la **structure** ou le **comportement** défini dans une (**héritage simple**) ou plusieurs (**héritage multiple**) autres classes.
- 2 On nomme **super-classe** la classe de laquelle une autre classe hérite.
- 3 On appelle une classe qui hérite d'une ou plusieurs classes une **sous-classe**.

Relations d'héritage (II)

- 1 Par exemple, prenons une classe `Surface2DSymetrique`.
Considérons les classes `Pave2D` et `Disque` héritant de `Surface2DSymetrique`.
- 2 L'héritage définit donc une hiérarchie de la forme est un entre classes. C'est le test de vérité de l'héritage.
- 3 Dans une relation d'héritage, les sous-classes héritent de la structure de leur super-classe.

Relations d'héritage (III)

- 1 Par exemple, la classe `Surface2DSymetrique` peut avoir comme champs :

l'abscisse de son centre de symétrie	x
l'ordonnée de son centre de symétrie	y
sa taille	size
sa couleur	color

Et les classes `Pave2D` et `Disque` hériteront de ces champs. Une sous-classe peut définir d'autres champs qui viennent s'ajouter à ceux hérités des super-classes.

- 2 De plus, toujours dans une relation d'héritage, les sous-classes héritent du comportement de leur super-classe.

Relations d'héritage (IV)

Par exemple, la classe `Surface2DSymetrique` peut avoir comme opérations :

<code>getSize()</code>	pour obtenir la taille de la surface
<code>getX()</code>	pour obtenir l'abscisse du centre de gravité
<code>getY()</code>	pour obtenir l'ordonnée du centre de gravité
<code>setXY()</code>	pour fixer la position de la surface
<code>setColor()</code>	pour fixer la couleur de la surface

Relations d'héritage (V)

- 1 Et les classes `Pave2D` et `Disque` hériteront de ces champs.
- 2 Une sous-classe peut définir d'autres opérations qui viennent s'ajouter à celles héritées des super-classes.
- 3 Une sous-classe peut redéfinir tout ou partie des opérations héritées des super-classes.

Relations d'héritage (VI)

- 1 Le **polymorphisme** est un mécanisme par lequel un nom peut désigner **des objets de nombreuses classes** différentes, tant qu'elles sont reliées par une super-classe commune.
- 2 Tout objet désigné par ce nom est alors capable de répondre de différentes manières à un ensemble commun d'opérations.

Bases orientées objet de Java

Références :

- The Java Language Specification,
J. Gosling, B. Joy et G. Steele
- Java in a Nutshell,
D. Flanagan

Classes et objets Java

- Classe et objet Java
- méthode `main()`
- Compilation, exécution
- Référence, `new`
- Variables & méthode d'instance
- Constructeurs, `this`
- Héritage

Constitution d'une classe

- ❶ Rappel des notions de classe et d'objet, en deux mots
 - **Classe** : squelette ; **structure de données et code des méthodes** ; statique, sur disque
 - **Objet** : incarnation ; **état, comportement, identité** ; dynamique, en mémoire
- ❷ Une classe définit :
 - Les **Structures de données** associées aux objets
 - variables désignant ces données : **champs**.
 - Les **services** rendus par les objets : **méthodes**
- ❸ Une **Classe est déclarée** par le mot clé `class`.

Champs et méthodes

- 1 Champ \longleftrightarrow déclaration de variable
- 2 Nom de variable suit la déclaration du type :

```
class Point {  
    int x;  
    int y;  
    ...  
}
```

- 3 Une méthode est constituée de :
 - Un nom
 - Des paramètres, en nombre fixe
 - D'un type de retour :
 - soit void,

Champs et méthodes (suite)

- soit un type primitif
- soit une référence vers un objet.
- Du **corps** (instructions java) de la méthode.

④ Exemple de classe décrivant un cercle

```
class Cercle {  
    // champs : rayon du cercle  
    double r;  
    // methode de calcul d'une surface  
    double calculeSurface() {  
        return(3.1416*r*r);  
    }  
} // Fin de class Cercle
```


Déclaration de classe

- 1 Un fichier source java doit **porter le même nom** que celui de la classe publique qui y est définie.

- 2 Syntaxe générique

```
class NomClasse {  
    type variableInstance1;  
    type variableInstanceN;  
    type nomMethode1(liste-parametres) {  
        corps-methode;  
    }  
    type nomMethodeN(liste-parametres) {  
        corps-methode;  
    }  
}
```

- 3 Exemple

Déclaration de classe (suite)

```
class Chat {  
    String nom;           // nom du fauve  
    int    age;           // en années  
    float  tauxRonronnement; // entre 0 et 1  
  
    void vieillir() {  
        age += 1;  
    }  
  
    int retournerAge() {  
        return(age);  
    }  
}
```

Déclaration de classe (suite)

- ④  Les **déclaration et implantation** d'une méthode sont **dans le même fichier**.

Point d'entrée d'un programme (main())

- 1 Un programme Java est constitué d'une ou de plusieurs classes.
- 2 Au moins une classe contenant le `main()`, méthode **statique et publique**.
- 3 Est le point d'entrée de l'exécution du programme.

```
// Fichier Bonjour.java
public class Bonjour {
    public static void main(String args[]) {
        System.out.println("Bonjour ! ") ;
    }
}
```

Point d'entrée d'un programme (main()) (suite)

- ④ Cette classe définit une classe Bonjour qui ne possède qu'une seule méthode.
- ⑤ La méthode main() doit être déclarée static et public pour qu'elle puisse être invoquée par l'interpréteur Java.
- ⑥ L'argument args est un tableau de String qui correspond aux arguments de la ligne de commande lors du lancement du programme.
- ⑦ args[0] est le 1^{er} argument, args[1] est le 2^{ième} argument, ...

Compilation

- 1 Avant de pouvoir exécuter ce programme, il faut tout d'abord le compiler, par exemple avec la commande `javac` .
`javac Bonjour.java`
- 2 La commande `javac` traduit le code source en code intermédiaire (`p-code`) java.
- 3 Ce code (une forme d'assembleur générique) est évidemment **indépendant de la plate forme** sur laquelle il a été compilé.

Exécution

- 1 **Autant de fichiers** que **de classes** qui ont été définies dans le fichier source sont produits.
- 2 Les fichiers **compilés** ont l'extension **.class**.
- 3 Enfin, pour exécuter ce programme, il faut utiliser l'interpréteur de code Java
- 4 et lui fournir le **nom de la classe** contenant le **main(...)** (sans l'extension).
java Bonjour

Référence à un objet

- 1 En Java, on ne peut accéder aux objets **qu'à travers une référence** vers celui-ci.
- 2 Déclaration d'une variable `p` avec pour type un nom de classe :

`Point p;`

`p` : **référence à un objet** de la classe `Point`.

- 3 Lorsque l'on déclare une classe comme type d'une variable, cette dernière a, par défaut, la valeur `null`.
- 4 **`null`** est une référence à un `Object` (mère de toutes les classes Java), qui n'a pas de valeur (distinct de 0); par ex. dans

Référence à un objet (suite)

Point p;
p a la valeur null.

- 5 En fait, référence à un objet : pointeur.
- 6 Mais l'**arithmétique** sur les **pointeurs est impossible** en java.
- 7 **Seule chose permise : changer la valeur** de la référence pour pouvoir "faire référence" à un autre objet.
- 8 Plus précisément, une référence pointe sur une structure où se trouve
 - des informations sur le type
 - l'adresse réelle des données (instance d'objet)

Opérateur new

- ① **new** : **création** d'une instance **d'objet** d'une classe ; retourne une référence à cette instance d'objet.

```
Point p  = new Point();    // ligne 1
Point p2 = p;              // ligne 2
p = null;                  // ligne 3
```

- Ligne 2 : tout changement à p2 affecte l'objet référencé par p.
p2 = p : aucune copie de l'objet ou allocation mémoire.
 - Ligne 3 : décrochage de p de l'objet originel. p2 permet toujours d'y accéder.
- ② Objet qui n'est plus référencé \Rightarrow le **ramasse-miettes** (**garbage collector**) récupère automatiquement la mémoire associée.

Instance d'objet

- 1 **Instance** : copie individuelle de prototype de la classe, avec ses propres données : **variables d'instance**.
- 2 Une fois la classe déclarée, pour pouvoir utiliser un objet de cette classe, il faut définir une **instance (d'objet)** de cette classe.
- 3 Or les objets ne sont accessibles qu'à travers des références .
- 4 Donc une définition qui spécifie un objet comme "une variable ayant le type de la classe choisie " ne fait que définir une référence vers un éventuel objet de cette classe.

Date d;

Instance d'objet (suite)

- 5 La variable `d` représente une référence vers un objet de type `Date`.
- 6 En interne, cela réserve de la place pour le pointeur sous-jacent à la référence `d`.
- 7 Mais cela **ne réserve pas de place mémoire pour une variable** de type `Date`.
- 8 Si l'on veut une instance d'objet effective, il faut la créer explicitement avec le mot clé `new` et le constructeur de la classe `Date`.

```
Date d;  
d = new Date();
```

Méthode d'instance

- 1 On peut voir une méthode comme un message envoyé à une instance d'objet.
- 2 Pour afficher la date contenue dans l'objet d, on lui envoie le message imprimer :
`d.imprimer() ;`
- 3 De telles méthodes sont appelées **méthodes d'instance**.

Variables d'instance

- ❶ Les **Variables d'instance** sont déclarées en dehors de toute méthode

```
class Point {  
    int x, y;  
}
```

Op. point (.) – Déclaration de méthode

- ❶ Opérateur . : accéder à des variables d'instance et à des méthodes d'un(e instance d'un) objet.
- ❷ Ex. de déclaration de méthode

```
class Point {  
    int x, y;  
    void init(int a, int b) {  
        x = a;  
        y = b;  
    }  
}
```

Op. point (.) – Déclaration de méthode (suite)

- 3 En C, méthode sans paramètre : `nommethode(void)`. **illégal en java.**
- 4 Les **objets sont passés par référence** (références d'instances à un objet passés par valeur).
- 5 Les **types primitifs sont passés par valeur.**
- 6 Les méthodes java sont donc similaires aux fonctions virtuelles du C++.

Instruction `this`

- 1 `this` : référence à l'instance d'objet courante.
- 2 🐼 Il est permis à une variable locale de **porter le même nom** qu'une variable d'instance ...
- 3 Exemple d'utilisation de `this` évitant cela

```
void init(int x, int y) {  
    this.x = x;  
    this.y = x;  
}
```

Constructeurs

- ❶ Même nom que celui de la classe. Pas de type de retour (pas même void).
- ❷ Classe décrivant un cercle

```
class Cercle {  
    double r; // champs : rayon du cercle  
    // Constructeur : initialisation des champs  
    Cercle(double nouvRayon) {  
        r = nouvRayon;  
    }  
    double calculeSurface() {  
        return(3.1416*r*r); // methode de calcul  
    }  
} // Fin de class Cercle
```

Constructeurs (suite)

3 Exemple animalier

```
class Chat {  
    String nom;           // nom du fauve  
    int    age;           // en annees  
    float  tauxRonronnement; // entre 0 et 1  
  
    public Chat(String sonNom,  
                 int    sonAge,  
                 float  sonTauxRonron) {  
        nom          = sonNom;  
        age          = sonAge;  
        tauxRonronnement = sonTauxRonron;  
    }  
}
```

Constructeurs (suite)

- ❶ this peut-être également un appel à un **constructeur**

```
class Point {  
    int x, y;  
    // constructeur exhaustif  
    Point(int x, int y) {  
        this.x = x;    // var d'instance Point.x  
        this.y = y;  
    }  
    // Appel du constructeur exhaustif  
    Point() {  
        this(-1, -1); // Point(int x, int y)  
    }  
}
```

Exemple de constructeurs

1 Exemple animalier

```
class Chat {  
    String  nom;           // nom du fauve  
    int     age;           // en anneés  
    Color[] couleurPelage; // ses différentes couleurs  
    float   tauxRonronnement; // entre 0 et 1  
  
    public Chat(String sonNom,  
                  int    sonAge,  
                  float  sonTauxRonron,  
                  Color[] sesCouleurs) {  
        nom           = sonNom;  
        age           = sonAge;  
        tauxRonronnement = sonTauxRonron;  
        couleurPelage  = sesCouleurs;  
    }  
  
    public Chat() {  
        this(new String("minou"), 1, 0.5,  
              {Color.black, Color.white});  
    }  
}
```

Exemple de constructeurs (suite)

- 1 Technique de **réutilisation** : créer un constructeur exhaustif (doté de tous les paramètres),
- 2 puis créer d'autres constructeurs appelant systématiquement le constructeur exhaustif.

Héritage

- Héritage de classe

Héritage

- ① Les descendants par héritage sont nommés des **sous classes**.
- ② Le parent direct est une **super classe**.
- ③ Une sous classe est une version **spécialisée** d'une classe qui **hérite** de toutes les variables d'instance et méthodes.

Héritage (suite)

① Mot-clé **extends**

```
class Point3D extends Point {  
    int z;  
    Point3D(int x, int y, int z) {  
        this.x = x;  
        this.y = y;  
        this.z = z;  
    }  
    Point3D() {  
        Point3D(-1, -1, -1);  
    }  
}
```

Héritage (suite)

❶ Syntaxe générique

```
class NomClasse {  
    type variableInstance1;  
    type variableInstanceN;  
    type nomMethode1(liste-parametres) {  
        corps-methode;  
    }  
    type nomMethodeN(liste-parametres) {  
        corps-methode;  
    }  
}
```

Héritage (suite)

- ❶ Pas d'héritage multiple, pour des raisons de performances et de complexité (en maintenance). À la place, notion d'interface.
- ❷ Il existe une classe au sommet de la hiérarchie, `Object`. Sans mot-clé `extends`, le compilateur met automatiquement `extends Object`.
- ❸ De la même manière que l'on peut assigner à une variable `int` un `byte`, on peut déclarer une variable de type `Object` et y stocker une référence à une instance de toute sous classe d'`Object`.

Surcharge, redéfinition

- `super`
- Sous-typage, transtypage, appartenance de type
- Surcharge
- Redéfinition
- Répartition de méthode dynamique
- `final`, `static`, `abstract`

Instruction super

- 1 **super** réfère aux variables d'instance et aux constructeurs de la **super classe**.

```
class Point3D extends Point {  
    int z;  
    Point3D(int x, int y, int z) {  
        super(x, y); // Appel de Point(x,y).  
        this.z = z;  
    }  
}
```

- 2 Cet appel au constructeur de la classe mère doit être la 1^{ère} ligne du constructeur.

Instruction super (suite)

- 1 `super` peut également se référer aux méthodes de la super classe :
`super.distance(x, y)` appelle la méthode `distance()` de la super classe de l'instance `this`.
- 2 Exemple animalier (voir l'**excellent** ouvrage "**le mystère des chats peintres**" de Heather Busch et Burton Silver, [http ://www.monpa.com/wcp/index.html](http://www.monpa.com/wcp/index.html))

Instruction super (suite)

```
class ChatPeintre extends Chat {  
    // Variables d'instances  
    String    style;  
    int       coteMoyenne; // cote moyenne d'une oeuvre  
  
    // Constructeurs  
    public ChatArtiste(String sonNom, int sonAge,  
                        float sonTauxRonron,  
                        Color[] sonPelage,  
                        String sonStyle, int saCote) {  
        super(sonNom, sonAge, sonTauxRonron, sonPelage);  
        style      = sonStyle;  
        coteMoyenne = saCote;  
    }  
  
    // Methodes  
    public peindre() {  
        ...  
    }  
}
```

Un artiste en pleine action



source : [http ://www.monpa.com/wcp/index.html](http://www.monpa.com/wcp/index.html)

Sous-typage, transtypage, instanceof

- 1 Le typage d'une variable lui permet de référencer tout sous type (classe parente);
- 2 la méthode miauler() est définie dans Chat. La méthode peindre() n'est définie que dans ChatPeintre.

```
Chat gouttiere = new Chat("zephir", 1, 0.9);
ChatPeintre moustacheDeDali =
    new ChatPeintre("dali",    // nom de l'artiste
                    2,        // son age
                    0.1,      // son taux rr
                    {Color.white, Color.black},
                    "aLaDali", // son style
                    20000);    // sa cote moyenne

moustacheDeDali.peindre(); // valide
gouttiere.peindre();       // illegal
```

Sous-typage, transtypage, instanceof (suite)

- 1 instanceof permet de savoir si un objet est d'un type donné ou non.

```
// true
System.out.print(gouttiere instanceof Chat);
// true
System.out.print(moustacheDeDali instanceof Chat);
// false
System.out.print(gouttiere instanceof ChatPeintre);
moustacheDeDali = null;
// false
System.out.print(moustacheDeDali instanceof ChatPeintre);
```

Sous-typage, transtypage, instanceof (suite)

- 1 Transtypage (ou “cast” en anglais) permet de changer le type, lorsque cela est permis.

```
Chat ch = new Chat("zephir", 1, 0.9);  
ChatPeintre chP;  
chP = ch;                               // Erreur de compilation  
  
if (ch instanceof ChatPeintre) // Bonnes manieres  
    chP = (ChatPeintre)ch;      // transtypage
```

Surcharge de méthode

- ➊ Plusieurs méthodes peuvent porter le même nom : **surcharge de méthode**.
- ➋ Différentiation sur la **signature de type** : le **nombre et le type des paramètres**.
- ➌ Deux méthodes d'une même classe de mêmes nom et signature de type est illégal.
- ➍ Exemple de surcharge

Surcharge de méthode (suite)

```
class Point {
    int x, y;
    Point(int x, int y) {
        this.x = x;          this.y = y;
    }
    double distance(int x, int y) {
        int dx = this.x - x; int dy = this.y - y;
        return Math.sqrt(dx*dx + dy*dy);
    }
    double distance(Point p) {
        return distance(p.x, p.y);
    }
}

class PointDist {
    public static void main(String args[]) {
        Point p1 = new Point(0, 0);
        Point p2 = new Point(30, 40);
        System.out.println("p1.distance(p2) = " +
```

Surcharge de méthode (suite)

```
                p1.distance(p2));  
System.out.println("p1.distance(60, 80) = " +  
                p1.distance(60, 80));
```

```
}
```

5 Exemple animalier

```
class Chat {
```

```
    ....
```

```
void vieillir() {
```

```
    age += 1;
```

```
}
```

```
void vieillir(int n) { // Surcharge de methode
```

```
    age += n;
```

```
}
```

Redéfinition de méthode

- 1 Distance en perspective dans Point3D (distance 2D entre x/z et y/z) \Rightarrow **redéfinir** distance(x, y) de Point2D.
- 2 Ex. de surcharge de distance 3D et de redéfinition de distance 2D

```
class Point {
    int x, y;
    Point(int x, int y) {
        this.x = x;
        this.y = y;
    }
    double distance(int x, int y) {
        int dx = this.x - x;
        int dy = this.y - y;
        return Math.sqrt(dx*dx + dy*dy);
    }
    double distance(Point p) {                // Surcharge
        return distance(p.x, p.y);
    }
} // class Point
```

Redéfinition de méthode (suite)

```
class Point3D extends Point {
    int z;
    Point3D(int x, int y, int z) {
        super(x, y);        // Appel de Point(x,y)
        this.z = z;
    }
    double distance (int x, int y, int z) {
        int dx = this.x - x;  int dy = this.y - y;
        int dz = this.z - z;
        return Math.sqrt(dx*dx + dy*dy + dz*dz);
    }
    double distance(Point3D other) { // Surcharge
        return distance(other.x, other.y, other.z);
    }
    double distance(int x, int y) { // Redéfinition
        double dx = (this.x / z) - x;
        double dy = (this.y / z) - y;
        return Math.sqrt(dx*dx + dy*dy);
    }
}
```


Redéfinition de méthode (suite)

```
class Point3DDist {  
    public static void main(String args[]) {  
        Point3D p1 = new Point3D(30, 40, 10);  
        Point3D p2 = new Point3D(0, 0, 0);  
        Point p = new Point(4, 6);  
        System.out.println("p1.distance(p2) = " +  
                           p1.distance(p2));  
        System.out.println("p1.distance(4, 6) = " +  
                           p1.distance(4, 6));  
        System.out.println("p1.distance(p) = " +  
                           p1.distance(p));  
    }  
}
```

L'affichage du programme est le suivant. Pourquoi ?

```
Prompt > java Point3DDist  
p1.distance(p2) = 50.9902  
p1.distance(4,6) = 2.23607  
p1.distance(p) = 2.23607
```

Redéfinition de méthode (suite)

- Appel de distance sur un Point3D (p1) :
exécution de `distance(Point p)` héritée de la super classe
(méthode non redéfinie).
Mais ensuite **appel de** `distance(int x, int y)` **de**
Point3D, pas de Point.
- Sélection de méthode **selon le type de l'instance** et non selon
la classe dans laquelle la méthode courante s'exécute :
répartition de méthode dynamique.

Répartition de méthode dynamique

```
class Parent {  
    void appel() {  
        System.out.println("Dans Parent.appel()");  
    }  
}  
class Enfant extends Parent {  
    void appel() {  
        System.out.println("Dans Enfant.appel()");  
    }  
}  
class Repartition {  
    public static void main(String args[]) {  
        Parent moi = new Enfant();  
        moi.appel();  
    }  
}
```

Répartition de méthode dynamique (suite)

- ① Lors de `moi.appel()`
 - Le compilateur vérifie que Parent a une méthode `appel()`,
 - l'environnement d'exécution remarque que la référence `moi` est en fait vers une instance d'Enfant \Rightarrow appel de `Enfant.appel()`
- ② Il s'agit d'une forme de **polymorphisme** à l'exécution.
- ③ ➦ Cela permet à des bibliothèques existantes d'appeler des méthodes sur des instances de **nouvelles** classes **sans recompilation**.


Instruction final

- ① Variable d'instance ou **méthode non redéfinissable** : `final`.
- ② Pour des variables, convention de majuscules
`final int FILE_QUIT = 1 ;`
- ③ Les sous classes ne peuvent redéfinir les méthodes `final`.
- ④ Petites méthodes `final` peuvent être optimisées (appels "en ligne" par recopie du code).
- ⑤ 🐼 `final` pour les variables est similaire au `const` du C++. Il n'y a pas d'équivalent de `final` pour les méthodes en C++.

Méthode finalize()

- 1 Instance d'objet ayant une ressource non java (descripteur de fichier) : moyen de la libérer.
- 2 Ajout d'une **méthode** finalize() à la classe. **Appelée à chaque libération** d'une instance d'objet de cette classe.

Instruction static

- **méthode** `static` : utilisée **en dehors de tout contexte d'instance**.
- Méthode `static` ne peut appeler directement que des méthodes `static`. Ne peut utiliser `this` ou `super`. Ne peut utiliser une variable d'instance.
-  **Variables** `static` : **visibles de toute autre portion de code**. Quasiment des variables globales. À utiliser avec parcimonie ...
- **Bloc** `static` : **exécuté une seule fois**, au premier chargement de la classe.

Instruction static (suite)

- Exemple

```
class Statique {  
    static int a = 3;  
    static int b;  
    static void methode(int x) {  
        System.out.println("x = " + x +  
                           ", a = " + a +  
                           ", b = " + b);  
    }  
    static {  
        System.out.print("Initialisation" +  
                          " du bloc statique");  
        b = a * 4;  
    }  
}
```


Instruction static (suite)

```
        public static void main(String args[]) {  
            methode(42);  
        }  
    }
```

- L'affichage est

```
Prompt > java Statique  
Initialisation du bloc static  
x = 42, a = 3, b = 12
```

- Initialisation de a et b.
- Exécution du bloc static.
- Appel de main().

Instruction static (suite)

- Appel d'une variable ou méthode static par le nom de la classe

```
class ClasseStatique {  
    static int a = 42;  
    static int b = 99;  
    static void appel() {  
        System.out.println("a = " + a);  
    }  
}  
  
class StatiqueParNom {  
    public static void main(String args[]) {  
        ClasseStatique.callme();  
        System.out.println("b = " +  
                            ClasseStatique.b);  
    }  
}
```

Instruction static (suite)

```
}  
}
```

- Exemple animalier

```
class Chat {  
    String    nom;           // nom  
    int       age;           // annees  
    Color[]   couleurPelage; // couleurs  
    float     tauxRonronnement; // de 0 a 1  
    static int ageSevrage = 1; // statique  
  
    boolean estAdoptable() {  
        if (age > ageSevrage) {  
            return true;  
        } else {
```

Instruction static (suite)

```
        return false;
    }
}
```

Instruction abstract

- 1 **Partie spécification, partie implantation** : classes abstraites .
- 2 Certaines méthodes, sans corps, **doivent être redéfinies** par les sous classes : méthodes **abstraites**. C'est la **responsabilité de sous classe**.
- 3 Toute classe contenant des méthodes abstraites (mot clé `abstract`) doit être déclarée abstraite.
- 4 Les classes abstraites ne peuvent être instanciées par `new`. Pas de constructeurs ou de méthodes `static`.
- 5 Une **sous classe** d'une classe statique **soit implante** toutes les méthodes abstraites, **soit est elle-même abstraite**.

Instruction abstract (suite)

Exemple

```
abstract class ParentAbstrait {
    abstract void appel();
    void moiaussi() {
        System.out.print("Dans ParentAbstrait.moiaussi()");
    }
}

class EnfantConcret extends ParentAbstrait {
    void appel() {
        System.out.print("Dans EnfantConcret.moiaussi()");
    }
}

class AbstractionMain {
    public static void main(String args[]) {
        ParentAbstrait etre = new EnfantConcret();
        etre.appele();
        etre.moiaussi();
    }
}
```

Paquetages et interfaces

- Paquetages & import
- Protection d'accès & modificateurs
- Interfaces

Paquetages

- 1 À la fois un **mécanisme de nommage** et un **mécanisme de restriction de visibilité**.
- 2 Forme générale d'un source java
 - une unique declaration de paquetage (optionnel)
 - declarations d'importations (optionnel)
 - une unique declaration de classe publique
 - declarations de classes privees (optionnel)
- 3 Pas de déclaration de paquetage : les classes déclarées font partie du paquetage par défaut, sans nom.
- 4 Une classe déclarée dans le paquetage `monPaquetage` \Rightarrow le source **doit être dans le répertoire** `monPaquetage` (il y a distinction minuscule-majuscule).

Paquetages (suite)

- 1 Syntaxe générique :
`package pkg1[.pkg2[.pkg3]] ;`
- 2 Par exemple
`package java.awt.image ;`
doit être stocké dans
 - `java/awt/image` (sous UNIX),
 - `java\awt\image` (sous Windows)
 - ou `java :awt :image` (sous Macintosh).
- 3 La racine de **toute hiérarchie de paquetage** est une entrée de la variable d'environnement CLASSPATH.

Paquetages (suite)

✚ Ayant une classe ClasseTest dans un paquetage test, il faut

- soit **se mettre dans le répertoire père** de test et lancer
`java test.ClasseTest`,
- soit **ajouter le répertoire test à la variable CLASSPATH** :

`CLASSPATH=.;c:\code\test;c:\java\classes`

- soit **lancer** :

`java -dclasspath=.;c:\code\test;c:\java\classes ClasseTest`

Instruction import

- 1 Entrer les noms complets de classes et méthodes fort long
⇒ Tout ou partie d'un paquetage est amené en visibilité directe, avec import.

- 2 Syntaxe générique

```
import pkg1[.pkg2].(nomclasse|*);
```

- 3 Exemple

```
import java.util.Date;  
import java.io.*;
```

- 4 Chargement de gros paquetages ⇒ perte de performance en compilation. Pas d'effet à l'exécution.
- 5 Toutes les classes livrées dans la distribution java sont dans le **paquetage** java.
- 6 Les **Classes de base** du langage se trouvent dans java.lang.

Instruction import (suite)

- 7 Il y a une importation implicite de `import java.lang.*`
- 8 Deux classes de même nom dans 2 paquetages différents importés avec `*` : le compilateur ne dit rien jusqu'à l'utilisation d'une des classes, où c'est une erreur de compilation.
- 9 Utilisation de noms complets. Au lieu de


```
import java.util.*;  
class MaDate extends Date { ... }  
on peut utiliser class MaDate extends java.util.Date  
...
```

Protections d'accès

4 catégories de visibilité :

- Sous classe dans le même paquetage.
 - Non sous classe dans le même paquetage.
 - Sous classe dans des paquetages différents.
 - Classes ni dans le même paquetage, ni sous classes.
- 1 Déclaré `public` : peut être **vu de partout**.
 - 2 Déclaré `private` : **ne peut** être vu en **dehors d'une classe**.
 - 3 Pas de modificateur : visible des **sous classes et des autres classes du même paquetage**. **Situation par défaut**.
 - 4 Déclaré `protected` : peut être vu hors du paquetage, mais **seulement des sous classes**.

Protections d'accès (suite)

- 5 Déclaré `private` `protected` : ne peut être vu que des sous classes.
- 6  `protected` pas la même signification qu'en C++. Plutôt similaire au `friend` du C++. Le `protected` du C++ est émulé par `private` `protected` en java.

Exemple animalier

```
class Chat {  
    // Les différents champs sont protected (et non private),  
    // de façon à être visibles des sous-classes  
    protected String      nom;           // nom du fauve  
    protected int         age;           // en années  
    protected Color[]     couleurPelage; // ses couleurs  
    protected float       tauxRonronnement; // entre 0 et 1  
    protected static int  ageSevrage = 1; // Champ statique
```

Protections d'accès (suite)

```
// Les constructeurs doivent etre vus de partout
public Chat(String sonNom, int sonAge, float sonTauxRonron,
             Color[] sesCouleurs) {
    nom          = sonNom;
    age          = sonAge;
    tauxRonronnement = sonTauxRonron;
    couleurPelage  = sesCouleurs;
}

public Chat() {
    this(new String("minou"), 1, 0.5,
         {Color.black, Color.white});
}
```

Protections d'accès (suite)

```
// Accesseurs
public int retournerAge() {
    return(age);    }
public String retournerNom() {
    return(nom);    }
public Color[] retournerCouleurPelage() {
    return(couleurPelage);    }
public float retournerTauxRonron() {
    return(tauxRonronnement);    }

// Autres methodes
public void vieillir() {
    age += 1;
}
public void vieillir(int n) {
    age += n;
}
public boolean estAdoptable() {
```


Protections d'accès (suite)

```
        if (age > ageSevrage) {
            return true;
        } else {
            return false;
        }
    }

// Methode privée
private void emettreSon(String adire) {
    // Emulation ultra pauvre du son
    System.out.println(" " + adire);
}

// Utilisation de la methode privée
public void miauler(int nbMiaulements) {
    for(int i = 0; i < nbMiaulements; i++) {
        emettreSon("Miaou !");
    }
}
```

Protections d'accès (suite)

```
}  
}
```

Interfaces

- ❶ **Interfaces** : comme des classes, mais **sans variable d'instance** et des **méthodes** déclarées **sans corps**.
- ❷ Une classe peut implanter un nombre quelconque d'interfaces.
- ❸ Pour cela, la classe **doit fournir l'implantation de toutes les méthodes de l'interface**.
- ❹ La signature de type doit être respectée.
- ❺ Les interfaces vivent dans une hiérarchie différente de celles des classes
- ❻ \Rightarrow **deux classes sans aucun lien hiérarchique peuvent implanter la même interface**.
- ❼ Les interfaces sont aussi utiles que l'héritage multiple, mais donnent du **code plus facile à maintenir**.

Interfaces (suite)

- 8 En effet, ne repose pas sur des données, juste sur des méthodes.
- 9 Syntaxe générique

```
interface nom {  
    type-retour nom-methode1(liste-parametres);  
    type nomvariable-finale = valeur;  
}
```
- 10 Toutes les méthodes implantant une interface doivent être déclarées public.
- 11 Variables déclarées à l'intérieur d'une interface implicitement final.

Exemple d'interface

- 1 Syntaxe générique d'implantation d'interface

```
class nomclasse [extends superclasse]
                 [implements interface0
                  [,interface1...]] {
    corps-de-classe
}
```

- 2 Les crochets désignent des mots optionnels

- 3 Exemple

```
interface Callback {
    void callback(int parametre) {
}

class Client implements Callback {
    void callback(int p) {
        System.out.println("Callback de " + p);
    }
}
```

Interface & résolution dynamique de méthode

- On peut déclarer des variables références à des objets utilisant une **interface comme type** au lieu d'une classe.
- Toute instance d'une classe implantant cette interface peut être stockée dans cette variable.
- Si l'on veut appeler une méthode *via* une telle variable, l'implantation correcte sera appelée selon l'instance courante.
- Les **classes** peuvent donc être **créées après le code qui les appelle**.
- Cette technique de **résolution dynamique de méthode** est coûteuse en temps.

Interface & résolution dynamique de méthode (suite)

⑥ Aspect d'encapsulation

```
class TestInterface {  
    public static void main(String args[]) {  
        Callback c = new Client();  
        c.callback(12);  
    }  
}
```

c ne peut être utilisé **que pour accéder à la méthode callback()** et non à un autre aspect de Client.

Exceptions

Références :

- The Java Language Specification,
J. Gosling, B. Joy et G. Steele

Fonctionnement général du système d'exceptions

- Génération
- try/catch
- throw, throws
- finally

Génération et gestion d'exceptions

- ① **Exception** : condition anormale survenant lors de l'exécution.
- ② Lorsqu'une exception survient :
 - un objet représentant cette exception est créé ;
 - cet objet est **jeté (thrown)** dans la méthode ayant provoqué l'erreur.
- ③ Cette méthode peut choisir :
 - de gérer l'exception elle-même,
 - de la passer sans la gérer.

De toutes façons l'exception est **captée (caught)** et traitée, en dernier recours par l'environnement d'exécution Java.

Génération et gestion d'exceptions (suite)

- ④ Les exceptions peuvent être générées
 - par l'environnement d'exécution Java,
 - manuellement par du code.
- ⑤ Les exceptions jetées (ou levées) par l'environnement d'exécution résultent de violations des règles du langage ou des contraintes de cet environnement d'exécution.

Les 5 mots clés

- ❶ Il y a 5 mots clés d'instructions dédiées à la gestion des exceptions :
`try`, `catch`, `throw`, `throws` et `finally`.
- ❷ Des instructions où l'on veut surveiller la levée d'une exception sont mises dans un bloc précédé de l'instruction `try`.
- ❸ Le code peut capter cette exception en utilisant `catch` et la gérer.

Les 5 mots clés (suite)

- ④ Les exceptions générées par le système sont automatiquement jetées par l'environnement d'exécution Java. Pour jeter une exception manuellement, utiliser `throw`.
- ⑤ Toute exception qui est jetée hors d'une méthode doit être spécifiée comme telle avec `throws`.
- ⑥ Tout code qui doit absolument être exécuté avant qu'une méthode ne retourne est placé dans un bloc `finally`.

Schéma

Le schéma est donc

```
try {  
    // bloc de code a surveiller  
}  
catch (ExceptionType1 exceptObj) {  
    // gestionnaire d'exception pour ExceptionType1  
}  
catch (ExceptionType2 exceptObj) {  
    // gestionnaire d'exception pour ExceptionType2  
}  
...  
finally {  
    // bloc de code a executer  
    // avant de sortir de la methode  
}
```

Types d'exceptions

- ① Une classe est au sommet de la hiérarchie des exceptions : `Throwable`
- ② Deux sous-classes de `Throwable` :
 - `Exception` : conditions exceptionnelles que les programmes utilisateur devraient traiter.
 - `Error` : exceptions catastrophiques que normalement seul l'environnement d'exécution devrait gérer.
- ③ Une sous-classe d'`Exception`, `RuntimeException`, pour les exceptions de l'environnement d'exécution.

Exceptions non gérées

- 1 Considérons le code suivant où une division par zéro n'est pas gérée par la programme :

```
class ExcepDiv0 {  
    public static void main(String args[]) {  
        int d = 0;  
        int a = 42 / d;  
    }  
}
```

- 2 Lorsque l'environnement d'exécution essaie d'exécuter la division, il construit un nouvel objet exception afin d'arrêter le code et de gérer cette condition d'erreur.
- 3 Le flux de code est alors interrompu et la pile d'appels (des différentes méthodes invoquées) est inspectée en quête d'un gestionnaire d'exceptions.

Exceptions non gérées (suite)

- 1 N'ayant pas fourni de gestionnaire au sein du programme, le gestionnaire par défaut de l'environnement d'exécution se met en route.
- 2 Il affiche la valeur en String de l'exception et la trace de la pile d'appels :

```
/home/mounier> java ExcepDiv0  
java.lang.ArithmeticException: / by zero  
    at ExcepDiv0.main(ExcepDiv0.java:4)
```

Instructions try et catch

- 1 Un bloc `try` est destiné à être protégé, gardé contre toute exception susceptible de survenir.
- 2 Juste derrière un bloc `try`, il faut mettre un bloc `catch` qui sert de gestionnaire d'exception. Le paramètre de l'instruction `catch` indique le type et le nom de l'instance de l'exception gérée.

Instructions try et catch (suite)

```
class ExcepDiv0 {  
    public static void main(String args[]) {  
        try {  
            int d = 0;  
            int a = 42 / d;  
        } catch (ArithmeticException e) {  
            System.out.println("Div par zero");  
        }  
    }  
}
```

- 3 La portée d'un bloc catch est restreinte aux instructions du bloc try immédiatement précédent.

Instructions catch multiples

- 1 On peut gérer plusieurs exceptions à la suite l'une de l'autre.
- 2 Lorsqu'une exception survient, l'environnement d'exécution inspecte les instructions catch les unes après les autres, dans l'ordre où elles ont été écrites.
- 3 Il faut donc mettre les exceptions les plus spécifiques d'abord.

Instruction throw

- 1 Elle permet de générer une exception, *via* un appel de la forme
`throw ThrowableInstance ;`
Cette instance peut être créée par un `new` ou être une instance d'une exception déjà existante.
- 2 Le flux d'exécution est alors stoppé et le bloc `try` immédiatement englobant est inspecté, afin de voir s'il possède une instruction `catch` correspondante à l'instance générée.
- 3 Si ce n'est pas le cas, le 2^{ème} bloc `try` englobant est inspecté ; et ainsi de suite.

Instruction throw (suite)

❶ Exemple

```
class ThrowDemo {  
    static void demoproc() {  
        try {  
            throw new NullPointerException("demo");  
        } catch (NullPointerException e2) {  
            System.out.print("attrapee ds demoproc()");  
            throw e2;  
        }  
    }  
  
    public static void main(String args[]) {  
        try {  
            demoproc();  
        } catch (NullPointerException e1) {  
            System.out.print("attrapee ds main()");  
        }  
    }  
}
```

Instruction throws

- 1 Si une méthode est susceptible de **générer une exception qu'elle ne gère pas, elle doit le spécifier**, de façon que ceux qui l'appellent puissent se prémunir contre l'exception.
- 2 L'instruction `throws` est utilisée pour spécifier la liste des exceptions qu'une méthode est susceptible de générer.
- 3 Pour la plupart des sous-classes d'`Exception`, le compilateur **forcera à déclarer** quels types d'exception peuvent être générées (sinon, le programme ne compile pas).
- 4 Cette règle ne s'applique pas à `Error`, `RuntimeException` ou à leurs sous-classes.

Instruction throws (suite)

- ❶ L'exemple suivant ne compilera pas :

```
class ThrowsDemo1 {  
    static void proc() {  
        System.out.println("dans proc()");  
        throw new IllegalAccessException("demo");  
    }  
  
    public static void main(String args[]) {  
        proc();  
    }  
}
```

Ce programme ne compilera pas :

- parce que `proc()` doit déclarer qu'elle peut générer `IllegalAccessException`;
- parce que `main()` doit avoir un bloc `try/catch` pour gérer l'exception en question.

Instruction throw (suite)

- ❶ L'exemple correct est :

```
class ThrowsDemo1 {  
    static void proc()  
        throws IllegalAccessException {  
        System.out.println("dans proc()");  
        throw new IllegalAccessException("demo");  
    }  
  
    public static void main(String args[]) {  
        try {  
            demoproc();  
        } catch (IllegalAccessException e) {  
            System.out.println(e + "attrapee");  
        }  
    }  
}
```

Instruction finally

- 1 Un bloc `finally` est toujours exécuté, qu'une exception ait été générée ou non. Il est exécuté avant l'instruction suivant le bloc `try` précédent.
- 2 Si le bloc `try` précédent contient un `return`, le bloc `finally` est exécuté avant que la méthode ne retourne.
- 3 Ceci peut être pratique pour fermer des fichiers ouverts et pour libérer diverses ressources.
- 4 Le bloc `finally` est optionnel.

Conclusion

① Le code suivant

```
FileInputStream fis;  
try {  
    fis = new FileInputStream("readme.txt");  
} catch (FileNotFoundException e) {  
    fis = new FileInputStream("default.txt");  
}
```

② est plus propre que

```
#include <sys/errno.h>  
  
int fd;  
fd = open("readme.txt");  
if (fd == -1 && errno == EEXIST)  
    fd = open("default.txt");
```

Classes utilitaires de base

Références :

- **Java et Internet – Concepts et programmation**, G. Roussel, E. Duris, N. Bedon et R. Forax
- **Java in a Nutshell**, D. Flanagan,
- **The Java Language Specification**, J. Gosling, B. Joy, G. Steele

Classes Object, System, PrintStream

- Classe mère `java.Object`
- Classe `java.lang.Object`
- Classe d'entrée-sortie `java.io.PrintStream`

methode()	But
<code>String toString()</code>	Renvoie une vue en chaîne de caractères de <code>this</code> ; par défaut, renvoie le nom de la classe suivi de son code de hachage.
<code>int hashCode()</code>	Renvoie le code de hachage associé à l'objet.
<code>boolean equals()</code>	Teste l'égalité, la plus sémantiquement significative possible .
<code>protected Object clone()</code>	Renvoie une copie superficielle (champ à champ) de l'objet (throws <code>CloneNotSupportedException</code>).
<code>protected void finalize()</code>	Appelée en libération mémoire (throws <code>Throwable</code>).

Méthodes toString(), hashCode()

- 1 toString() : **Forme affichable** de l'objet par `System.out.println()`.
- 2 **La redéfinir** est de bon ton.
- 3 hashCode() : **code de hachage** de l'objet ;
- 4 utilisé dans `java.util.HashMap`.
- 5 Contrat de la méthode hashCode() :
Pour 2 Object, c1 et c2,
 $c1.equals(c2) \Rightarrow c1.hashCode() == c2.hashCode()$
- 6 Donc, si l'on redéfinit equals(), on doit redéfinir également hashCode().

Méthode equals()

- 1 Par défaut, teste l'égalité des références.
- 2 Il est de bon ton de la **redéfinir** en test d'**égalité de contenu**.
- 3 Erreur commune : surcharge au lieu de redéfinition ;
- 4 le paramètre doit être **de type Object**.

Méthode equals() (suite)

- ① Exemple sur des classes de nombres complexes :

```
public class Complexe {  
    protected double partieReelle, partieImaginaire;  
  
    public Complexe(double r, double i) {  
        partieReelle = r;  
        partieImaginaire = i;  
    }  
  
    public boolean equals(Object obj) {  
        if(!(obj instanceof Complexe)) {  
            return false;  
        }  
        Complexe c = (Complexe)obj;  
        return (partieReelle == c.partieReelle &&  
            partieImaginaire == c.partieImaginaire);  
    }  
}
```

Méthode equals() (suite)

- 1 Vérifier que la relation binaire induite est réflexive, symétrique et transitive.
- 2 Vérifier également l'idempotence (plusieurs évaluations de `x.equals(y)` donne toujours le même résultat),
- 3 et que `null` est "absorbant" : `x.equals(null)` est toujours `false`.

Champs et méthodes de la classe System

- ❶ Méthodes et champs utilitaires java.
- ❷ Champs :
 - `static InputStream in` entrée standard (par défaut le clavier)
 - `static PrintStream out` sortie standard (par défaut l'écran)
 - `static PrintStream err` sortie erreur standard (par défaut l'écran)

Champs et méthodes de la classe System (suite)₁₆₄

Méthodes :

methode()	But
<code>static long currentTimeMillis()</code>	renvoie le nombre de millisecondes depuis le 1 ^{er} janvier 1970.
<code>static void exit(int status)</code>	arrête la machine virtuelle java en cours d'exécution.
<code>static void gc()</code>	demande au ramasse-miettes de récupérer la mémoire inutilisée.

❶ Méthodes :

methode()	But
<code>void close()</code>	Ferme le flux d'entrée/sortie
<code>void flush()</code>	Vide le tampon mémoire associé au flux (force l'écriture)
<code>void print(...)</code>	Affiche l'argument sur la sortie standard. Accepte des <code>boolean</code> , <code>char</code> , <code>int</code> , <code>long</code> , <code>float</code> , <code>double</code> , <code>Object</code> et <code>String</code> .
<code>void println(...)</code>	Même effet que <code>print()</code> , mais rajoute un saut de ligne
<code>void write(int b)</code>	Écriture binaire d'un octet sur le flux d'entrée/sortie

Méthode main() et classes d'emballage des types primitifs

- Méthode main() et ses arguments
- classes d'emballage des types primitifs (boolean, int, float, ...)

Méthode main() et ses arguments

- ① Syntaxe
`public static void main(String args[]) ...`
- ② `public` : la méthode peut être appelée de partout
- ③ `static` : pas besoin de créer d'objet pour l'appeler
- ④ `void` : elle ne renvoie rien
- ⑤ `String args[]` : `args` est un tableau de `String`
- ⑥ 1^{ier} argument `args[0]`, 2^{ième} argument `args[1]`, ...

Méthode main() et ses arguments (suite)

- 1 Nombre d'arguments : `args.length`
- 2 Attention ! Ne pas confondre
 - le champ `length` : nombre d'éléments d'un tableau
 - la méthode `length()` de la classe `String` : longueur de la chaîne de caractères
- 3 Exemple d'affichage des arguments de la ligne de commande ainsi que de leur longueur :

```
class TestMain {  
    public static void main(String args[]) {  
        for(int i = 0; i < args.length; i++)  
            System.out.println("arg no " + i+1 +  
                               " : " + arg[i] +  
                               " de longueur : " +  
                               args[i].length());  
    }  
}
```


Méthode main() et ses arguments (suite)

- 1 Par un appel dans une fenêtre Dos (resp. une fenêtre terminal Unix/Linux) de la forme

```
java TestMain toto 4 gabuzomeu 7.8 +&)
```

affiche

```
arg no 1 : toto de longueur : 4
```

```
arg no 2 : 4 de longueur : 1
```

```
arg no 3 : gabuzomeu de longueur : 9
```

```
arg no 4 : 7.8 de longueur : 3
```

```
arg no 4 : +&) de longueur : 3
```

Liste des classes d'emballage

- ❶ Permettent de disposer de méthodes utilitaires de manipulation des types primitifs.
- ❷ Héritent de la classe abstraite `Number`.
- ❸ Les classes d'emballage des types primitifs sont : `Boolean`, `Byte`, `Character`, `Short`, `Integer`, `Long`, `Float` et `Double`.
- ❹ Méthode `xxxValue()`, où `xxx` est l'un des noms de type primitif correspondant ;
- ❺ elle permet d'obtenir une variable du type primitif correspondant.

```
Integer un = new Integer(1);  
int i      = un.intValue();
```

- ❻ Méthode `parseXXX(String)` où `XXX` est l'un des noms de classe précédent ;

Liste des classes d'emballage (suite)

- 1 elle permet d'obtenir un objet de type numérique ou booléen à partir d'une chaîne de caractères.
- 2 Par ex. `parseDouble("2.5")` ; renvoie un `Double`.
- 3 L'inverse est réalisé par `toString()`.
- 4 Les constantes `MIN_VALUE` et `MAX_VALUE` contiennent les valeurs minimale et maximale.

Scanner (java.util.Scanner)

Classe Scanner : aperçu

- 1 La classe Scanner permet entre autres l'entrée facile de types primitifs et de String au clavier.
- 2 Il suffit de créer un objet Scanner avec en argument le flux à lire,
- 3 puis d'appeler une méthode nextXXX() selon le type primitif XXX à lire
- 4 Exemple d'entrée d'un entier au clavier :

```
Scanner sc = new Scanner(System.in); // Creation d'un Scanner
                                         // le flux System.in (1
int i = sc.nextInt();                  // prise d'un entier sur
                                         // (au clavier)
```

Classe Scanner : constructeurs

Différents constructeurs sont disponibles

methode()	But
Scanner(File source)	Construit un objet de type Scanner produisant des valeurs à partir du fichier spécifié.
Scanner(InputStream source)	Construit un objet de type Scanner produisant des valeurs à partir du flux d'entrée spécifié.
Scanner(String source)	Construit un objet de type Scanner produisant des valeurs à partir de la chaîne spécifiée.

Classe Scanner : méthodes essentielles

- 1 Rappel : un flux d'entrée est composé de lexèmes, ou atomes syntaxiques, qui sont séparés par des délimiteurs.
- 2 Les méthodes boolean `hasNextXXX()` renvoient true si le prochain lexème correspond au type attendu.
- 3 La chaîne XXX précédente est l'une des suivantes : `BigDecimal`, `BigInteger`, `Boolean`, `Byte`, `Double`, `Float`, `Int`, `Long`, `Short`, `Line`
- 4 selon le type attendu, qui sera respectivement `BigDecimal`, `BigInteger`, `boolean`, `byte`, `double`, `float`, `int`, `long`, `short` pour les 9 premières, et une nouvelle ligne pour la dernière.
- 5 Ainsi, `hasNextInt()` renvoie true si le prochain lexème est un `int`.
- 6 Les méthodes `YYY nextXXX()` renvoient la valeur du prochain lexème selon le type correspondant à la chaîne XXX

Classe Scanner : méthodes essentielles (suite)

- 7 Ainsi, `int nextInt()` renvoie le prochain `int`, `String nextLine()` renvoie la prochaine ligne, `int nextDouble()` renvoie le prochain double, etc.
- 8 La méthode boolean `hasNext()` renvoie `true` s'il y a un prochain lexème.
- 9 La méthode `String next()` renvoie le prochain lexème disponible.

Classe Scanner : Exemples

❶ Exemple de lecture dans un fichier :

```
Scanner sc = new Scanner(new File("myNumbers"));  
while (sc.hasNextLong()) {  
    long aLong = sc.nextLong();  
}
```

Exemple d'un cercle

Exemple d'une classe Cercle avec utilisation d'un Scanner

```
import java.util.Scanner;
/**
 * Classe representant un cercle
 */
class Cercle {
    // champs : rayon du cercle
    double r;
    // Constructeur : initialisation des champs
    Cercle(double nouvRayon) {
        r = nouvRayon;
    }
    // methode de calcul d'une surface
    double calculeSurface() {
```

Exemple d'un cercle (suite)

```
        return(3.1416*r*r);
    }
} // fin de class Cercle

/**
 * Ce programme affiche la surface d'un cercle dont
 * l'utilisateur entre le rayon
 */
public class CercleMain {
    // methode main() : point d'entree du programme
    public static void main(String[] args) {
        // pour les entrees de donnees au clavier
        Scanner entreeClavier = new Scanner(System.in);
        // capture d'un double au clavier
```

Exemple d'un cercle (suite)

```
double rayon = entreeClavier.nextDouble();  
// creation d'un objet de type Cercle  
Cercle monCercle = new Cercle(rayon);  
// calcul de sa surface  
surface = monCercle.calculerSurface();  
// affichage du resultat  
System.out.println("Voici la surface du cercle" +  
                    "de rayon " + monCercle.r +  
                    " : " + surface);  
}  
} // fin de class CercleMain
```

Classes `java.applet.Applet` et `java.lang.String`

- Notion d'applet, méthodes et exemple
- Classe `String`
- Classe `StringBuffer`

Notion d'applet

- 1 Applet : mini-application, dont le code est téléchargé à travers le réseau.
- 2 Est visualisée par un navigateur ou par un visualiseur d'applets ("applet viewer").
- 3 Diverses restrictions de sécurité.

Notion d'applet (suite)

- ④ Une applet n'a pas de méthode `main()`.
- ⑤ On étend la classe `java.Applet`, en redéfinissant diverses méthodes.
- ⑥ Une applet n'est pas sous le contrôle de l'activité (thread) d'exécution : elle répond lorsque le navigateur le lui demande.
- ⑦ Donc, pour des tâches longues, l'applet doit créer sa propre activité.

Méthodes à redéfinir

Méthodes de base d'Applet :

- ❶ `void init()`
Appelée lors du premier chargement de l'applet. Utilisée pour des initialisations, de préférence à un constructeur.
- ❷ `void destroy()`
Appelée lors du déchargement de l'applet. Utilisée pour libérer des ressources.
- ❸ `void start()`
Appelée lorsque l'applet devient visible. Souvent utilisée avec des animations et des activités (threads).
- ❹ `void stop()`
Appelée lorsque l'applet est masquée.

Méthodes à redéfinir (suite)

Une méthode héritée de Container :

```
public void paint(Graphics g)
```

que le navigateur appelle pour demander à l'applet sa mise à jour graphique.

Autres méthodes d'Applet :

- 1 `String getAppletInfo()`
Pour obtenir des informations à propos de l'applet
- 2 `String[] [] getParameterInfo()`
Description des paramètres de l'applet.

Méthodes à redéfinir (suite)

- ① `AudioClip getAudioClip(URL url)`
Renvoie une référence à une instance d'objet de type `AudioClip`.
- ② `void play(URL url)`
joue l'`AudioClip` spécifié à l'adresse `url`.
- ③ `Image getImage(URL url)`
Renvoie une référence à une instance d'objet de type `Image`.

Exemple : un disque coloré

Classe Disk : surface circulaire colorée

```
import java.awt.*;

public class Disk {
    protected int      x, y;           // position du disque
    protected int      size;           // diametre du disque
    protected Color     color;          // couleur du disque

    public Disk(int Xpos, int Ypos, int radius) {
        x      = Xpos;    y  = Ypos;
        size   = radius;
        color  = Color.red;           // Initialement rouge
    }

    // methodes fixant des attributs (modificateurs)
    public void setXY(int newX, int newY) { x = newX; y = newY;}
    public void setSize(int newSize)      { size = newSize; }
    public void setColor(Color newColor)  { color = newColor;}
```

Exemple : un disque coloré (suite)

```
// methodes accedant aux attributs (accesseurs)
public int getX()                { return x; }
public int getY()                { return y; }
public int getSize()             { return size; }
public Color getColor()          { return color; }

// Afficher le disque
public void paint(Graphics g) {
    g.setColor(color);
    g.fillOval(x-(size/2), y-(size/2), size, size);
}

} // public class Disk
```

Classe DiskField, qui affiche le disque précédent :

Exemple : un disque coloré (suite)

```
import java.applet.*;
import java.awt.*;

public class DiskField extends Applet {

    int    x = 150, y = 50, size = 100;    // position et diametre
    Disk    theDisk = null;

    // Initialisation de l'applet
    public void init() {
        theDisk = new Disk(x, y, size);  }

    // Dessiner le disque
    public void paint(Graphics g) {
        // Demander au navigateur d'appeler la methode paint()
        // pour afficher le disque
        theDisk.paint(g);
    }
}
```

Exemple : un disque coloré (suite)

```
public void start() { ; }  
public void stop() { ; }
```

```
}// class DiskField
```

Pour afficher l'applet, on a besoin d'un fichier HTML qui la référence.


```
<APPLET code="DiskField.class" width=150 height=100>  
</APPLET>
```

Construction de String

- 1 Dans `java.lang` : `String` pour les chaînes à immuables et `StringBuffer` pour celles qui sont modifiables.
- 2 `String` et `StringBuffer` sont déclarées `final`, de façon à réaliser certaines optimisations.
- 3 Constructeur de recopie `public String(String original)`.
- 4 Il y a une syntaxe spéciale pour les chaînes qui permet une création-initialisation rapide :

```
String s = "abc";  
System.out.println(s.length());  
System.out.println("abcdef".length());
```

Les 2 dernières lignes vont afficher respectivement 3 et 6.

- 5  Ne pas confondre la méthode `length()` avec la variable d'instance `length` de références à des tableaux.

Concaténation de chaînes

- 1 Pas de surcharge, à part +.
- 2 Par exemple :

```
String s = "Impossible " + "d'eternuer " +  
           "les yeux ouverts.";
```

est nettement plus lisible que

```
String s = new StringBuffer("Impossible ")  
           .append("d'eternuer ")  
           .append("les yeux ouverts.")  
           .toString();
```

qui est exactement ce qui se passe lorsque le code est exécuté.

Conversion de chaînes

- 1 `append()` appelle en fait la méthode `valueOf()`. Pour des types primitifs, cette dernière renvoie une représentation en chaîne.
- 2 Pour des objets, elle appelle la méthode `toString()` de l'objet.
- 3 `toString()` est une méthode de `Object`, donc tout objet en hérite.

Conversion de chaînes (suite)

C'est une BONNE PRATIQUE que de redéfinir toString() pour ses propres classes. Exemple

```
class Point {
    int x, y;
    Point(int x, int y) {
        this.x = x;
        this.y = y;
    }
    public String toString() {
        return "Point[" + x + "," + y + "]";
    }
}
class toStringDemo {
    public static void main(String args[]) {
        Point p = new Point(20, 20);
        System.out.println("p = " + p);
    }
}
```

Extraction/Comparaison

- 1 `charAt()` permet d'extraire un caractère. Par ex.
`"abc".charAt(1)` renvoie `'b'`.
- 2 `startsWith()` (resp. `endsWith()`) teste si la chaîne appelante commence (resp. finit) par la chaîne fournie en paramètre.
- 3 `"Nabuchodonosor".endsWith("nosor")` et
- 4 `"Nabuchodonosor".startsWith("Nabu")` renvoient tous deux `true`.
- 5 On peut également spécifier l'indice de début de comparaison.
- 6 Par exemple l'expression
`"HoueNeng".startsWith("Neng", 5)`
renvoie `true`.

Égalité

- ① `equals()` teste l'égalité caractère à caractère.
- ② `==` teste l'égalité des références (des adresses mémoires, ou pointeurs) pour voir si elles se réfèrent à la même instance.

Relation d'ordre

- ❶ `compareTo()` compare 2 `String` selon un ordre alphabétique.
- ❷ `int compareTo(String s)` renvoie un résultat négatif si la chaîne appelante est inférieure à `s` (le paramètre), 0 si elles sont égales et un résultat positif sinon.

Recherche de sous-chaîne

- 1 Recherche de l'indice d'occurrence d'un caractère ou d'une sous-chaîne dans une chaîne.
- 2

```
int indexOf(int car);  
int lastIndexOf(int car);
```

renvoient l'indice de la première (resp. la dernière) occurrence (c.à.d. apparition) du caractère car.

Modifications sur une copie de String

- 1 Puisque les String sont immuables, pour modifier une chaîne, on peut soit utiliser un StringBuffer ou utiliser l'une des méthodes suivantes, qui fournissent une copie modifiée d'une String.
- 2 substring() extrait une String d'une autre. Par exemple :
"Bonjour a tous".substring(8) -> "a tous"
"Bonjour a tous".substring(6, 5) -> "r a t"

Modifications sur une copie de String (suite)

- 1 `concat()` crée un nouvel objet, la concaténation de la chaîne appelante et du paramètre :
`"Bonjour".concat(" a tous") -> "Bonjour a tous"`
- 2 `replace(char carSrc, char carDst)` remplace toutes les occurrences de `carSrc` par `carDst` :
`"Bonjour".replace('o', 'a') -> "Bajaur"`

Modifications sur une copie de String (suite)

- 1 toLowerCase() et toUpperCase() : conversion en majuscules (resp. minuscules)

"Grenouille".toUpperCase() -> "GRENOUILLE"

"BOEuf".toLowerCase() -> "bouef"

- 2 trim() enlève les espaces avant et après :

" J'ai besoin d'air ".trim() ->

"J'ai besoin d'air"

methode()	But
<code>String concat(String str)</code>	Concaténation de <code>this</code> à celle fournie en argument.
<code>boolean contains(String s)</code>	renvoie <code>true</code> si <code>this</code> contient la <code>String</code> argument.
<code>boolean contentEquals(StringBuffer sb)</code>	renvoie <code>true</code> si <code>this</code> est égale (au sens du contenu) à la <code>StringBuffer</code> argument.
<code>static String copyValueOf(char[] data)</code>	Conversion d'un tableau de caractères en <code>String</code> .
<code>static String format(String format, Object... args)</code>	Renvoie une <code>String</code> formatée (voir la documentation des API pour les chaînes format).
<code>boolean matches(String regex)</code>	renvoie <code>true</code> si <code>this</code> correspond à l'expression régulière <code>regex</code> .
<code>String[] split(String regex)</code>	Découpe <code>this</code> selon les délimiteurs fournis en tant qu'expression régulière.

StringBuffer

- ① C'est une chaîne modifiable et susceptible de croître et de décroître.
- ② `charAt()` renvoie un caractère à un indice donné;
`setCharAt()` remplace un caractère à un indice donné
- ③ `getChars()` fonctionne de la même manière que son homologue de `String`. Prototype identique :

```
void getChars(int srcBegin, int srcEnd,  
              char [] dst, int dstBegin);
```

- ④ `append()` concatène le paramètre à la chaîne appelante. En général appelé *via* `+`.
- ⑤ `insert()` insère une sous-chaîne à un indice spécifié :
`"L'envie d'être roi".insert(8, "de tout sauf ")`
résulte en
`"L'envie de tout sauf d'être roi"`

StringBuilder

- ❶ C'est une chaîne modifiable ayant les même fonctionnalités que `StringBuffer` mais sans synchronisation multi-threads.
- ❷ Il est conseillé de l'utiliser pour les applications mono-thread.

java.util : Conteneurs et autres utilitaires

Références :

- Java et Internet – Concepts et programmation,
G. Roussel, E. Duris, N. Bedon et R. Forax
- Data Structures & Problem Solving Using Java,
M.A. Weiss
- Algorithms,
R. Sedgewick

Classes de java.util ; Classes et interfaces de comparaison

- Contenu du paquetage java.util
- Interface java.lang.Comparable
- Interface java.util.Comparator

Classes et interfaces de java.util

On trouve les groupes de classes suivants :

- Comparaison sur des objets (interfaces Comparable et Comparator).
- Structures de données conteneurs (listes chaînées, arbres, tables de hachage).
- Expressions régulières (paquetage java.util.regex).
- Classe Date, gestion de la date.
- Classe EventObject

Classes et interfaces de java.util (suite)

- Classes Timer et TimerTask
- Classe Observable, super classe des objets observables.
- Classe Random, générateur de nombres pseudo-aléatoires.
- Classe Stack, pile d'objets.
- Classe StringTokenizer, lorsqu'instanciée avec un objet String, casse la chaîne en unités lexicales séparées par n'importe quel caractère.
- Journalisation (paquetage java.util.logging).

Classes et interfaces de java.util (suite)

- Stockage de paramètres (paquetage java.util.prefs).
- Classe BitSet, ensemble de bits arbitrairement grand.
- Des classes de gestion de zone géographique, de gestion des fuseaux horaires, de gestion du calendrier.

Classes et interfaces de java.util (suite)

- On trouve diverses interfaces :
 - 10 interfaces associées aux conteneurs.
 - `EventListener`, interface marqueur pour tous les gestionnaires d'événements.
 - `Comparator`, pour les objets définissant une relation d'ordre (via `compare(Object o1, Object o2)` et `equals(Object o)`).
 - `Observer`, définit la méthode `update()` nécessaire pour qu'un objet "observe" des sous-classes de `Observable`.

Interface `java.lang.Comparable`

- 1 Deux éléments sont comparables (implanter `Comparable`) si l'on peut leur appliquer
`public int compareTo(Object other)`
- 2 Cette méthode renvoie la distance entre `this` et `other`, au sens de la relation d'ordre induite.

Interface `java.util.Comparator`

- 1 Objets comparateurs : spécialisés dans la définition de relations d'ordre.
- 2 Deux méthodes à implanter :
 - `int compare(Object o1, Object o2)`, offrant le même service que `compareTo()` de `java.lang.Comparable`
 - `boolean equals(Object o)` testant l'égalité de contenu.
- 3 Les méthodes de comparaison doivent en général être compatibles avec le test d'égalité.

Classes et interfaces conteneurs

- Catégories
- Transitions entre conteneurs
- Interfaces Collection et Map
- Arrays
- Itération

Cadre de collections

Un cadre logiciel de collections est formé de

- ❶ Interfaces, ou types de données abstraits.
- ❷ Implantations, classes concrètes (structures de données réutilisables).
- ❸ Algorithmes, méthodes utilitaires comme tri ou recherche, polymorphes (fonctionnalités réutilisable).

Catégories de conteneurs

- ① Deux grandes catégories :
 - Type (interface) `Collection`, ou groupe d'objets.
 - Type (interface) `Map`, table d'association de couples clé-valeur.
- ② Dans `Collection`, deux sous-catégories :
 - Type `Set`, ne pouvant contenir 2 fois le même élément.
 - Type `List`, éléments indicés par des entiers positifs.
- ③ Dans `Map`, l'objet clé permet d'accéder à l'objet valeur.
- ④ Dans `Map`, couple clé-valeur : entrée, de type `Map.entry`.

Différents types de conteneurs, selon l'interface et la structure de donnée.

		Implantations			
		Table de hachage	Tableau à taille variable	Arbre équilibré	Liste chaînée
Interfaces	Set	HashSet		TreeSet	
	List		ArrayList		LinkedList
	Map	HashMap		TreeMap	

Transitions entre conteneurs

1 Dans Map :

- `values()` renvoie une `Collection` des valeurs de la table
- `keySet()` renvoie un `Set` des clés de la table
- `entrySet()` renvoie un `Map.entry` des entrées (paires clés/valeur) de la table
- Ce sont des **vues** de la table.
- Une modification d'une vue est faite sur la table et vice versa.

2 Dans Collection :

Transitions entre conteneurs (suite)

- `toArray()` renvoie un tableau contenant tous les objets de la collection.
 - Ce n'est pas une vue qui est renvoyée.
- ③ Dans la classe utilitaire `Arrays` :
- `toArray()` renvoie un tableau contenant tous les objets de la collection.
 - Ce n'est pas une vue qui est renvoyée.

Interface Collection

Résumé des méthodes :

- ① `boolean add(Object o)`
ajoute l'élément spécifié à la collection. renvoie `true` si la collection a été modifiée par l'opération (un `Set` ne peut contenir 2 fois le même élément).
- ② `boolean contains(Object o)`
teste si la collection contient `o`
- ③ `boolean equals(Object o)`
teste l'égalité de contenu de la collection avec `o`.

Interface Collection (suite)

- ④ `Iterator iterator()`
renvoie un itérateur sur les éléments de la collection.
- ⑤ `boolean remove(Object o)`
enlève une instance de `o` de la collection.
- ⑥ `int size()`
renvoie le nombre d'éléments de la collection.
- ⑦ `Object[] toArray()`
renvoie un tableau contenant tous les éléments de la collection.

Interface Map

Résumé des méthodes :

- ① `boolean containsKey(Object key)`
teste si la table contient une entrée avec la clé spécifiée.
- ② `boolean containsValue(Object value)`
teste si la table contient une entrée avec la valeur spécifiée.
- ③ `boolean equals(Object o)`
teste l'égalité de contenu de la table avec o.
- ④ `Object get(Object key)`
renvoie la valeur de la table correspondant à la clé key.

Interface Map (suite)

- 5 `Object put(Object key, Object value)`
associe la valeur `value` à la clé `key` dans la table. Si une valeur était déjà associée, la nouvelle remplace l'ancienne et une référence vers la nouvelle est renvoyée, sinon `null` est renvoyé.
- 6 `Object remove(Object key)`
enlève l'entrée associée à `key` de la table. Renvoie une référence sur la valeur retirée ou `null` si elle n'est pas présente.
- 7 `int size()`
renvoie le nombre d'entrées (paires clé-valeur) de la table.

Classe Arrays

- ❶ Classe de manipulation de tableaux.
- ❷ Méthode static List `asList`(Object[] a) renvoie une vue de type List de a.
- ❸ Sinon, 4 groupes de méthodes principales (en tout 54 méthodes) :
 - Dans ce qui suit, Type désigne soit un type primitif, soit Object.
 - static int `binarySearch`(Type[] a, Type key) effectuant une recherche de key dans a.
 - static int `equals`(Type[] a, Type[] b) teste l'égalité élt. à élt. de a et b.
 - static int `fill`(Type[] a, Type val) affecte tous les éléments de a à val.
 - static int `sort`(Type[] a) trie a selon un algorithme quicksort modifié.

Conteneurs immuables

- ❶ Toutes les méthodes de modification de `Collection` et `Map` sont documentées comme optionnelles.
- ❷ On doit les redéfinir, mais le code peut juste lever une `UnsupportedException`.
- ❸ Si toutes ces méthodes lèvent une telle exception, le conteneur est dit immuable.
- ❹ Vues immuables d'un conteneur :
 - `static Collection unmodifiableCollection(Collection c),`
 - `static List unmodifiableList(List list),`
 - `static Map unmodifiableMap(Map m),`
 - `static Set unmodifiableSet(Set s),`

Concurrence et synchronisation

- 1 La classe `Collections` contient des méthodes renvoyant des vues synchronisées :
 - `static Collection synchronizedCollection(Collection c),`
 - `static List synchronizedList(List list),`
 - `static Map synchronizedMap(Map m),`
 - `static Set synchronizedSet(Set s).`

Itération de conteneurs

- ❶ Par le biais de l'interface `Iterator`.
- ❷ Elle définit des méthodes par lesquelles on peut énumérer (un à la fois) des éléments d'une collection.
- ❸ Elle spécifie 3 méthodes :
 - `boolean hasNext()`
renvoie `true` s'il y a encore des éléments dans la collection,
 - `Object next()`
renvoie une référence sur l'instance suivante de la collection,
 - `remove()`
enlève l'élément renvoyé dernièrement par l'itérateur.

Itération de conteneurs (suite)

- ④ Un exemple typique d'itération est la boucle for suivante

```
static void filter(Collection c) {  
    for (Iterator i = c.iterator(); i.hasNext();)   
        if (!cond(i.next()))  
            i.remove();  
}
```

Noter que ce code est polymorphe (il fonctionne pour toute instance de Collection)

Squelettes d'implantation

- 1 Des classes abstraites squelettes facilitent l'implantation.
- 2 Les opérations (méthodes) de modification ne font rien sauf générer une exception de type `UnsupportedOperationException`.
- 3 Par ex., pour créer une classe immuable de type `Collection`, il suffit d'hériter de `AbstractCollection` et d'implanter `Iterator iterator()` et `int size()`

Squelettes d'implantation (suite)

- ④ Pour définir des conteneurs modifiables, il faut implanter `boolean add(Object o)` et la méthode `boolean remove(Object o)` de l'itérateur renvoyé par `Iterator iterator()`.
- ⑤ De la même manière, on dispose des classes `AbstractMap`, `AbstractSet`, `AbstractList` et `AbstractSequentialList`.

Conteneurs de type Map

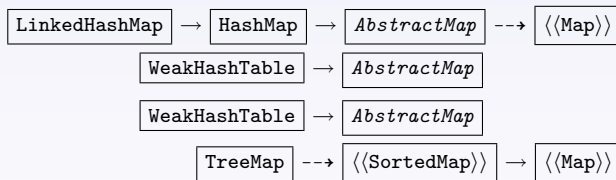
- Types de conteneurs
- Hachage

Conteneurs de type Map

- 1 Map est une interface qui représente un mécanisme de stockage clé/valeur.
- 2 Une **clé** est un nom que l'on utilise pour accéder à une **valeur**.
- 3 Il s'agit d'une représentation abstraite d'un **tableau associatif**.
- 4 Les couples (clé, valeur) sont des instances de classes implantant l'interface Map.entry.

Conteneurs de type Map (suite)

- ❶ La hiérarchie des classes est la suivante (\rightarrow : hérite de, $--\rightarrow$: implante) :



- ❷ les classes `WeakHashMap`, `HashMap`, `LinkedHashMap` et `IdentityHashMap` utilisent des tables de hachage.
- ❸ `TreeMap` utilise des arbres rouges-noirs.

Classe HashMap

- ① HashMap est la plus utilisée des Map en pratique.
- ② **Table de hachage** : une représentation d'une clé est utilisée pour déterminer une valeur autant que possible unique, nommée **code de hachage**
- ③ Le code hachage est alors utilisé comme indice auquel les données associées à la clé sont stockées.

Classe HashMap (suite)

- ❶ Pour utiliser une table de hachage :
 - On fournit un objet utilisé comme clé et des données que l'on souhaite voir liées à cette clé.
 - La clé est hachée.
 - Le code de hachage résultant est utilisé comme indice auquel les données sont stockées dans la table.
- ❷ Les valeurs de codes de hachage sont cachées (encapsulées).
- ❸ Une table de hachage ne peut stocker que des clés qui redéfinissent les méthodes `hashCode()` et `equals()` de `Object`.

Classe HashMap (suite)

- 1 La méthode `hashCode()` doit calculer le code de hachage de l'objet et le renvoyer.
- 2 `equals()` compare 2 objets.
- 3 Beaucoup de classes courantes de Java implantent la méthode `hashCode()`. C'est le cas de `String`, souvent utilisée comme clé.

Classe HashMap (suite)

① Les constructeurs de HashMap sont :

- `HashMap()`, constructeur par défaut, construit une table de hachage vide.
- `HashMap(int capaciteInitiale)`, construit une table de hachage de taille initiale `capaciteInitiale`.
- `HashMap(int capaciteInitiale, float tauxCharge)`, construit une table de hachage de taille initiale `capaciteInitiale` et de taux de remplissage `tauxCharge`; ce taux, nécessairement compris entre 0.0 et 1.0, détermine à quel pourcentage de remplissage la table sera re-hachée en une plus grande.

Si `tauxCharge` n'est pas spécifié, 0.75 est utilisé.

Classe HashMap (suite)

- 1 Si l'on veut utiliser ses propres classes comme clé de hachage, il faut redéfinir `hashCode()` et `equals()` de `Object`.
- 2 La valeur (`int`) renvoyée par `hashCode()` est ensuite automatiquement réduite par une opération modulo la taille de la table de hachage.
- 3 Il faut s'assurer que la fonction de hachage utilisée répartit aussi uniformément que possible les valeurs renvoyées entre 0 et `capaciteInitiale`, la taille initiale de la table.

① Méthodes de HashMap :

methode()	But
<code>boolean containsKey(Object key)</code>	Renvoie <code>true</code> s'il existe une clé égale à <code>key</code> .
<code>boolean containsValue(Object value)</code>	Renvoie <code>true</code> s'il existe une valeur égale à <code>value</code> .
<code>Object get(Object key)</code>	Renvoie une référence sur l'objet contenant la valeur associée à la clé <code>key</code> ou <code>null</code> .
<code>Object put(Object key, Object value)</code>	Insère une clé et sa valeur dans la table. Renvoie <code>null</code> si <code>key</code> n'est pas déjà dedans, ou la valeur précédente associée à <code>key</code> sinon.
<code>Object remove(Object key)</code>	Enlève la clé <code>key</code> et sa valeur. Renvoie la valeur associée à <code>key</code> ou <code>null</code> .

- 1 Une **fonction de hachage** est une fonction $f : x \rightarrow h$ aisément calculable, qui transforme une très longue entrée x en une sortie h nettement plus courte, (typiquement de 10^6 bits à 200 bits) et qui a la propriété suivante :
- 2 (Phach) : Il n'est pas "calculatoirement faisable" de trouver deux entrées différentes x et x' telles que $f(x) = f(x')$.
- 3 L'expression "l'opération \mathcal{O} n'est pas calculatoirement faisable" signifie simplement tous les algorithmes actuellement connus pour réaliser \mathcal{O} sont de complexité exponentielle.

Hachage : authentification

Application à l'authentification de messages :

- 1 Supposons que Alice veuille envoyer un message à Bob, en signant son message.
- 2 Les données qu'Alice veut transmettre sont constituées d'un message en clair suivi de ses prénom et nom, en clair, à la fin du message. Nommons x cet ensemble de données.
- 3 Alice transmet alors x , en clair, suivi de $h = f(x)$ où f est une fonction de hachage.
- 4 À la réception, Bob applique la fonction de hachage f au texte en clair x et le compare à h .

Hachage : authentification (suite)

- 5 Ainsi, Bob peut vérifier non seulement que le message provient bien d'Alice (que sa signature n'a pas été falsifiée), mais également que son message, en clair, n'a pas été altéré.
- 6 Par supposition, aucun pirate n'aurait été capable de modifier x sans changer la valeur de $h = f(x)$.

- 1 Application à la recherche. Supposons avoir une clé de recherche relativement longue (un entier ou une chaîne de caractères).
- 2 La sortie de la table de hachage sera un indice d'une table dans laquelle sont rangées les valeurs associées aux différentes clés.
- 3 Prenons le cas où la clé est une chaîne de caractères x et où la fonction de hachage f choisie la transforme en $h = f(x)$ un indice entre 1 et p (il y a p indices différents dans la table de hachage).

Hachage : recherche (suite)

- 1 La propriété (Phach) assure que les sorties de f sont quasi-uniformément distribuées, en un sens probabiliste, dans $[1, p]$.
- 2 Prenons comme exemple de fonction de hachage simple la fonction modulo un nombre premier.
- 3 Prenons alors pour p un nombre premier (par exemple 101) et considérons la clé suivante :
VERYLONGKEY

- 1 On décompose la clé selon la base de son alphabet (ici, il y a 32 signes dans l'alphabet considéré) :

$$22.32^{10} + 5.32^9 + 18.32^8 + 25.32^7 + 12.32^6 + 15.32^5 + 14.32^4 + 7.32^3 + 11.32^2 + 5.32 + 25$$

- 2 La fonction de hachage considérée ne prend pas directement ce nombre pour en faire l'opération modulo 101, sa représentation machine étant lourde à manier ; il s'écrit en effet en binaire par

1011000101100101100101100011110111000111010110010111001

- 3 Il est bien plus efficace de se servir de la représentation d'un polynôme par l'algorithme de Hörner, où VERYLONGKEY s'écrit, en base 32, de la façon suivante :

$$((((((((((22.32+5)32+18)32+25)32+12)32+15)32+14)32+7)32+11)32+5)32+25$$

Hachage : recherche (suite)

- 1 L'algorithme de calcul de la fonction de hachage est alors

```
public final int hache(String cle, int tailleTable)
{
    int valHach = 0;

    h = cle.charAt(0);
    for(int i = 1; i < cle.length(); i++)
        valHach = ((valHach*32)+cle.charAt(i)) % tailleTable;

    return valHach;
}
```

où cle est une String dans lequel on a stocké la clé.

Hachage : recherche (suite)

- 1 Pour `p == 101` et `cle[]` valant "VERYLONGKEY", cette fonction de hachage fournit 97.
- 2 🦋 Le calcul d'un indice à partir d'une clé est rapide, mais rien ne garantit que 2 clés distinctes donneront des indices distincts.
- 3 On nomme *collision* d'indice le fait que 2 clés distinctes donnent le même indice.
- 4 Il faut alors une stratégie de *résolution de collision*.

Hachage : recherche (suite)

- 1 Une stratégie simple et efficace est le **chaînage séparé**. À chaque fois qu'il y a une collision pour l'indice i , les clés sont rangées dans une liste chaînée n° i , associée à la case d'indice i de la table. Les différents éléments de la liste chaînée peuvent être rangés en ordre alphabétique croissant des clés, pour un accès plus rapide.
- 2 Cette stratégie est bien adaptée au cas où l'on ne connaît pas, *a priori*, le nombre d'enregistrements (de paires clés/valeurs) à traiter, ce qui est le cas de la classe `HashMap` de Java.

Hachage : recherche (suite)

- 1 En Java, un code de hachage est généré (*via* la méthode `hashCode()` définie dans la classe `Object`. Elle renvoie alors en général une conversion de l'adresse de l'objet en `int`, bien que ceci ne soit pas une obligation d'implantation du langage.
- 2 La méthode `hashCode()` est redéfinie par les types suivants : `BitSet`, `Boolean`, `Character`, `Date`, `Double`, `File`, `Float`, `Integer`, `Long`, `Object` et `String`,

Hachage : recherche (suite)

- ① Pour `String`, le code est obtenu de l'une des 2 manières suivantes, selon sa longueur. Soit n la longueur de la suite de caractères et c_i le caractère d'indice i .
- Si $n \leq 15$, le code de hachage est calculé par

$$\sum_{i=0}^{n-1} c_i \cdot 37^i$$

en utilisant l'arithmétique des `int`

Hachage : recherche (suite)

- Si $n > 15$, le code de hachage est calculé par

$$\sum_{i=0}^m c_{i.k} \cdot 39^i$$

en utilisant l'arithmétique des `int`, où $k = \lfloor \frac{n}{8} \rfloor$ et $m = \lceil \frac{n}{k} \rceil$, ne prenant (dans la décomposition) que 8 ou 9 caractères de la chaîne.

Itération d'une HashMap

- ① L'opération d'itération sur une HashMap est possible (via `values()`), mais présente 2 inconvénients :
 - (1) L'ordre d'itération est indéterminé.
 - (2) La complexité de l'itération est linéaire en la **capacité** de la table.
 - Pour un conteneur adapté à l'itération, c'est une fonction linéaire de la **taille** du conteneur.

Classe LinkedHashMap

- 1 Pour corriger les insuffisances en itération de `HashMap`, le conteneur contient une table de hachage ainsi qu'une liste doublement chaînée de ses éléments.
- 2 L'itération est ainsi de complexité linéaire en la taille de la table.
- 3 L'ordre d'itération est celui d'insertion des clés.

Classe WeakHashMap

- 1 Se comporte comme `HashMap`, mais les clés sont des références faibles.
- 2 Les clés qui ne sont référencées que par la table sont susceptibles d'être détruites par le ramasse-miettes pour libérer de la mémoire.

Interface SortedMap

- ❶ Implantation de Map dans laquelle les entrées peuvent être ordonnées suivant les clés.
- ❷ Il faut fournir 2 constructeurs supplémentaires :
 - l'un avec un paramètre de type SortedMap, réalisant une copie de la table fournie, avec le même ordre.
 - l'autre avec un paramètre de type Comparator fixant l'ordre.

Interface SortedMap (suite)

① Méthodes supplémentaires de SortedMap

- `Comparator comparator()`
renvoie le comparateur associé à la table triée, ou `null` s'il utilise l'ordre naturel des clés.
- `Object firstKey()`
renvoie la plus petite clé de la table triée.
- `SortedMap headMap(Object toKey)`
renvoie une vue de la partie de la table triée dont les clés sont strictement plus petites que `toKey`.

Interface SortedMap (suite)

- `Object lastKey()`
renvoie la plus grande clé de la table triée.
- `SortedMap subMap(Object fromKey, Object toKey)`
renvoie une vue de la partie de la table triée dont les clés sont comprises strictement entre `fromKey` et `toKey`.
- `SortedMap tailMap(Object fromKey)`
renvoie une vue de la partie de la table triée dont les clés sont strictement plus grandes que `fromKey`.

Classe TreeMap

- 1 Implante l'interface SortedMap.
- 2 structure de données sous-jacente : arbres rouges-noirs.
- 3 Les clés sont constamment ordonnées en ordre croissant,
- 4 selon l'ordre naturel des clés, ou selon un objet de comparaison fourni à la création, selon le constructeur utilisé.
- 5 Temps d'accès en insertion, recherche, suppression (`containsKey()`, `get(...)`, `put(...)` et `remove(...)`) en $\mathcal{O}(\log n)$ où n est la taille du conteneur.

Conteneurs de type Collection et Listes

- Suites à accès direct et séquentiel
- Tableau dynamique ArrayList
- Listes chaînées LinkedList

Conteneurs de type Collection

Conventions :

- indentation : héritage,
- <<interface>> ,
- [[classe abstraite]]

<<Collection>>

<<List>>

[[AbstractCollection]] (implements Collection)

[[AbstractList]] (implements List)

[[AbstractSequentialList]]

LinkedList (implements List)

ArrayList (implements List, RandomAccess)

[[AbstractSet]] (implements Set)

HashSet (implements Set)

LinkedHashSet (implements Set)

TreeSet (implements SortedSet)

Ensembles et itération des listes

- Ensembles : classes `HashSet`, `LinkedHashSet` et `TreeSet`.
- Itération des listes : `ListIterator`, méthodes en plus pour les élts. précédents.

Suites

- Suites à accès direct (dans n'importe quel ordre) : implantent `RandomAccess`.
- Suites à accès séquentiel (pour accéder à $i + 1$, accéder à i d'abord) : héritent de `AbstractSequentialList`.
- Suites à accès direct : `ArrayList`.
- Suites à accès séquentiel : `LinkedList`.

Suites à accès direct

- Un telle suite possède une capacité initiale.
- S'il ne reste plus de place, il faut augmenter la taille de la liste, opération en $\mathcal{O}(n)$.
- L'insertion a donc une complexité au pire de $\mathcal{O}(n)$.
- En augmentant la taille astucieusement, on assure que la complexité de i insertions est en $\mathcal{O}(i)$.
- Le calcul de la taille, l'accès à un élément et l'affectation à une position donnée a une complexité en $\mathcal{O}(1)$
- La suppression est en $\mathcal{O}(n)$.

Classe ArrayList

- ① ArrayList est, grossièrement parlant, un tableau à longueur variable de références à des objets.
- ② ArrayList n'est pas synchronisée par défaut. Si l'on désire avoir un tableau à longueur variable synchronisé, utiliser `static Collection synchronizedCollection(Collection c)` de la classe Collections.
- ③ On dispose de 3 constructeurs :
 - `ArrayList()` crée une liste de taille initiale 10 références.
 - `ArrayList(int size)` crée une liste de taille initiale `size` références.
 - `ArrayList(Collection c)` crée une liste avec les élts de `c`. La capacité initiale de la liste est de 110% celle de `c`.

Classe ArrayList (suite)

264

Les différentes méthodes sont :

methode()	But
<code>void add(int index, Object element)</code>	L'objet spécifié par <code>element</code> est ajouté à l'endroit spécifié de la liste.
<code>boolean contains(Object element)</code>	Renvoie <code>true</code> si <code>element</code> est contenu dans la liste et <code>false</code> sinon.
<code>Object get(int index)</code>	renvoie l'élément situé à la position spécifiée de la liste.
<code>final int indexOf(Object element)</code>	Renvoie l'indice de la 1 ^{ière} occurrence de <code>element</code> . S'il n'y est pas, <code>-1</code> est renvoyé.
<code>Object remove(Object element)</code>	Enlève la première occurrence de <code>element</code> trouvée dans la liste. Renvoie une référence sur l'élément enlevé.
<code>Object set(int index, Object element)</code>	Remplace l'élément à la position spécifiée par <code>element</code> .

Classe ArrayList (suite)

Exemple :

```
import java.util.ArrayList;
import java.util.Iterator;

class DemoArrayList {
    public static void main(String args[]) {
        // Taille initiale de 3
        ArrayList l = new ArrayList(3);

        System.out.println("Taille initiale : " + l.size());

        l.add(new Integer(1));
        l.add(new Integer(2));
        l.add(new Integer(3));
        l.add(new Integer(4));
        l.add(new Double(18.23));
        l.add(new Integer(5));
        System.out.println("1er element : " +
```

Classe ArrayList (suite) (suite)

```
                (Integer)l.get(0));  
System.out.println("Dernier element : " +  
                (Integer)l.get(l.size()-1));  
  
if (l.contains(new Integer(3)))  
    System.out.println("l contient l'entier 3");  
  
// Listons les elements de la liste  
ListIterator it = l.listIterator();  
  
System.out.println("\n Elements dans la liste : ");  
while (it.hasNext())  
    System.out.print(it.next() + " ");  
System.out.println();  
}  
}
```

La sortie écran du programme est :

Classe ArrayList (suite) (suite)

```
Taille initiale : 3  
1er element : 1  
Dernier element : 5  
l contient l'entier 3  
Elements dans la liste :  
1 2 3 4 18.23 5
```

Suites à accès séquentiel

- Classe `LinkedList`, de structure sous-jacente une liste doublement chaînée.
- Ajout/suppression en début de liste en temps constant ($\mathcal{O}(1)$).
- Insertion/suppression d'un élt. juste après un élt. donné (par ex. par un itérateur) en temps constant.
- Accès à l'élt. i en $\mathcal{O}(i)$.

Méthodes supplémentaires de LinkedList

methode()	But
<code>void addFirst(Object o)</code>	insère l'élémt. spécifié au début de la liste.
<code>void addLast(Object o)</code>	ajoute l'élémt. spécifié à la fin de la liste.
<code>Object getFirst()</code>	renvoie le 1er élmt. de la liste.
<code>Object getLast()</code>	renvoie le dernier élmt. de la liste.
<code>Object removeFirst()</code>	enlève et renvoie le 1er élmt. de la liste.
<code>Object removeLast()</code>	enlève et renvoie le dernier élmt. de la liste.

Classe Stack

- 1 Stack implante une pile (file LIFO, Last In/First Out) standard.
- 2 Stack est une sous classe de Vector. Elle hérite donc de toutes les méthodes de Vector, et en définit certaines qui lui sont propres.
- 3 \Rightarrow Stack n'est pas une pile au sens puriste du terme ...
- 4 Il est toutefois utile d'avoir accès aux méthodes de Vector.
- 5 Un seul constructeur, sans arguments, qui crée une pile vide.

❶ Méthodes propres de Stack :

methode()	But
<code>boolean empty()</code>	Renvoie <code>true</code> si la pile est vide et <code>false</code> sinon.
<code>Object peek()</code>	Renvoie l'élément du dessus de la pile, mais ne l'enlève pas.
<code>Object pop()</code>	Renvoie l'élément du dessus de la pile, en l'enlevant.
<code>Object push(Object element)</code>	Pousse <code>element</code> sur la pile. <code>element</code> est également renvoyé.
<code>int search(Object element)</code>	Cherche <code>element</code> dans la pile. S'il est trouvé, son offset par rapport au dessus de la pile est renvoyé. Sinon, <code>-1</code> est renvoyé.

- ❷ Une `EmptyStackException` est jetée si l'on appelle `pop()` lorsque la pile est vide.

```
// For a set or list
for(Iterator it=collection.iterator(); it.hasNext(); ) {
    Object element = it.next(); }
```

```
// For keys of a map
for(Iterator it=map.keySet().iterator(); it.hasNext(); ) {
    Object key = it.next(); }
```

```
// For values of a map
for(Iterator it=map.values().iterator(); it.hasNext(); ) {
    Object value = it.next(); }
```

```
// For both the keys and values of a map
for(Iterator it=map.entrySet().iterator(); it.hasNext(); ) {
    Map.Entry entry = (Map.Entry)it.next();
    Object key = entry.getKey();
    Object value = entry.getValue(); }
```



```
LinkedList stack = new LinkedList();

// Push on top of stack
stack.addFirst(object);

// Pop off top of stack
Object o = stack.getFirst();

// If the queue is to be used by multiple threads,
// the queue must be wrapped with code to synchronize the methods
stack = (LinkedList)Collections.synchronizedList(stack);
```

Exemple : création d'une table de hachage

```
// Create a hash table
Map map = new HashMap();      // hash table
map = new TreeMap();          // sorted map

// Add key/value pairs to the map
map.put("a", new Integer(1));
map.put("b", new Integer(2));
map.put("c", new Integer(3));

// Get number of entries in map
int size = map.size();        // 2

// Adding an entry whose key exists in the map causes
// the new value to replace the old value
Object oldValue = map.put("a", new Integer(9)); // 1

// Remove an entry from the map and
// return the value of the removed entry
oldValue = map.remove("c");    // 3
```

Exemple : itération de la table de hachage

```
// Iterate over the keys in the map
Iterator it = map.keySet().iterator();
while (it.hasNext()) {
    // Get key
    Object key = it.next();
}

// Iterate over the values in the map
it = map.values().iterator();
while (it.hasNext()) {
    // Get value
    Object value = it.next();
}
```